Higher Order Termination

Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems

This is a copy of the published thesis, which includes minor updates and corrections. For the original, please see http://dare.ubvu.vu.nl/bitstream/handle/1871/39346/dissertation.pdf?sequence=1 or contact the author for a physical copy.

Changes in this work compared to the published version:

- In the definition of β-reduced sub-meta-term (Def. 6.18), the accidentally omitted variable case was added.
- In the definitions of formative reduction and tagged formative reduction, a case for β-reduction was included; this case was always intended, and omitted only by accident. The corresponding proofs have been updated accordingly.
- Two reference items which were *To Appear* at the time of writing have appeared now, and have been updated.

Cynthia Kop

Higher Order Termination

Cover design by Melle Wynia
Printed and bound by Wöhrmann Print Service, Zutphen
ISBN: 978-94-6203-164-7
IPA Dissertation Series 2012-14

The work reported in this thesis has been carried out at the Vrije Universiteit Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was funded by the Netherlands Scientific Organisation (NWO).

VRIJE UNIVERSITEIT

Higher Order Termination

Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 13 november 2012 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Cynthia Louisa Martina Kop

geboren te Tilburg, Nederland

promotor:	prof.dr. J.W. Klop
copromotor:	dr. F. van Raamsdonk

Acknowledgements

To start off this thesis, I would like to express my gratitude to the many people who have had an influence on this work, or on my PhD period.

First, and above all, I would like to thank Femke van Raamsdonk, my supervisor and regular co-author. For tons of useful feedback. For patiently helping me turn my collections of proofs into decent papers. For giving me directions, but leaving me all the freedom I needed. A supervisor can make or break your PhD period, and Femke was the best anyone could wish for.

I also thank Jan Willem Klop, my promotor, who regularly dispensed good advice in our weekly meetings, checked my papers, pointed out flaws and was overall a great help. Moreover, I thank him and Vincent van Oostrom for all the suggestions for good terminology.

I would also like to say thank you to Kristoffer Rose, who was my mentor during my internship at IBM. Both for the many whiteboard sessions, where we enthusiastically determined features of the CRSX formalism, the complete willingness to incarnate new ideas and the good planning, and for all the help while I was there, including picking me up from the train station in the middle of the night.

Another very important person to mention is Carsten Fuhs. Apart from being a great friend to wander around foreign cities with, he has also been an invaluable help in my work, both as a co-author and a colleague in the field of term rewriting. In our many discussions on msn and in real life, Carsten has helped me refine ideas, suggested new directions, pointed out useful literature, warned me where my terminology deviated from the standard and indicated flaws or difficulties in my work. It was also he who convinced me to use SAT-encodings for my termination tool WANDA.

Originally I was not planning to write a tool as part of my PhD, although afterwards I am very happy that I did. For the encouragement to do so, I thank Aart Middeldorp. I also want to thank Cristina Borralleras and Albert Rubio, my opponents in the annual termination competition, who gave me a good motivation to improve WANDA.

For the lay-out of this thesis, I want to thank Rena Bakhshi, my colleague and for a long time office mate, who gave me her LaTeX style files. Also, Melle Wynia, my neighbour and friend, who expertly designed the cover.

Over the years, my colleagues at the VU have provided a very pleasant atmo-

sphere to work in, both with lively lunch discussions and by always being willing to help out with questions such as “how do I do this in LaTeX” or “how can I declare travel costs”: Jörg Endrullis, Helle Hansen, Dimitri Hendriks, Wan Fokkink, Clemens Grabmayer, Jan Willem Klop, Vincent van Oostrom, Andrew Polonsky, Femke van Raamsdonk, Stefan Vijzelaar, Roel de Vrijer and David Williams. I also want to express my gratitude to the patient members of the IT helpdesk, who helped me get my laptop up to scratch (and to set up a replacement) every few months when it broke again.

Then, I want to thank the many other people, both in the Netherlands and abroad, who made PhD life so much fun. The members of IPA research school, with many of whom I've hung around for hours either in a restaurant, swimming pool or entertainment park, discussing research, life or more mundane topics. On the risk of missing some important names: Alexandra, David, Felienne, Frank, Jeroen, Joost, Marijn, Mark, Meivan, Michel, Michiel, Paul, Pedro, Pieter, Sjoerd, Stephanie. Also the people in Marktoberdorf summer school, who inspired me both in researchy ways, and with whom I climbed mountains, ate ice cream or did the algorithm march: Arnar, Carsten, Eduardo, Josef and Willard. And then there are others, such as Marc, Carsten Otto and Thomas, with whom I socialised at conferences and twice a year at the term rewriting seminar.

Of course I should not forget my non-research friends either, who regularly provided a healthy distraction from thesis writing or computer science research. Freek, Ivo and Melle, who would often come for tea or dinner, or the occasional movie. Alex, Luuk and Ton, for dungeons and dragons, and Agnetha, Mirella and Ryanne, for parties and visits. And of course the community of Discworld MUD, who also had a more direct influence by sometimes helping me find the right English words.

Last but not least, I would like to thank my family, who have been very patient and supportive, and who have borne admirably with my sudden transformation into a workaholic in the last few months.

Thank you, everyone!

Contents

Introduction 1

Or, Why are we doing all this?

Term rewriting systems play an important role in many areas of computer science. In essence, they provide an abstract way to define algorithms. The theory is simple: *terms*, expressions over a fixed set of symbols, are rewritten according to a (usually finite) set of *rewrite rules*. The usage is widespread. There are applications of term rewriting in logic, program analysis, security, compiler building, automated theorem proving, process algebra and many more fields. Figure 1.1 lists some extracts of term rewriting systems used in practice.

$\mathtt{f}_S(\mathtt{TL}(\mathtt{null}, o_9), \mathtt{TL}(\mathtt{null}, o_9), o_6)$ $\Rightarrow \mathtt{f}_S(\mathtt{TL}(\mathtt{null}, o_9), o_9, o_6)$ (a) Part of a Java bytecode analysis [103]	$\mathtt{dec}(\mathtt{enc}(x, \mathtt{pk}(y)), \mathtt{sk}(y)) = x$ $\nu s.(\overline{a}\langle \mathtt{pk}(s)\rangle \mid b(x).\overline{c}\langle \mathtt{dec}(x, \mathtt{sk}(s))\rangle)$ (b) Asymmetric encryption in the applied π-calculus [1]
$\mathtt{Trans}(\mathtt{Refl}, x_1) \Rightarrow x_1$ $\mathtt{Trans}(x_1, \mathtt{Refl}) \Rightarrow x_1$ $\mathtt{CongrO1}(\mathtt{Refl}) \Rightarrow \mathtt{Refl}$ (c) Propositional proof reduction [124]	`map :: (a → b) → [a] → [b]` `map f [] = []` `map f (x:xs) = f x : map f xs` (d) Part of a Haskell module
$\mathtt{C}[\mathtt{Let}[\mathtt{E}_1, x.\mathtt{E}_2[x]]] \rightarrow$ $(C[\mathtt{E}_1, r_1]; C[\mathtt{E}_2[r_1], r];);$ $\mathtt{C}[\mathtt{Var}[r_1], r_2] \rightarrow (\mathtt{MOVE}[r_1, r_2];);$ (e) Register allocation in CRSX	$pX \Rightarrow^a pBX$ $pX \Rightarrow^c q$ $qB \Rightarrow^b q$ (f) A push-down automaton [100]

Figure 1.1: Examples of term rewriting systems used in practice

As a consequence, the properties of term rewriting systems have been well-studied. The main topics of research can roughly be divided into two categories: *confluence* and *normalisation*. The question of confluence is whether evaluation order matters: if s reduces both to t and q, is there a term u which both t and q reduce to? The question of normalisation is whether reduction sequences can be assumed to be finite. A system is *weakly normalising* if any term can be reduced to a *normal form* which cannot be reduced any further. A system is *strongly normalising*, or *terminating*, if any reduction sequence is finite. The areas of con-

fluence and normalisation are closely related to each other, for instance because the combination of confluence and weak normalisation leads to *unique normal forms*, and the combination of strong normalisation and local confluence (which is often easier to prove than confluence) implies full confluence (see e.g. [118]).

As the title suggests, the focus of this thesis is on *termination*. Apart from its use in the study of confluence, the study of termination has many practical and theoretical applications. Obviously it is a good property for a program to reach a result eventually, regardless of user input. In software for hospitals or aeroplane control, lives may depend on this. In a cryptography module, security may depend on this. On the theoretical side, a strongly normalising rewrite relation yields an induction principle.

Figure 1.1(d) shows part of a Haskell module. The variable `f`, a function, is recursively applied on the elements of a list. This is typical for *higher-order rewriting*. Higher-order term rewriting combines standard, first-order term rewriting with notions from the (simply-typed) λ-calculus. Using higher-order term rewriting we can represent "function pointers", and, consequently, handle a variety of problems which are difficult to treat natively in the first-order setting. The study of higher-order term rewriting also has various applications. For example, higher-order rewriting formalisms yield a natural model of functional programming languages. They are also used for defining compiler specifications of XQuery [108], and form the underlying model of theorem provers like Coq and Isabelle.

This thesis is devoted to the study of termination in higher-order term rewriting. Termination analysis forms an important part of the general study of higher-order rewriting, and plays a role in all of the applications mentioned above. In particular, this work aims to contribute to several projects:

- furthering the understanding of higher-order rewriting, by providing a common framework and considering ways to translate termination results between formalisms;
- investigating the possibilities as well as the limitations of general termination analysis for higher-order rewriting;
- strengthening the potential power of automated theorem provers such as Coq, by making it possible to automatically verify validity (which requires termination) of user-given functions;
- enabling the quick use of termination results by laymen, by providing an automatic tool.

Scepticism – Objections and Counterarguments. The study of higher-order termination provides some challenges which are absent in first-order rewriting, in particular due to the presence of β-reduction. In the higher-order setting, a rewriting step at the root might create a redex deep inside a term. Moreover,

β-reduction may lead to term explosion; a single step may quadratically increase the size of a term.

Consequently, it is often not deemed worthwhile to use higher-order term rewriting; a system that lends itself naturally to higher-order rewriting is instead modelled as a first-order system and approached with first-order techniques. An example is the study of termination of Haskell programs [44], where a termination problem in Haskell is transformed into a first-order term rewriting system.

The most common alternative for higher-order term rewriting systems are *applicative* systems. In such systems, types and function variables (which can be thought of as function pointers) are present, but binders are omitted. When we are only interested in termination of fixed terms, or terms of a given form, this is a fine model. But not when we want to prove termination of *all* terms, including terms with λ-abstractions. Higher-order term rewriting systems can often be converted into an applicative system. However, termination of the resulting system only implies termination of terms without binders in the original system. This is discussed in a bit more detail in Section 3.1.

Thus, using applicative systems we can more easily study termination of specific terms, or termination of all terms without binders. This is sufficient for some purposes, but not for all. For example, when a number of rewriting rules are given in an automated theorem prover, it is perfectly likely that user input will contain binders. When writing a compiler specification in the CRSX-language, the user-defined program can be seen as a higher-order term. In general, when studying termination of a module of code which can be seen as a higher-order term rewriting system, it is advantageous to know that *all* terms terminate, rather than just the purely applicative ones (although when the former does not apply, or cannot easily be derived, the latter may still be of use).

Another question is whether weak normalisation is not sufficient. Is it truly necessary that *all* reduction sequences terminate, when weak normalisation already guarantees that a result exists? In practice, however, weak normalisation is usually not enough; knowing that a result exists is nice, but it is much preferable to be able to *find* that result. Alternatively, termination using a specific rewriting strategy is often studied: innermost, outermost, outermost-fair, lazy, weak... Consider for example the following program, which is evidently not terminating (due to the rule for `first`):

$$\begin{aligned} \texttt{take}(0, y) &\Rightarrow \texttt{nil} \\ \texttt{take}(\texttt{s}(n), x : y) &\Rightarrow x : \texttt{take}(n, y) \\ \texttt{first}(n, m) &\Rightarrow \texttt{take}(n, m : \texttt{first}(n, \texttt{s}(m))) \end{aligned}$$

Using a *lazy* evaluation strategy like in Haskell, no infinite reduction is possible.

In this thesis I will sometimes consider a strategy, but most of the results target the case where no strategy is used. Since strong normalisation implies weak normalisation, as well as normalisation using any strategy, the results can be applied to functional programs in any language which corresponds to higher-order

term rewriting, whether the language uses lazy evaluation, innermost (which is common in imperative languages) or something else. Additionally, if we consider the evaluation strategy as part of the language, it may well be possible to encode it in the rewrite rules (as is done in [44]).

As a final observation, although it is perhaps easier to study termination behaviour in a restricted setting, it is far from impossible to obtain strong results in the general case. Let us not be intimidated then, and rise to the challenge!

I hope this thesis will go some way towards demonstrating that the additional challenges of the higher-order setting over the first-order one *can* be tackled, and that strong termination techniques are in reach. Many first-order results extend directly, sometimes in more than one way, to the higher-order case. Other results, specific to higher-order rewriting, can be derived as well. Of course we will run into limitations, things we typically cannot do (for instance because systems of a certain form are usually not terminating) – in which case we know where dedicated methods for termination analysis with a strategy, or for another restricted setting, have a task!

Existing Results. Termination analysis for higher-order rewriting has been an area of research for more than two decades now, although split over many different formalisms (as discussed in Chapter 3.1). Thus, this thesis has a solid groundwork to build on. The results can be categorised into a number of groups:

General Schema / Computability Closure The *general schema* is a principle for defining recursive functions. In the first definition [60], an AFS follows the general schema if it can be expressed as a system where function symbols are added iteratively to the language, and their rules either follow a certain recursive scheme, or only use previously defined symbols.

In later definitions [18, 61], the recursion scheme is given by a *computability closure*, and uses *inductive types*, which leads to the notion of *accessible subterms*. Extending the result to other formalisms, the general schema is presented for both HRSs and CRSs in [106], and for a generalised formalism (which is very similar to the formalism used in this thesis) in [13]. It is further improved in [18], and extended to the calculus of constructions in [15]. In [17] various definitions of the computability closure are discussed; plain, with inductive types, with matching modulo β/η and with matching modulo an equational theory.

Recursive Path Ordering Meanwhile, the recursive path ordering known from first-order termination analysis [27, 30] has been extended to the higher-order setting in a long line of research. The recursive path ordering is a reduction ordering: a well-founded, stable and monotonic relation $>$. Termination is proved by showing that the rewrite relation is included in $>$ (which can be done by orienting all rules).

In early definitions, higher-order terms (either in the HRS or AFS formalisms) are *translated* to first-order terms. Since β-reduction might cause problems, the thus induced ordering is very weak, so needs to be extended with cases to deal with sub-constraints $g(\vec{l}) \gtrsim F(\vec{t})$ with F a variable. In [94, 95], this is done using an *extended subterm relation*. The authors of [97] deal with the higher-order aspect using the notions of *dominating a free variable* and *critical positions*; a weight function is used to relate a functional term to a term headed by a free variable. In [62] a (rather restrictive) type ordering is introduced to deal with variable-headed subterms.

Later definitions of the higher-order recursive path ordering are defined directly on higher-order terms. The first of these is available in [63], where two versions of the recursive path ordering for polymorphic AFSs are defined: a basic version (which is extended to HRSs and CRSs in [106] and formalised in Coq in [81]), and a version which includes a *computability closure*. The journal edition of this paper, [64], includes a type ordering and accessibility relation, effectively including the general schema into the recursive path ordering. This definition is further extended to the *computability path ordering* in [19]. The computability path ordering does away with the computability closure, instead posing fewer type restrictions on the relation, and is therefore significantly simpler to use. However, this later definition does not use polymorphic types.

Weakly Monotonic Algebras Another reduction ordering for higher-order rewriting is given by *weakly monotonic algebras*, built on the notion of monotonic algebras in first-order rewriting. This is explored in particular in [105], for the HRS-formalism. A more detailed study, and extension to formalisms with a more sophisticated type system, appears in [104].

First-order polynomial interpretations (an instance of the monotonic algebra approach) are combined with the higher-order recursive path ordering in a recent paper. [22].

Dependency Pairs The recursion scheme and reduction orderings discussed so far exhibit some fundamental weaknesses; in particular, that they can only prove termination of so-called *simply terminating* systems. In the first-order setting, the common way to avoid this weakness is to use *dependency pairs*. Using dependency pairs, the termination question can be reduced to a series of constraints, by considering the shape of rules. The resulting constraints can then be solved by other techniques, such as a recursive path ordering or monotonic algebra.

In the higher-order setting, two different generalisations of the dependency pair approach exist, both for the formalism of HRSs: the *dynamic dependency pair approach* [112] and the *static dependency pair approach* [87, 111, 114]. A more elaborate overview of dependency pair results is given in Chapter 6.1.

Semantic Path Ordering A second approach to reduce the termination question to a series of constraints (and handle systems which are not simply terminating) is the *monotonic higher-order semantic path ordering* [24, 25], based on a first-order method in [66]. This method generalises the higher-order recursive path ordering, by considering a well-founded ordering on terms rather than a precedence on function symbols. To find a suitable well-founded ordering is the second challenge, for which methods like weakly monotonic algebra or recursive path ordering may be used.

Type-based Analysis In type-based analysis (also called *size-based* analysis) function symbols are assigned a dependent type. The "size" of arguments which a function symbol can take, as well as its output, are encoded in its type. For instance, the common addition function `add` may have a type $\mathtt{nat}^n \to \mathtt{nat}^m \to \mathtt{nat}^{n+m}$, indicating that given two input arguments of sizes at most n and m respectively, the result has size at most $n + m$.

This idea is first explored in [59], where the authors analyse a small lazy functional language; sized types are used to analyse termination, productivity and memory safety. A different early appearance of sized types for termination analysis is in [50], which proposes an extension of the calculus of constructions. Both [2] and [11] provide a simplification of the ideas in [50], using a language with an easier type system; any expression typable in the language is terminating. Various extensions and variations with different languages and type systems exist, e.g. [3, 128].

These results concern ML-like languages; type-based termination techniques for higher-order term rewriting first appear in [14], where sized types are combined with a computability closure in the calculus of constructions. In [20] a type-based termination criterion is considered for AFSs; a given AFS terminates if it admits a sized typing which satisfies a number of constraints. The author of [109] introduces dependency pairs based on subtypes rather than subterms, an idea which is further explored in [110].

Semantic Labelling In *semantic labelling* a term rewriting system is transformed by adding labels to the function symbols; the labelled system has the same termination behaviour as the original, and may be easier to handle.

For higher-order rewriting, a first definition of semantic labelling appears in [52]. In [21] it is demonstrated that both the first-order definition and the version of [52] can be seen as an instance of type-based termination.

It is worth noting that in this list I restrict attention to styles of higher-order rewriting with both λ-abstractions and higher-order rewrite rules. Results about termination of various λ-calculi (such as [115]) or modularity of first-order rewriting and typed λ-calculus (e.g. [117]), are omitted, as are results concerning applicative rewriting without binders. The list also does not include research focused on specific functional programming languages (for instance [44]).

Automation. Almost without exception, the results in this thesis can be used by a tool, which is demonstrated with the fully automated termination prover WANDA. This connection goes so far that one could argue that this thesis is essentially WANDA's documentation: almost every result given in this thesis has been implemented,[1] and every proof step taken by WANDA is justified by some result presented here.

Thus, WANDA itself should be considered as a fundamental part of this work. WANDA is completely written by me, and competes in the annual termination competition [125] since 2010. In 2011, WANDA won the higher-order category. An in-depth discussion of the features and methods used by WANDA can be found in Chapter 8.

Thesis Outline. First, in Chapter 2, I will give a brief overview of both first-order rewriting and the style of higher-order rewriting used in this thesis, *AFSMs*. This is followed by a number of transformation results, which provide alternative ways to present systems in the AFSM formalism, and also some tricks which we can use in termination proofs. The last part of the chapter explains how reduction pairs can be used to prove termination of AFSMs.

In Chapter 3, I will follow up on this by discussing the most prominent formalisms of higher-order rewriting, and how they relate to the *AFSM* formalism used here. It is somewhat ironic, given that one of the reasons to study higher-order rewriting is to have uniform proofs, that so many different formalisms exist. Nor is it immediately obvious whether results from one formalism carry over, or can be adapted, to another. However, as we will see, these frameworks are not as far apart as may appear at first glance, and all results put forward in this thesis are applicable to many of the common formalisms.

Then, we will move on to reduction orderings, well-founded ordering relations which can be used to prove termination of higher-order term rewriting systems. First, in Chapter 4, we will study *polynomial interpretations*, an application of the weakly monotonic algebra approach defined by van de Pol and Schwichtenberg [104, 105]. The method here aims for simplicity and automatability rather than generality, but can easily be extended using the results from [104].

In Chapter 5 we will consider a new variation of the *higher-order recursive path ordering*, which was originally defined in [63] and extended further in e.g. [19]. The version discussed here takes a different starting point, which leads to a method with very different strengths and weaknesses. As in Chapter 4, the technique is designed for simplicity.

Chapters 6 and 7 treat the *dependency pair approach*, a method for proving termination which involves looking at minimal terms which may lead to an infinite reduction. There are two styles of adapting dependency pairs to higher-order

[1] A notable exception to this claim of everything being implemented are most of the transformations in Chapter 3. This is not because it would be hard (it would not be), but because most formalisms do not have a standard available benchmark database, or even file format. For this reason, their implementation has been postponed.

rewriting. I will focus on the "dynamic" approach, which has been the object of my own studies, but also discuss the "static" approach in some detail. Chapter 6 discusses the basic method, while Chapter 7 considers a number of improvements to strengthen the approach.

Finally, Chapter 8 discusses *automation*. Most importantly, this includes the techniques used for the implementation of the results presented in this work in WANDA. Some methods are generalised from first-order implementation techniques, but there are also several new ideas, adapted for the higher-order definitions. Moreover, this chapter discusses experimental results on the *termination problem database* [126], an independent database used in the annual termination competition [125].

Note: a bullet-point overview of all contributions of this thesis is given in the appendix.

Origin of the Chapters. Many of the results in this work have been independently published, or accepted for publication by the time of writing. Since the results for some of the papers have been spread out a bit over the thesis, I will list here where the individual papers are represented.

[76] *(A Higher-Order Iterative Path Ordering)* This conference paper, written together with Femke van Raamsdonk and presented at LPAR 2008, forms the base for Chapter 5 (which, however, is significantly extended).

[75] *(Simplifying Algebraic Functional Systems)* This conference paper, presented at CAI 2011, is split over Sections 2.3.1, 2.3.2 and 3.4.4. The results presented here are somewhat less general than in the paper, however, because unlike this thesis, the paper considers a polymorphic formalism.

[41] *(Harnessing First Order Termination Provers Using Higher Order Dependency Pairs)* This conference paper, written together with Carsten Fuhs and presented at FroCoS 2011, forms the basis of Section 7.7.

[79] *(Higher Order Dependency Pairs for Algebraic Functional Systems)* This conference paper, written together with Femke van Raamsdonk and presented at RTA 2011, forms the base for Chapter 6 (which, however, is significantly extended), and Section 7.4.

[80] *(Dynamic Dependency Pairs for Algebraic Functional Systems)* This journal paper, written together with Femke van Raamsdonk for the special LMCS issue of RTA 2011, is an extended version of [79] and contains many results detailed in Chapters 6 and 7 (although for a simpler formalism).

[42] *(Polynomial Interpretations for Higher-Order Rewriting)* This conference paper, written together with Carsten Fuhs and presented at RTA 2012, is split over Chapter 4 and Section 8.5.

Algebraic Functional Systems with Meta-variables

2

Or, What is this actually all about?

In this chapter I will present the higher-order term rewriting formalism used in this thesis. This formalism, *Algebraic Functional Systems with Meta-variables* (AFSMs), is designed in such a way that most commonly used (simply-)typed higher-order formalisms can be embedded into it, so the results from this work can be used for all of these formalisms. An automated termination prover like WANDA, which is discussed at length in Chapter 8, could prove termination of any kind of higher-order term rewriting system simply by translating it into an AFSM in an input module. These other formalisms, and their transformations into AFSMs, are discussed in Chapter 3.

Apart from the AFSM formalism itself, this chapter considers a number of transformations on AFSM. These transformations make it possible to present a system in a form that is convenient for a given technique. We will also consider, as preliminaries, the notions of rule removal and reduction pairs.

Chapter Setup. This chapter consists of four parts. Section 2.1 presents the core principles of first-order term rewriting, simple types and the λ-calculus. In Section 2.2 the AFSM-formalism which is used in this work is defined, and in Section 2.3 we will see some transformations on AFSMs which reflect and sometimes preserve termination. This will for instance make it possible to swap between applicative and functional notation, and ignore differences between base types. Finally, Section 2.4 extends the notion of a reduction pair to AFSMs, and lays the basis for ordering-based termination proofs.

The techniques discussed in Sections 2.3.1 and 2.3.2 have previously been published (for the alternative formalism of AFSs) in [75]. Otherwise, this chapter contains both preliminaries, and several new results.

2.1 Core Definitions

Let us first discuss some of the core concepts which underlie (most forms of) higher-order term rewriting. For more details and examples, see e.g. [10, 118].

2.1.1 First-order Term Rewriting

In first-order term rewriting, *terms* are built from an infinite set of variables $\mathcal{V}$ and a signature $\mathcal{F}$ of function symbols f (disjoint from $\mathcal{V}$). Each function symbol is equipped with an arity $n \in \mathbb{N}$ (denoted $ar(f) = n$), by the following grammar:

$$\mathbb{T} ::= x \mid f(\mathbb{T}^n) \quad (x \in \mathcal{V},\ f \in \mathcal{F},\ ar(f) = n)$$

We typically omit empty argument lists in function applications, writing e.g. $\mathtt{0}$ instead of $\mathtt{0}()$.

A *substitution* is a mapping $\gamma = [x_1 := s_1, \ldots, x_n := s_n]$ with finite domain, where all x_i are variables and the s_i are terms. The application $t\gamma$ of a substitution γ to a term t is obtained by replacing each x_i in t by s_i. For example, $f(g(x,y),z)[x := g(z,z), z := h] = f(g(g(z,z),y),h)$. The *domain* of γ, denoted $dom(\gamma)$, is the set $\{x_1, \ldots, x_n\}$.

A *context* is a term with a single occurrence of a special symbol $\Box$ in it, denoted $C[]$ or just C. The term $C[s]$ is $C[]$ with the $\Box$ symbol replaced by s.

A *first-order rewrite rule* (also just called *rewrite rule* or *rule*) is a pair $l \Rightarrow r$ such that all variables occurring in r also occur in l, and l is not a variable. The rewrite relation $\Rightarrow_{\mathcal{R}}$ generated by a set of rules $\mathcal{R}$ is given by the following clause:

$$C[l\gamma] \Rightarrow_{\mathcal{R}} C[r\gamma] \quad (l \Rightarrow r \in \mathcal{R},\ \gamma \text{ a substitution},\ C[] \text{ a context})$$

A first-order term rewriting system (*TRS*) consists of the set of terms over some signature $\mathcal{F}$ together with the relation $\Rightarrow_{\mathcal{R}}$. It is usually specified by the pair $(\mathcal{F}, \mathcal{R})$, or just by $\mathcal{R}$ (in which case $\mathcal{F}$ consists of the symbols occurring in $\mathcal{R}$).

A term s in a given TRS $(\mathcal{F}, \mathcal{R})$ is called *terminating* if there is no infinite reduction $s \Rightarrow_{\mathcal{R}} t_1 \Rightarrow_{\mathcal{R}} t_2 \Rightarrow_{\mathcal{R}} \ldots$. A TRS $(\mathcal{F}, \mathcal{R})$ is called terminating if all terms over $\mathcal{F}$ are terminating.

Example 2.1. Consider the first-order term rewriting system with signature $\mathcal{F} = \{\mathtt{0}, \mathtt{s}, \mathtt{add}\}$, arities $ar(\mathtt{0}) = 0,\ ar(\mathtt{s}) = 1$ and $ar(\mathtt{add}) = 2$, and $\mathcal{R} = \{\mathtt{add}(x, \mathtt{0}) \Rightarrow x, \mathtt{add}(x, \mathtt{s}(y)) \Rightarrow \mathtt{s}(\mathtt{add}(x,y))\}$ (here, the x, y are variables). Then:

$$\begin{array}{rcl} \mathtt{add}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{0})) & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{add}(\mathtt{s}(\mathtt{0}), \mathtt{0})) \\ & \Rightarrow_{\mathcal{R}} & \mathtt{s}(\mathtt{s}(\mathtt{0})) \end{array}$$

This represents a calculation that $1 + 1 = 2$, as might be expected.

2.1.2 Simple Types

Higher-order rewriting commonly adds types and binders to first-order term rewriting. In this work I will only consider *simple types*. We will briefly discuss how the results can be extended to a polymorphic type system in Chapter 9.3.

From a given set $\mathcal{B}$ of *base types*, the set $\mathcal{T}$ of *simple types* (also just called *types*) is constructed by the following grammar:

$$\mathcal{T} ::= \mathcal{B} \mid (\mathcal{T} \to \mathcal{T})$$

A type of the form $\sigma \to \tau$ is *functional*. Simple types are written as $\sigma, \tau, \rho, \alpha$ and base types as ι, κ. The $\to$ associates to the right, and unnecessary parentheses are omitted. Thus, every type can be written in the form $\sigma_1 \to \ldots \to \sigma_n \to \iota$ with ι a base type.

Following [63], a *type declaration* is an expression of the form $[\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau$ with $\sigma_1, \ldots, \sigma_n, \tau \in \mathcal{T}$; such a declaration is said to have arity n. Type declarations are not types, but are used to assign input and output types to function symbols, as we will see later. $\sigma_1, \ldots, \sigma_n$ are *input types* and τ is the *output type*. A type declaration $[] \longrightarrow \tau$ is usually just denoted τ.

The *order* of a type is given by: $order(\iota) = 1$ if $\iota \in \mathcal{B}$, and $order(\sigma \to \tau) = \max(order(\sigma)+1, order(\tau))$. Extending this to type declarations, $order([\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau) = \max(order(\sigma_1) + 1, \ldots, order(\sigma_n) + 1, order(\tau))$.

2.1.3 The λ-calculus

In the λ-calculus, terms are built from a set $\mathcal{V}$ of variables, using λ-abstraction and application, by the following grammar:

$$\mathbb{T} ::= x \mid (\mathbb{T} \cdot \mathbb{T}) \mid (\lambda x.\mathbb{T}) \qquad (x \in \mathcal{V})$$

In a term $\lambda x.s$, the λx construct *binds* all occurrences of the variable x in s. λ-terms are considered modulo renaming of variables bound by an abstraction operator (α-conversion). Thus, for instance $\lambda x.\lambda y.x \cdot y$ and $\lambda y.\lambda z.y \cdot z$ are the same. Consequently, we can always assume variables in a binder to be *fresh*. The set $FV(s)$ of *free variables* of s consists of all variables occurring in s which are not bound by a λ. We say s is *closed* if $FV(s) = \emptyset$. We omit parentheses where possible, considering application left-associative; a term $s \cdot t \cdot q$ should be read as $(s \cdot t) \cdot q$. We also combine abstractions, writing for instance $\lambda xy.s$ for $\lambda x.\lambda y.s$.

As in the case of first-order rewriting, a substitution is a mapping $\gamma = [x_1 := s_1, \ldots, x_n := s_n]$ with finite domain. However, applying a substitution to a term does not affect bound variables. To calculate a substitute $t[\vec{x} := \vec{s}]$, choose a representative of t whose bound variables are not in the domain or range of γ, and then replace all occurrences of some x_i by s_i. For example, $(\lambda x.(y \cdot x))[y := x] = \lambda z.(x \cdot z)$.

On the other hand, a *context* (as before, a term with a single special symbol $\Box$ in it), may capture free variables: if $C = \lambda x.\Box$, then $C[x] = \lambda x.x$.

Terms in the λ-calculus are rewritten using the β-reduction rule:

$$C[(\lambda x.s) \cdot t] \Rightarrow_\beta C[s[x := t]]$$

A *β-redex* is an occurrence of $(\lambda x.s) \cdot t$ in a term. The relation $\Rightarrow_\beta$ is not terminating in general, as is demonstrated by the term $\omega \cdot \omega$, where $\omega = \lambda x.(x \cdot x)$; this term reduces in one step to itself. However, when restricting attention to λ-terms which can be assigned a simple type, termination is guaranteed [115]. That is, assigning a (simple) type to all variables in $\mathcal{V}$ (notation: $x : \sigma \in \mathcal{V}$), a λ-term s is terminating if we can derive $s : \sigma$ for some type σ using the following clauses:

$$\begin{array}{lll} x : \sigma & \text{if} & x : \sigma \in \mathcal{V} \\ s \cdot t : \tau & \text{if} & s : \sigma \to \tau \text{ and } t : \sigma \\ \lambda x.s : \sigma \to \tau & \text{if} & x : \sigma \in \mathcal{V} \text{ and } s : \tau \end{array}$$

A term is in β-normal form if it has no β-redexes, so cannot be further rewritten using $\Rightarrow_\beta$. The $\Rightarrow_\beta$ relation is terminating on all typed terms and has unique normal forms; that is, if $s \Rightarrow_\beta^* t$ and $s \Rightarrow_\beta^* q$ and both t and q are in β-normal form, then $t = q$. This property is preserved even if typed function symbols are added to the term formation. The β-normal form of a term s is denoted $s{\downarrow_\beta}$.

Restricted η-expansion. The relation of *restricted η-expansion* [5], $\hookrightarrow_\eta$, is defined as follows: $C[s] \hookrightarrow_\eta C[\lambda x.(s \cdot x)]$ if $x : \sigma \in \mathcal{V}$, $s : \sigma \to \tau$, and the following conditions are satisfied:

1. x is a fresh variable;
2. s is not an abstraction $\lambda x.s'$;
3. s in $C[s]$ is not the left-hand side of an application $s \cdot t$;

By part 3, the head of a subterm $s \cdot t_1 \cdots t_n$ is not expanded; requirements 2 and 3 together guarantee that $\hookrightarrow_\eta$ does not create β-redexes. As demonstrated in [29], every term s has a unique *η-long form* $s{\uparrow^\eta}$ which can be found by applying $\hookrightarrow_\eta$ until it is no longer possible. The η-long β-normal form of a term s, denoted $s{\updownarrow_\beta^\eta}$, is the unique form that can be found by applying $\Rightarrow_\beta$ and $\hookrightarrow_\eta$ as long as possible.

2.2 Algebraic Functional Systems with Meta-variables

Now we are ready to define the formalism of *Algebraic Functional Systems with Meta-variables*, which will be used in this thesis. This formalism is a direct extension of both Jouannaud's and Okada's Algebraic Functional Systems and Blanqui's Inductive Data Type Systems, both of which will be discussed in Chapter 3. AFSMs are simply typed, and use both application as in the λ-calculus and function construction as in first-order term rewriting, with meta-variables for matching as first introduced in Aczel's Contraction Schemes [4].

Terms and Meta-terms. To start, let us consider the notion of terms and meta-terms. This definition includes both typed λ-terms and TRS-terms.

Definition 2.2. Let a signature $\mathcal{F}$ of function symbols be given, where each symbol is equipped with a type declaration. Let us also have a set $\mathcal{V}$ of variables, each equipped with a type, and a set $\mathcal{M}$ of *meta-variables*, each equipped with a type declaration. The set of meta-terms consists of those expressions s such that $s : \sigma$ can be derived for some type σ with the following clauses:

$$\begin{array}{lllll}
(\texttt{var}) & x : \sigma & \text{if} & x : \sigma \in \mathcal{V} \\
(\texttt{fun}) & f(s_1, \ldots, s_n) : \tau & \text{if} & f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F} \\
 & & & \text{and } s_1 : \sigma_1, \ldots, s_n : \sigma_n \\
(\texttt{abs}) & \lambda x.s : \sigma \to \tau & \text{if} & x : \sigma \in \mathcal{V} \text{ and } s : \tau \\
(\texttt{app}) & s \cdot t : \tau & \text{if} & s : \sigma \to \tau \text{ and } t : \sigma \\
(\texttt{meta}) & Z(s_1, \ldots, s_n) : \tau & \text{if} & Z : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{M} \\
 & & & \text{and } s_1 : \sigma_1, \ldots, s_n : \sigma_n
\end{array}$$

A *term* is a meta-term without meta-variables, so its type can be derived without clause (`meta`). A *meta-variable application*, or shortly *meta-application*, is a meta-term of the form $Z(s_1, \ldots, s_n)$, a *functional meta-term* is a meta-term $f(s_1, \ldots, s_n)$, an *application* is a meta-term $s \cdot t$ and an *abstraction* is a meta-term $\lambda x.s$. Generally, variables are denoted x, y, z, function symbols are denoted f, g, h (or using more suggestive notation), meta-variables are denoted Z, X, Y, F, G and terms and meta-terms are denoted s, t, q, u, v, w.

We say that the *arity* of a function symbol f is n if $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}$; this is shortly denoted $ar(f) = n$. Let $\texttt{head}(s)$ denote the left-most part of an application: $\texttt{head}(t \cdot q) = \texttt{head}(t)$, and $\texttt{head}(s) = s$ if s is not an application.

In a meta-term $\lambda x.s$, the λx construct binds the variable x in s, as in the λ-calculus. Only variables can be bound, not meta-variables, and there are no binders other than λ. Variables which occur in a term are called *free* if they are not bound by some λ. As in the λ-calculus, (meta-)terms are considered modulo renaming of variables bound by an abstraction operator (α-conversion). Let $FV(s)$ be the set of free variables in s and $FMV(s)$ the set of meta-variables of s.

The binary application operator $\cdot$ associates to the left, so a meta-term $s \cdot t \cdot r$ should be read as $(s \cdot t) \cdot r$. We often omit multiple λs, writing just $\lambda x_1 x_2 \ldots x_n.s$ for $\lambda x_1.\lambda x_2 \ldots \lambda x_n.s$. Moreover, we leave out empty argument lists in function and meta-variable applications, writing e.g. `0` instead of `0`().

Comment: The definition of meta-terms in AFSMs is unusual because it has both application (as in HRSs and AFSs) and meta-variables (as in CRSs). Intuitively, meta-variables are used for matching in rules, where variables are used only as binders. Meta-variables replace reasoning modulo β-reduction as used in HRSs. Application is used for explicit β-steps, and to model (partially) applicative systems.

Substitutions. *Substitution* is defined much like in the (typed) λ-calculus, but with special cases to deal with meta-variables: a substitution is a mapping $[a_1 := s_1, \ldots, a_n := s_n]$, where:

- each s_i is a term;
- each a_i is either a variable or a meta-variable;
- if a_i is a variable with $a_i : \sigma_i \in \mathcal{V}$ for some type σ_i, then $s_i : \sigma_i$;
- if a_i is a meta-variable with $a_i : [\sigma_1 \times \ldots \times \sigma_m] \longrightarrow \tau \in \mathcal{M}$, then s_i has the form $\lambda y_1 \ldots y_m.q$ with $q : \tau$ and each y_i a distinct variable of type σ_i; note that the "types" of a_i and s_i are not the same: a_i is a meta-variable with a type declaration, while s_i is a term with a type $\sigma_1 \to \ldots \to \sigma_m \to \tau$.

A substitution replaces the variables and meta-variables in its domain everywhere in the meta-term. When substituting for a meta-variable we additionally β-reduce the result. For example, if x is a variable and Z a meta-variable,
$(x \cdot (\lambda y.\mathtt{s}(y)))[x := \lambda z.(z \cdot a)] = (\lambda z.(z \cdot a)) \cdot (\lambda y.\mathtt{s}(y))$, while
$Z(\lambda y.\mathtt{s}(y))[Z := \lambda z.(z \cdot a)] = (\lambda y.\mathtt{s}(y)) \cdot a$.
Formally, for any meta-term s and substitution $\gamma = [a_1 := s_1, \ldots, a_n := s_n]$ the result $s\gamma$ of applying γ to s is generated by the following clauses:

- $x\gamma = s_i$ if $x = a_i$ is a variable;
- $Z(t_1, \ldots, t_m)\gamma = q[y_1 := t_1\gamma, \ldots, y_m := t_m\gamma]$ if $Z = a_i$ is a meta-variable and $s_i = \lambda y_1 \ldots y_m.q$;
- $f(t_1, \ldots, t_m)\gamma = f(t_1\gamma, \ldots, t_m\gamma)$ for $f \in \mathcal{F}$;
- $(s \cdot t)\gamma = (s\gamma) \cdot (t\gamma)$;
- $(\lambda y.s)\gamma = \lambda y.(s\gamma)$ if y does not occur in domain or range of γ (using α-conversion we can rename y if necessary);
- $y\gamma = y$ if y is a variable but not one of the a_i;
- $Z(s_1, \ldots, s_n)\gamma = Z(s_1\gamma, \ldots, s_n\gamma)$ if Z a meta-variable and Z is not one of the a_i.

Note that substitution is well-defined: the step where a meta-term is replaced uses a second substitution, but this second substitution has no meta-variables in its domain. If a substitution γ is applied to a meta-term s, we usually assume that the domain of γ contains all meta-variables in s.

A *context* is a term C with a single occurrence of a special typed symbol $\Box_\sigma$ occurring in it. For a term $s : \sigma$ the notation $C[s]$ denotes the term obtained by replacing $\Box_\sigma$ in C by s. Placing a term in a context may capture free variables, for instance if $C = \lambda x.\Box_\sigma$, then $C[\mathtt{s}(x)] = \lambda x.\mathtt{s}(x)$. Note that a context, being a

term, may not contain meta-variables. The subterm relation, $s \trianglerighteq t$, indicates that $s = C[t]$ for some context C. The strict subterm relation, $s \triangleright t$, indicates that $s \trianglerighteq t$ and $s \neq t$ (so the context C is not empty).

Parallel to contexts, a *meta-context* is a meta-term C with a single occurrence of a special typed symbol $\Box_\sigma$ occurring in it, which may be filled with a meta-term of type σ. We say s is a sub-metaterm of t if we can write $t = C[s]$ for some meta-context C. Whenever the notation $C[]$ refers to a meta-context rather than a context, this will always be explicitly stated.

A meta-term is a *pattern* if it does not contain any sub-metaterms $Z(s_1, \ldots, s_n) \cdot t$ with Z a meta-variable, nor sub-metaterms $(\lambda x.s) \cdot t$, and in sub-metaterms $Z(s_1, \ldots, s_n)$ with Z a meta-variable, all s_i are distinct bound variables. Patterns will be used as the left-hand sides of rewrite rules.

Rules and Rewriting. A *rewrite rule* (or just *rule*) is a pair $l \Rightarrow r$ of meta-terms such that:

1. l and r have the same type;
2. all meta-variables occurring in r also occur in l;
3. l and r are closed (so contain no free variables);
4. l is a pattern;
5. l has the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ with $f \in \mathcal{F}$ and $m \geq n \geq 0$.

A set of rules $\mathcal{R}$ generates a *rewrite relation* on terms by the following clauses:

`(rule)`	$C[l\gamma]$	$\Rightarrow_\mathcal{R}$	$C[r\gamma]$	if $l \Rightarrow r \in \mathcal{R}$, C a context, γ a substitution, $dom(\gamma) = FMV(l)$
`(beta)`	$C[(\lambda x.s) \cdot t]$	$\Rightarrow_\mathcal{R}$	$C[s[x := t]]$	if C is a context

This rewrite relation is denoted by $\Rightarrow_\mathcal{R}$. Note that by clause `(beta)` the β-reduction relation from the λ-calculus is explicitly included in the rewrite relation, for any set of rules. The notation $\Rightarrow_\beta$ may be used for a rewrite step using this clause. I will sometimes write $s \Rightarrow_{\mathcal{R},top} t$ (or just $s \Rightarrow_{top} t$ if $\mathcal{R}$ is clear from context) to denote a *topmost* step, that is, a step with either clause `(rule)` or `(beta)`, but where C is an empty context $\Box_\sigma$. A *headmost* step, notation $s \Rightarrow_{head} t$, is a reduction at the head of a term, where C has the form $\Box_\sigma \cdot s_1 \cdots s_n$ (with $n \geq 0$).

Note that the left-hand side l of a rewrite rule must be a pattern. This restriction guarantees that the rewrite relation is decidable. In fact, to obtain decidability of the rewrite relation it is not necessary that l has no subterms $(\lambda x.s) \cdot t$ or $Z(\vec{x}) \cdot t$; these restrictions have been included because they make it significantly easier to obtain results. In Section 3.4, where we study AFSs which do not have the pattern restriction, we will see that we do not lose expressivity by posing these limitations.

Comment: The rewrite relation $\Rightarrow_{\mathcal{R}}$ is defined as a relation on *terms* – so not on meta-terms. The meta-variables of the rules themselves cannot be rewritten with $\Rightarrow_{\mathcal{R}}$. We could extend the definition to rewrite also meta-terms – in fact, this will be done in Definition 5.9 – but in such an extension the rewrite relation is not preserved under substitution.

An *Algebraic Functional System with Meta-variables* (AFSM) consists of the set of well-typed terms over some signature $\mathcal{F}$ and some set $\mathcal{V}$ of variables, together with the relation $\Rightarrow_{\mathcal{R}}$ induced by a set of rewrite rules. It is usually specified by the pair $(\mathcal{F}, \mathcal{R})$, or just by $\mathcal{R}$.

An AFSM $(\mathcal{F}, \mathcal{R})$ is *terminating* if all its terms are terminating, that is, if there is no infinite reduction $s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \ldots$ We say an AFSM has *order* n if for all meta-variables $Z : \sigma$ occurring in any rule of the AFSM, the type declaration σ has order $\leq n$. In particular, we will occasionally consider *second-order* AFSMs with some interest, as many common examples are second order. A rule is called *left-linear* if the left-hand side l of the rule is linear. That is, no meta-variable occurs more than once in l. An AFSM is left-linear if all its rules are left-linear.

Example 2.3. The second-order system $(\mathcal{F}_{\mathtt{map}}, \mathcal{R}_{\mathtt{map}})$ has the following signature:

$$\mathcal{F}_{\mathtt{map}} = \left\{ \begin{array}{rcl} \mathtt{nil} & : & \mathtt{list}, \\ \mathtt{cons} & : & [\mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list}, \\ \mathtt{map} & : & [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{list}] \longrightarrow \mathtt{list} \\ \mathtt{0} & : & \mathtt{nat} \\ \mathtt{s} & : & [\mathtt{nat}] \longrightarrow \mathtt{nat} \end{array} \right\}$$

and it has two rules:

$$\mathcal{R}_{\mathtt{map}} = \left\{ \begin{array}{lcl} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y)) \end{array} \right\}$$

An example reduction in this system:

$$\begin{array}{ll} \mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{s}(\mathtt{0}), \mathtt{nil}))) & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}(\mathtt{s}(\mathtt{0}), \mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{cons}(\mathtt{s}(\mathtt{0}), \mathtt{nil}))) & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}(\mathtt{s}(\mathtt{0}), \mathtt{cons}(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{nil}))) & \Rightarrow_{\mathcal{R}} \\ \mathtt{cons}(\mathtt{s}(\mathtt{0}), \mathtt{cons}(\mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{nil})) & \end{array}$$

This reduction uses both rules. For example in the first step, we use the second rule with a substitution $[F := \lambda x.\mathtt{s}(x), X := \mathtt{s}(\mathtt{0}), Y := \mathtt{nil}]$. In the last step we use the first rule. This reduction does not use any β-steps.

Apart from the full rewrite relation, we could investigate termination of rewriting with a certain *reduction strategy*. A reduction strategy essentially defines a subset of the rewrite relation. Common reduction strategies are, for example, innermost ($s \Rightarrow_{\mathcal{R},\mathrm{innermost}} t$ if either $s \Rightarrow_{\mathcal{R},top} t$ and the direct subterms of s cannot be reduced with $\Rightarrow_{\mathcal{R}}$, or a direct subterm of s reduces innermost), outermost

($s \Rightarrow_{\mathcal{R},\text{outermost}} t$ if either $s \Rightarrow_{\mathcal{R},top} t$ or s cannot be top-reduced and one of its direct subterms reduces outermost) and beta-first ($s \Rightarrow_{\mathcal{R},\text{beta-first}} t$ if either $s \Rightarrow_\beta t$ or $s \Rightarrow_{\mathcal{R}} t$ and s is β-normal). In this thesis we will not really study termination using a strategy (this is implied by full termination anyway), but strategies sometimes appear naturally, for instance in the transformations of Chapter 3.

Restricted η-expansion. Finally, since we will often use it, let us extend the definition of restricted η-expansion to deal with meta-variables. Define $C[s] \hookrightarrow_\eta C[\lambda x.s \cdot x]$ for a *meta-context* C if $s : \sigma \to \tau$ and the following conditions are satisfied:

- x is a fresh variable of type σ;
- s is not an abstraction $\lambda x.t$ or a meta-variable application $Z(s_1, \ldots, s_n)$;
- s in $C[s]$ is not the left part of an application;
- s in $C[s]$ is not a direct argument of a meta-variable application (so C is not $Z(t_1, \ldots, \Box_\sigma, \ldots, t_n)$).

This definition differs from the one in Section 2.1 by its treatment of meta-variables: neither meta-variable applications nor their immediate subterms are η-expanded. This might be somewhat counter-intuitive for those with a background in for instance HRSs (see Chapter 3.3): a meta-term of functional type does not need to be an abstraction. However, on *terms* the new definition coincides with the old one.

For s a meta-term, consider the following definition of constructs $s{\uparrow^\eta}$ and $\overline{s}$:

- $s{\uparrow^\eta} = \overline{s}$ if s is an abstraction, meta-variable application or base-type term, $s{\uparrow^\eta} = \lambda x.(s \cdot x{\uparrow^\eta})$ otherwise;
- $\overline{\lambda x.s} = \lambda x.(s{\uparrow^\eta})$;
- $\overline{Z(s_1, \ldots, s_n)} = Z(\overline{s_1}, \ldots, \overline{s_n})$ if $Z \in \mathcal{M}$;
- $\overline{f(s_1, \ldots, s_n)} = f(s_1{\uparrow^\eta}, \ldots, s_n{\uparrow^\eta})$ if $f \in \mathcal{F}$;
- $\overline{s \cdot t} = \overline{s} \cdot (t{\uparrow^\eta})$.

Then it is clear that $s{\uparrow^\eta}$ cannot be η-expanded any further. Moreover, with induction on the size of s we can see that if $s \hookrightarrow_\eta t$, then $s{\uparrow^\eta} = t{\uparrow^\eta}$. Thus, $s{\uparrow^\eta}$ is the (unique!) normal form of s under restricted η-expansion.

Example 2.4. If $Z : [\mathtt{o} \times (\mathtt{o} \to \mathtt{o})] \longrightarrow \mathtt{a} \to \mathtt{b}$ is a meta-variable and $\mathtt{0} : \mathtt{o}, \mathtt{f} : \mathtt{o} \to \mathtt{o}$ are function symbols, then we have $Z(\mathtt{0}, \mathtt{f}){\uparrow^\eta} = Z(\mathtt{0}, \mathtt{f})$. With an additional function symbol $\mathtt{g} : \mathtt{o} \to \mathtt{o} \to \mathtt{o}$ we have $(\lambda x.(\mathtt{g} \cdot x)) \cdot \mathtt{0} \cdot \mathtt{0}{\uparrow^\eta} = (\lambda xy.(\mathtt{g} \cdot x \cdot y)) \cdot \mathtt{0} \cdot \mathtt{0}$.

2.3 Transformations of AFSMs

Since existing techniques are defined on various formalisms, they often use assumptions which are not present in AFSMs, such as the rules being η-expanded, or function symbols having arity 0. Fortunately, it is often possible to transform an AFSM to a form which has such properties. Here we shall see the basic transformations; in Chapter 3 we will discuss how these results can be used to relate AFSMs to other formalisms.

2.3.1 Currying and Uncurrying

First let us consider the status of *application*. When extending first-order results it is often convenient to consider the $\cdot$ operator just as a family of binary function symbols. Unfortunately, this leads to some problems for systems given in an applicative style. Consider, for example, an applicative version of Example 2.3:

Example 2.5. The "applicative version" of `map` has symbols which do not take arguments directly:

$$\begin{array}{rcl} \mathtt{nil} & : & \mathtt{list}, \\ \mathtt{cons} & : & \mathtt{nat} \to \mathtt{list} \to \mathtt{list}, \\ \mathtt{map} & : & (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{list} \to \mathtt{list} \\ \mathtt{0} & : & \mathtt{nat} \\ \mathtt{s} & : & \mathtt{nat} \to \mathtt{nat} \end{array}$$

And the following rules:

$$\begin{array}{rcl} \mathtt{map} \cdot (\lambda x.F(x)) \cdot \mathtt{nil} & \Rightarrow & \mathtt{nil} \\ \mathtt{map} \cdot (\lambda x.F(x)) \cdot (\mathtt{cons} \cdot X \cdot Y) & \Rightarrow & \mathtt{cons} \cdot F(X) \cdot (\mathtt{map} \cdot (\lambda x.F(x)) \cdot Y) \end{array}$$

Translating the application operator to a different function symbol for every type it occurs with leads to the following system:

$$\begin{array}{rcl} @_1(@_3(\mathtt{map}, \lambda x.F(x)), \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ @_1(@_3(\mathtt{map}, \lambda x.F(x)), @_1(@_2(\mathtt{cons}, X), Y)) & \Rightarrow & \\ \multicolumn{3}{r}{@_1(@_2(\mathtt{cons}, F(X)), @_1(@_3(\mathtt{map}, \lambda x.F(x)), Y))} \end{array}$$

Typically, methods like the dependency pair approach (Chapter 6) and path orderings (Chapter 5) have some trouble when the same symbol occurs over and over and over. Thus, if we consider application just as a function symbol, these methods are relatively weak on applicative systems. Systems like the one above are not far-fetched: functional programming languages like Haskell generally employ an applicative syntax. We might alternatively consider extending results from applicative systems; results are available both for typed applicative systems [129] and untyped ones [47]. However, it would be preferable to deal with both applicative and functional AFSMs without using separate methods.

In first-order rewriting, the question whether properties such as confluence and rewriting are preserved under *currying* (presenting a system in an applicative form) and *uncurrying* (presenting an applicative system in functional form) is studied in [56, 65, 67, 113]. In [67] a currying transformation from (functional) term rewriting systems (TRSs) into applicative term rewriting systems (ATRSs) is defined; it is demonstrated that a TRS is terminating if and only if its curried form is. In [56], an uncurrying transformation from ATRSs to TRSs is defined that can deal with partial application and leading variables, as long as these variables do not occur in the left-hand side of rewrite rules. This transformation is sound and complete with respect to termination. Unfortunately, in higher-order rewriting, these results do not apply due to typing restrictions, and because application is already a part of term formation. A typical AFSM has *both* function application and normal application, which is not the case in first-order or applicative systems.

Thus, a higher-order variation of currying and uncurrying results is needed. We could go two ways. Usually, we would like to *uncurry* an applicative system, transforming a term $f \cdot s \cdot t$ into $f(s,t)$. Such a form is more convenient in for instance path orderings, or to define argument filterings (see Chapter 6.6.3). On the other hand, we will have to deal with application anyway, since it is part of the term formation. To simplify the formalism it might be a good move to *curry* terms, making the system entirely applicative.

Transformation 2.6 *(Currying)* Let $\mathcal{R}$ be a set of rules over a signature $\mathcal{F}$, and define the following mapping on type declarations: $\mathtt{cur}([\sigma_1 \times \ldots \sigma_n] \longrightarrow \tau) = \sigma_1 \to \ldots \to \sigma_n \to \tau$. Next, define the mapping $\mathtt{cur}$ from meta-terms over $\mathcal{F}$ to meta-terms over the 'curried version' of $\mathcal{F}$, notation $\mathcal{F}^{\mathtt{cur}}$, as follows:

$$\begin{array}{lcll}
\mathtt{cur}(f(s_1,\ldots,s_n)) & = & f \cdot \mathtt{cur}(s_1) \cdots \mathtt{cur}(s_n) & (f \in \mathcal{F}) \\
\mathtt{cur}(x) & = & x & (x \in \mathcal{V}) \\
\mathtt{cur}(Z(s_1,\ldots,s_n)) & = & Z(\mathtt{cur}(s_1),\ldots,\mathtt{cur}(s_n)) & (Z \in \mathcal{M}) \\
\mathtt{cur}(\lambda x.s) & = & \lambda x.\mathtt{cur}(s) & \\
\mathtt{cur}(s \cdot t) & = & \mathtt{cur}(s) \cdot \mathtt{cur}(t) &
\end{array}$$

The curried version $\mathcal{R}^{\mathtt{cur}}$ of the set of rules $\mathcal{R}$ consists of the rules $\mathtt{cur}(l) \Rightarrow \mathtt{cur}(r)$ for every rule $l \Rightarrow r$ in $\mathcal{R}$.

Note that meta-variable applications are *not* made applicative; doing so would cause a pattern to be transformed into a non-pattern.

The curried version of the system $(\mathcal{F}_{\mathtt{map}}, \mathcal{R}_{\mathtt{map}})$ from Example 2.3 is exactly the system $(\mathcal{F}_{\mathtt{map}'}, \mathcal{R}_{\mathtt{map}'})$ from Example 2.5. Every term in the original system corresponds to a unique term in the curried system. *Not* every curried term corresponds to a term in the original system. For example, the well-defined applicative term $\mathtt{map} \cdot \mathtt{s}$ is not the curried version of a term in $\mathcal{F}_{\mathtt{map}}$. However, due to η-expansion this is not a significant problem. As we will see (in Theorem 2.10), termination of $\Rightarrow_{\mathcal{R}}$ is equivalent to termination of $\Rightarrow_{\mathcal{R}^{\mathtt{cur}}}$.

Lemma 2.7. *For a term s with symbols in $\mathcal{F}$, and substitution γ whose domain contains all meta-variables in s, let $\gamma^{\mathtt{cur}} = [x := \mathtt{cur}(\gamma(x)) \mid x \in dom(\gamma)]$. Then $\mathtt{cur}(s\gamma) = \mathtt{cur}(s)\gamma^{\mathtt{cur}}$.*

Proof. By induction, first on the number of meta-variables in s, second on its size.

The most interesting case is when s is a meta-application $Z(s_1, \ldots, s_n)$. We can write $\gamma(Z) = \lambda x_1 \ldots x_n.t$, and $\mathtt{cur}(s\gamma) = \mathtt{cur}(t[x_1 := s_1\gamma, \ldots, x_n := s_n\gamma])$. Note that, since the domain of γ contains all meta-variables in s, all $s_i\gamma$ are terms. Note also that t is a term, so does not contain meta-variables, while s does. Consequently, by the first part of the induction hypothesis, $\mathtt{cur}(s\gamma) = \mathtt{cur}(t)[x_1 := \mathtt{cur}(s_1\gamma), \ldots, x_n := \mathtt{cur}(s_n\gamma)]$, which by the second part equals $\mathtt{cur}(t)[x_1 := \mathtt{cur}(s_1)\gamma^{\mathtt{cur}}, \ldots, x_n := \mathtt{cur}(s_n)\gamma^{\mathtt{cur}}] = \mathtt{cur}(s)\gamma^{\mathtt{cur}}$.

The other cases are simple. If s is an application or abstraction, we only need a straightforward application of the induction hypothesis. If $s = x \in dom(\gamma)$, then $\mathtt{cur}(s\gamma) = \mathtt{cur}(\gamma(x)) = \gamma^{\mathtt{cur}}(x) = s\gamma^{\mathtt{cur}} = \mathtt{cur}(s)\gamma^{\mathtt{cur}}$. If s is a variable x which does not occur in $dom(\gamma)$, then both sides are just x.

Finally, if $s = f(s_1, \ldots, s_n)$ with $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}$, we have $\mathtt{cur}(s\gamma) = \mathtt{cur}(f(s_1\gamma, \ldots, s_1\gamma)) = f \cdot \mathtt{cur}(s_1\gamma) \cdots \mathtt{cur}(s_1\gamma)$. By the induction hypothesis this is equal to $f \cdot (\mathtt{cur}(s_1)\gamma^{\mathtt{cur}}) \cdots (\mathtt{cur}(s_n)\gamma^{\mathtt{cur}}) = (f \cdot \mathtt{cur}(s_1) \cdots \mathtt{cur}(s_n))\gamma^{\mathtt{cur}} = \mathtt{cur}(s)\gamma^{\mathtt{cur}}$. □

Using Lemma 2.7 we can derive that $\mathtt{cur}(s) \Rightarrow_{\mathcal{R}^{\mathtt{cur}}} \mathtt{cur}(t)$ whenever $s \Rightarrow_{\mathcal{R}} t$, as we will do in Theorem 2.10. Consequently, the rewrite relation induced by $\mathcal{R}$ is terminating if the relation induced by $\mathcal{R}^{\mathtt{cur}}$ is. For the other direction we need an "inverse" of the cur relation:

- $\mathtt{uncur}(x) = x$ if $x \in \mathcal{V}$;
- $\mathtt{uncur}(Z(s_1, \ldots, s_n)) = Z(\mathtt{uncur}(s_1), \ldots, \mathtt{uncur}(s_n))$ if $Z \in \mathcal{M}$;
- $\mathtt{uncur}(\lambda x.s) = \lambda x.\mathtt{uncur}(s)$;
- $\mathtt{uncur}(s \cdot t) = \mathtt{uncur}(s) \cdot \mathtt{uncur}(t)$ if $\mathtt{head}(s)$ is not a function symbol;
- $\mathtt{uncur}(f \cdot s_1 \cdots s_k) = \lambda x_{k+1} \ldots x_n.f(\mathtt{uncur}(s_1), \ldots, \mathtt{uncur}(s_k), x_{k+1}, \ldots, x_n)$ if $k < n$, where $n := ar(f)$;
- $\mathtt{uncur}(f \cdot s_1 \cdots s_k) = f(\mathtt{uncur}(s_1), \ldots, \mathtt{uncur}(s_n)) \cdot \mathtt{uncur}(s_{n+1}) \cdots \mathtt{uncur}(s_k)$ if $k \geq n := ar(f)$.

Here $ar(f)$ denotes the arity of f in the signature $\mathcal{F}$; obviously in $\mathcal{F}^{\mathtt{cur}}$ all symbols have arity 0. This translation affects only parts of a meta-term where a function symbol is applied to a lower number of arguments than expected by the arity in the original, functional, signature.

Note that uncur has the following property: for all s, t: $\mathtt{uncur}(s) \cdot \mathtt{uncur}(t) \Rightarrow_\beta^= \mathtt{uncur}(s \cdot t)$ (where $\Rightarrow_\beta^=$ is the reflexive closure of $\Rightarrow_\beta$). If s is an (application headed by an) abstraction, variable or meta-application, both sides are equal, if s has the form $f \cdot s_1 \cdots s_k$ a single $\Rightarrow_\beta$ step may be needed (but only if $k < ar(f)$).

Lemma 2.8. *Let l be a pattern over a signature $\mathcal{F}$ and γ a substitution over $\mathcal{F}^{\mathtt{cur}}$ whose domain contains only meta-variables, but does contain all meta-variables $Z \in FMV(\mathtt{cur}(l))$. Let $\gamma^{\mathtt{uncur}} = [Z := \mathtt{uncur}(\gamma(Z)) \mid Z \in dom(\gamma)]$.*

Then $\mathtt{uncur}(\mathtt{cur}(l)\gamma) = l\gamma^{\mathtt{uncur}}$.

Proof. By induction on the form of l.

If $l = Z(x_1, \ldots, x_n)$, then $\mathtt{cur}(l) = l$. Using α-conversion we can write $\gamma(Z) = \lambda x_1 \ldots x_n.t$, so $\mathtt{uncur}(\mathtt{cur}(l)\gamma) = \mathtt{uncur}(l\gamma) = \mathtt{uncur}(t) = l\gamma^{\mathtt{uncur}}$.

The cases where $l = \lambda x.l'$ and $l = x \cdot l_1 \cdots l_n$ are immediate with the induction hypothesis (in the latter case we note that the domain of γ does not contain x, as x is not a meta-variable), as is the case where $l = f(l_1, \ldots, l_m) \cdot l_{m+1} \cdots l_n$. Since l is a pattern, these are the only forms it can have. □

Lemma 2.9. *For any meta-term s over $\mathcal{F}^{\mathtt{cur}}$ and substitution γ whose domain contains all meta-variables in $FMV(s)$, we have: $\mathtt{uncur}(s)\gamma^{\mathtt{uncur}} \Rightarrow_\beta^* \mathtt{uncur}(s\gamma)$.*

Proof. By induction, first on size of $FMV(s)$, second on the size of s.

The most interesting case is when $s = Z(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ with $m \geq n \geq 0$. In this case, $\mathtt{uncur}(s) = Z(\mathtt{uncur}(s_1), \ldots, \mathtt{uncur}(s_n)) \cdot s_{n+1} \cdots s_m$.

Let $\gamma(Z) = \lambda x_1 \ldots x_n.t$.

Then $\mathtt{uncur}(s)\gamma^{\mathtt{uncur}} = \mathtt{uncur}(t)[x_1 := \mathtt{uncur}(s_1)\gamma^{\mathtt{uncur}}, \ldots, x_n := \mathtt{uncur}(s_n)\gamma^{\mathtt{uncur}}] \cdot \mathtt{uncur}(s_{n+1})\gamma^{\mathtt{uncur}} \cdots \mathtt{uncur}(s_m)\gamma^{\mathtt{uncur}}$.

By the second part of the induction hypothesis, this β-reduces to the term $\mathtt{uncur}(t)[x_1 := \mathtt{uncur}(s_1\gamma), \ldots, x_n := \mathtt{uncur}(s_n\gamma)] \cdot \mathtt{uncur}(s_{n+1}\gamma) \cdots \mathtt{uncur}(s_m\gamma)$. Noting that t is a term, while s is not, we can use the first part of the induction hypothesis to see that this term $\Rightarrow_\beta^* \mathtt{uncur}(t[x_1 := s_1\gamma, \ldots, x_n := s_n\gamma]) \cdot \mathtt{uncur}(s_{n+1}\gamma) \cdots \mathtt{uncur}(s_m\gamma)$.

By the observation below the definition of $\mathtt{uncur}$, this term $\Rightarrow_\beta^* \mathtt{uncur}(t[\vec{x} := \vec{s\gamma}] \cdot s_{n+1}\gamma \cdots s_m\gamma) = \mathtt{uncur}(s\gamma)$.

The case where s is headed by a variable in the domain of γ is very similar; we just do not need the first part of the induction hypothesis.

The other cases:

- $s = x \cdot s_1 \cdots s_n$ with $x \notin dom(\gamma)$
- $s = (\lambda x.s_0) \cdot s_1 \cdots s_n$
- $s = f \cdot s_1 \cdots s_n$ with $n < ar(f)$
- $s = f \cdot s_1 \cdots s_n$ with $n \geq ar(f)$

are all completely straightforward using the induction hypothesis. □

With these preparations, we are ready to move on to the main result:

Theorem 2.10. *$\Rightarrow_{\mathcal{R}}$ is terminating on terms over $\mathcal{F}$ if and only if $\Rightarrow_{\mathcal{R}^{\mathtt{cur}}}$ is terminating on terms over $\mathcal{F}^{\mathtt{cur}}$.*

Proof. It is easy to see that $s \Rightarrow_{\mathcal{R}} t$ implies $\mathtt{cur}(s) \Rightarrow_{\mathcal{R}^{\mathtt{cur}}} \mathtt{cur}(t)$, using induction on the form of s:

- the base cases, when $s \Rightarrow_{\mathcal{R},top} t$ with either clause `(rule)` or clause `(beta)` use Lemma 2.7: for example, if $l\gamma \Rightarrow_{\mathcal{R}} r\gamma$, then $\mathtt{cur}(l\gamma) = \mathtt{cur}(l)\gamma^{\mathtt{cur}} \Rightarrow_{\mathcal{R}^{\mathtt{cur}}} \mathtt{cur}(r)\gamma^{\mathtt{cur}} = \mathtt{cur}(r\gamma)$;
- the induction cases, when the reduction takes place in a subterm, are straightforward: for example, if $s = f(s_1, \ldots, s_i, \ldots, s_n) \Rightarrow_{\mathcal{R}} f(s_1, \ldots, s'_i, \ldots, s_n) = t$, then $\mathtt{cur}(s) = f \cdot \mathtt{cur}(s_1) \cdots \mathtt{cur}(s_i) \cdots \mathtt{cur}(s_n) \Rightarrow_{\mathcal{R}^{\mathtt{cur}}} f \cdot \mathtt{cur}(s_1) \cdots \mathtt{cur}(s'_i) \cdots \mathtt{cur}(s_n)$.

Thus, any infinite reduction in $\Rightarrow_{\mathcal{R}}$ leads to an infinite reduction in $\Rightarrow_{\mathcal{R}^{\mathtt{cur}}}$, which gives the implication from right to left.

For the other direction, we also use induction on the size of s, to derive that $\mathtt{uncur}(s) \Rightarrow_{\mathcal{R}} \cdot \Rightarrow_{\beta}^{*} \mathtt{uncur}(t)$ whenever $s \Rightarrow_{\mathcal{R}^{\mathtt{cur}}} t$. The induction steps (when the reduction takes place in a subterm) are trivial. Lemmas 2.8 and 2.9 and the observation that $\mathtt{uncur}(\mathtt{cur}(r)) = r$, combine to give the two base cases. □

Note the *if and only if* in Theorem 2.10. Because of this equivalence the theorem works in two ways. We can curry a functional system, but also uncurry an applicative system, simply by taking the inverse of Transformation 2.6. For an applicative system, there are usually many sets of corresponding functional rules, all of which are equivalent for purposes of termination.

Example 2.11. Consider the following system:

$$\begin{array}{rcl} \mathtt{emap}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{emap}(F, \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F \cdot X, \mathtt{emap}(\lambda x.\mathtt{twice}(F) \cdot x, Y)) \\ \mathtt{twice}(F) \cdot X & \Rightarrow & F \cdot (F \cdot X) \end{array}$$

Note that the `twice` always appears with an additional argument. Thus, it has the same curried form as the following system:

$$\begin{array}{rcl} \mathtt{emap}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{emap}(F, \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F \cdot X, \mathtt{emap}(\lambda x.\mathtt{twice}(F, x), Y)) \\ \mathtt{twice}(F, X) & \Rightarrow & F \cdot (F \cdot X) \end{array}$$

By Theorem 2.10, their termination is equivalent.

Example 2.12. Note that Theorem 2.10 *only* says that termination of a set of rules and its curried version are equivalent. It is not allowed to choose an arity for all function symbols and apply `uncur` on the rule schemes. Consider for instance the following applicative system:

$$\mathcal{F} = \left\{ \begin{array}{lcl} f & : & (\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat} \\ g & : & \mathtt{nat} \to \mathtt{nat} \end{array} \right\} \quad \mathcal{R} = \{g \cdot X \Rightarrow f \cdot g\}$$

This system is terminating. However, if we naively uncurried it, the result, which has a rule $g(X) \Rightarrow f(\lambda x.g(x))$, is not terminating. This is because the curried version of the latter rule is not $g \cdot X \Rightarrow f \cdot g$.

Comment: In most existing termination techniques, and certainly the methods discussed in this thesis, it is optimal for function symbols to have an arity that is as high as possible. Because of Theorem 2.10 we can avoid making special cases for systems where this does not hold. We simply assume that any AFSM under consideration has first been uncurried as much as possible.

2.3.2 Eta-expanding Rules

Now let us consider η-expansion. It is often convenient to assume that every term of some functional type $\sigma \to \tau$ has the form $\lambda x.s$, which only reduces if its subterm s does. This is the case if we work modulo η, equating $s : \sigma \to \tau$, with s not an abstraction, to $\lambda x.(s \cdot x)$. This is for instance done in Nipkow's HRSs, which we will discuss in Section 3.3. However, in AFSMs β-reduction is a separate step, and in such a setup working modulo η causes problems (since this would give $s \cdot t =_\eta (\lambda x.(s \cdot x)) \cdot t \Rightarrow_\beta s \cdot t$). Therefore, instead of using η-equality, we will limit reasoning to η-long terms.

Theorem 2.13. *Let $\mathcal{R}$ be a set of rules in η-long form. Then the set of η-long terms is closed under rewriting, and the rewrite relation $\Rightarrow_\mathcal{R}$ is terminating if and only if it is terminating on η-long terms.*

Let us postpone the proof of Theorem 2.13 for the time being, as we will be able to derive it as a consequence of Theorem 2.16.

The requirement that the rules are η-long is essential. Consider for example the system from Example 2.12, which has a single rule $g \cdot X_{\mathtt{nat}} \Rightarrow f \cdot g$. Evidently, the set of η-long terms is not closed under rewriting. This rule generates a terminating rewrite relation, but its η-long variation, $g \cdot X \Rightarrow f \cdot (\lambda x.(g \cdot x))$, does not: the left-hand side can be embedded in the right-hand side. This example is contrived, but it shows that we cannot be careless with η-expansion.

However, to prove termination of a system we can use a transformation provided it preserves *non-termination*. At the price of completeness, we can use Transformation 2.14 which, as we will see in Theorem 2.16, has this property.

Transformation 2.14 *(η-expanding rules)* Let $\mathcal{R}$ be a set of rules. Define $\mathcal{R}^\uparrow$ as the set consisting of all rules $(l \cdot Z_1 \cdots Z_n)\uparrow^\eta \Rightarrow (r \cdot Z_1 \cdots Z_n)\uparrow^\eta$ with $l \Rightarrow r$ in $\mathcal{R}$, with $l : \sigma_1 \to \ldots \to \sigma_n \to \iota$ and each Z_i a fresh meta-variable of type σ_i.

Note that, given η-long rules, $\mathcal{R} = \mathcal{R}^\uparrow$ (since the left-hand side of a rule cannot be an abstraction, so η-long rules necessarily have base type).

Lemma 2.15. *For meta-terms s, t, substitutions γ whose domain contains only meta-variables, and contains all meta-variables in $FMV(s)$, and substitutions δ whose domain contains no meta-variables:*

1. $\lambda x_1 \dots x_n.((s{\uparrow^\eta}) \cdot (x_1{\uparrow^\eta}) \cdots (x_n{\uparrow^\eta})) \Rightarrow_\beta^* s{\uparrow^\eta}$;
2. $(s{\uparrow^\eta}) \cdot (t{\uparrow^\eta}) \Rightarrow_\beta^* (s \cdot t){\uparrow^\eta}$ *if* s, t *are terms;*
3. *if* s *is a term, then* $(s{\uparrow^\eta})\gamma^\uparrow \Rightarrow_\beta^* (s{\uparrow^\eta})\overline{\gamma} = (s\gamma){\uparrow^\eta}$, *where* $\gamma^\uparrow = [x := \gamma(x){\uparrow^\eta} \mid x \in dom(\gamma)]$ *and* $\overline{\gamma} = [x := \overline{\gamma(x)} \mid x \in dom(\gamma)]$;
4. *if* s *is a pattern, then* $(s{\uparrow^\eta})\gamma^\uparrow = (s\gamma){\uparrow^\eta}$;
5. $(s{\uparrow^\eta})\gamma^\uparrow \Rightarrow_\beta^* (s\gamma){\uparrow^\eta}$;
6. *if* $q, u, v_1, \dots, v_n$ *are terms, then* $((\lambda x.q) \cdot u \cdot \vec{v}){\uparrow^\eta} \Rightarrow_\beta^+ (q[x := u] \cdot \vec{v}){\uparrow^\eta}$;
7. *if* s, t *are terms, and* $s \Rightarrow_\mathcal{R} t$, *then* $s{\uparrow^\eta} \Rightarrow_{\mathcal{R}^\uparrow}^+ t{\uparrow^\eta}$.

Here, recall that $\overline{s}$ and $s{\uparrow^\eta}$ are defined as in Section 2.2, below the definition of restricted η-expansion.

Proof. (1) and (2) are well-known properties of η-expansion on terms; it is proved by induction on the type of s, using that $x{\uparrow^\eta}\ [x := x{\uparrow^\eta}] \Rightarrow_\beta^* x{\uparrow^\eta}$ for variables x (which holds by induction on the type of x).

(3) holds by induction on the form of s. The cases where s is (headed by) an abstraction, a function application or a variable not in $dom(\gamma)$ are all straightforward. If $s = x \cdot s_1 \cdots s_n : \sigma_1 \to \ldots \to \sigma_m \to \iota$ with $\gamma(x) = t$, then $(s\gamma){\uparrow^\eta} = \lambda y_1 \dots y_m.(t{\uparrow^\eta}) \cdot ((s_1{\uparrow^\eta})\gamma^\uparrow) \cdots ((s_n{\uparrow^\eta})\gamma^\uparrow) \cdot (y_1{\uparrow^\eta}) \cdots (y_m{\uparrow^\eta})$. With the induction hypothesis, each $(s_i{\uparrow^\eta})\gamma^\uparrow$ reduces to $(s_i\gamma){\uparrow^\eta}$, and if we subsequently use (2) $n + m$ times, this term β-reduces to $\lambda\vec{y}.(t \cdot \vec{s\gamma} \cdot \vec{y}){\uparrow^\eta} = (s\gamma){\uparrow^\eta}$. Since also $(t \cdot \vec{s\gamma} \cdot \vec{y}){\uparrow^\eta} = \overline{t} \cdot (\vec{s\gamma}{\uparrow^\eta}) \cdot (\vec{y}{\uparrow^\eta})$ as seen in Section 2.2, and by the induction hypothesis each $(s_i\gamma){\uparrow^\eta} = (s_i{\uparrow^\eta})\overline{\gamma}$, this term is equal to $(s{\uparrow^\eta})\overline{\gamma}$, so both parts of the induction step are proved.

(4) holds by induction on the form of s. In the base case, $s = Z(x_1, \dots, x_n)$, let $\gamma(Z) = \lambda x_1 \dots x_n.t$ with t in η-long form. Then $(s{\uparrow^\eta})\gamma^\uparrow = s\gamma^\uparrow = t{\uparrow^\eta} = (s\gamma){\uparrow^\eta}$. All the induction cases, when $s = \lambda x.t$ or $s = x \cdot s_1 \cdots s_n$ or $s = f(s_1, \dots, s_n) \cdot s_{n+1} \cdots s_m$, are straightforward with the induction hypothesis. Note that, since s is a pattern, there is no case $Z(s_1, \dots, s_n)$ with the s_i not variables.

(5) is proved with a shared induction on two claims; we will see that both $(s{\uparrow^\eta})\gamma^\uparrow \Rightarrow_\beta^* (s\gamma){\uparrow^\eta}$, and that $\overline{s}\gamma^\uparrow$ β-reduces to either $(s\gamma){\uparrow^\eta}$ or to $\overline{s\gamma}$.

First consider the second claim. The cases where s is an abstraction or function application are immediate with the induction hypothesis (using the

first claim), and if s is a variable then $\overline{s}\gamma^\uparrow = s = \overline{s\gamma}$. If $s = t \cdot q$, then $\overline{s}\gamma^\uparrow = (\overline{t}\gamma^\uparrow) \cdot ((q\uparrow^\eta)\gamma^\uparrow)$. By the induction hypothesis $(q\uparrow^\eta)\gamma^\uparrow \Rightarrow_\beta^* (q\gamma)\uparrow^\eta$ and either $\overline{t}\gamma^\uparrow \Rightarrow_\beta^* \overline{t\gamma}$ or $\overline{t}\gamma^\uparrow \Rightarrow_\beta^* (t\gamma)\uparrow^\eta$. In the first case, $\overline{s}\gamma^\uparrow \Rightarrow_\beta^* \overline{t\gamma} \cdot ((q\gamma)\uparrow^\eta) = \overline{s\gamma}$. In the second case, $\overline{s}\gamma^\uparrow \Rightarrow_\beta^* ((t\gamma)\uparrow^\eta) \cdot ((q\gamma)\uparrow^\eta)$, which by (2) $\Rightarrow_\beta^* (s\gamma)\uparrow^\eta$.

Finally, suppose $s = Z(s_1, \ldots, s_n)$ and $\gamma(Z) = \lambda x_1 \ldots x_n.t$. Then $\overline{s}\gamma^\uparrow = t\uparrow^\eta [x_1 := \overline{s_1}\gamma^\uparrow, \ldots, x_n := \overline{s_n}\gamma^\uparrow]$. Noting that each $\overline{s_i}\gamma^\uparrow$ β-reduces to either $(s_i\gamma)\uparrow^\eta$ or to $\overline{s_i\gamma}$, we can use (3) to see that $\overline{s}\gamma^\uparrow \Rightarrow_\beta^* t[\vec{x} := \vec{s\gamma}]\uparrow^\eta = (s\gamma)\uparrow^\eta$.

For the first claim, we are done if s is an abstraction or base-type meta-term (so both $\overline{s} = s\uparrow^\eta$ and $\overline{s\gamma} = (s\gamma)\uparrow^\eta$), and if s is a meta-variable application (so $s\uparrow^\eta = \overline{s}$ and $\overline{s}\gamma^\uparrow \Rightarrow_\beta^* (s\gamma)\uparrow^\eta$ as we saw above). Otherwise, let $s : \sigma_1 \to \ldots \to \sigma_n \to \iota$. Then $(s\uparrow^\eta)\gamma^\uparrow = (\lambda x_1 \ldots x_n.\overline{s} \cdot (x_1\uparrow^\eta) \cdots (x_n\uparrow^\eta))\gamma^\uparrow = \lambda\vec{x}.\overline{s}\gamma^\uparrow \cdot (\vec{x}\uparrow^\eta)$. As we saw above, $\overline{s}\gamma^\uparrow$ reduces to either $\overline{s\gamma}$, in which case we are done because $\lambda\vec{x}.\overline{s\gamma} \cdot (\vec{x}\uparrow^\eta) = (s\gamma)\uparrow^\eta$, or to $(s\gamma)\uparrow^\eta$, in which case we are done with (2).

(6) uses (1–3). Let $s := (\lambda x.q) \cdot u \cdot \vec{v}$ and $t := q[x := u] \cdot \vec{v}$. Suppose $s : \sigma_{n+1} \to \ldots \to \sigma_m \to \iota$, and write $norm(s) = \lambda y_{n+1} \ldots y_m.t \cdot (y_{n+1}\uparrow^\eta) \cdots (y_m\uparrow^\eta)$. Then $s\uparrow^\eta = norm((\lambda x.(q\uparrow^\eta)) \cdot (u\uparrow^\eta) \cdot (\vec{s}\uparrow^\eta))$. This term $\Rightarrow_\beta norm((q\uparrow^\eta)[x := u\uparrow^\eta] \cdot \vec{s}\uparrow^\eta)$. By (3) and (2) this term $\Rightarrow_\beta^* norm((q[x := u] \cdot \vec{s})\uparrow^\eta) = norm(t\uparrow^\eta)$, which by (1) $\Rightarrow_\beta^* t\uparrow^\eta$ as required.

(7), finally, follows by induction on the size of s. The only two cases which are not trivial with the induction hypothesis are when $s = (\lambda x.q) \cdot u \cdot s_1 \cdots s_n$ and $t = q[x := u] \cdot s_1 \cdots s_n$, or when $s = l\gamma \cdot s_1 \cdots s_n$ and $t = r\gamma \cdot s_1 \cdots s_n$ for some rule $l \Rightarrow r$ and substitution γ whose domain contains all meta-variables in l.

The first of these cases we considered in (6). As for the second, suppose again that $s : \sigma_{n+1} \to \ldots \to \sigma_m \to \iota$. Let $l' := l \cdot Z_1 \cdots Z_m$ and $r' := r \cdot Z_1 \cdots Z_m$; note that $l'\uparrow^\eta \Rightarrow r'\uparrow^\eta$ is a rule in $\mathcal{R}^\uparrow$. Let $\delta := \gamma \cup [Z_1 := s_1, \ldots, Z_n := s_n, Z_{n+1} := x_{n+1}, \ldots, Z_m := x_m]$. Then $s\uparrow^\eta = \lambda x_{n+1} \ldots x_m.(l'\delta)\uparrow^\eta$. If either $n > 0$ or $r\gamma$ is not an abstraction, then also $t\uparrow^\eta = \lambda x_{n+1} \ldots x_m.(r'\delta)\uparrow^\eta$. If $n = 0$ and $r\gamma$ is an abstraction, then $\overline{r\gamma} = (r\gamma)\uparrow^\eta$, so $\lambda x_{n+1} \ldots x_m.(r'\delta)\uparrow^\eta = norm(\overline{r\gamma}) = norm((r\gamma)\uparrow^\eta) \Rightarrow_\beta^* (r\gamma)\uparrow^\eta = t\uparrow^\eta$ by (1). Thus, $s\uparrow^\eta = \lambda\vec{x}.(l'\delta)\uparrow^\eta = \lambda\vec{x}.(l'\uparrow^\eta)\delta^\uparrow$ by (4), $\Rightarrow_{\mathcal{R}^\uparrow} \lambda\vec{x}.(r'\uparrow^\eta)\delta^\uparrow \Rightarrow_\beta^* \lambda\vec{x}.(r'\delta)\uparrow^\eta$ by (5), $\Rightarrow_\beta^* t\uparrow^\eta$. □

Lemma 2.15 provides all the technical background to derive Theorems 2.13 and 2.16:

Theorem 2.16. *Let $\mathcal{R}$ be a set of rules, and $\mathcal{R}^\uparrow$ as obtained from Transformation 2.14. If there is no infinite reduction $s_1 \Rightarrow_{\mathcal{R}^\uparrow} s_2 \Rightarrow_{\mathcal{R}^\uparrow} \ldots$ with all s_i terms in η-long form, then $\Rightarrow_\mathcal{R}$ is terminating.*

Proof. Suppose $\Rightarrow_{\mathcal{R}^\uparrow}$ is terminating on η-long terms, and, towards a contradiction, that $\Rightarrow_\mathcal{R}$ is not terminating. Thus, there is a reduction $s_1 \Rightarrow_\mathcal{R} s_2 \Rightarrow_\mathcal{R} \ldots$. Since by Lemma 2.15(7) also $s_i\uparrow^\eta \Rightarrow_{\mathcal{R}^\uparrow}^+ s_{i+1}\uparrow^\eta$ for all i, the theorem follows. □

Theorem 2.13 follows from Theorem 2.16, because $\mathcal{R} = \mathcal{R}^{\uparrow}$ if the rules are η-long to begin with (and evidently, if there is no infinite reduction, there is also no infinite reduction on η-long terms).

In most examples which are commonly considered, η-expanding the rules does not lead to non-termination. Since it is very convenient to have η-long rules (and especially to be able to assume that all function symbols have a base type as output type), Theorem 2.16 is an important result.

2.3.3 Simple Meta-Variables

Another transformation we should consider is the "flattening" of meta-variable applications. That is, changing a rule of the form

$$\mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y)) \Rightarrow \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y))$$

into a rule where the meta-variables have arity 0:

$$\mathtt{map}(F, \mathtt{cons}(X, Y)) \Rightarrow \mathtt{cons}(F \cdot X, \mathtt{map}(F, Y))$$

Part of the role of meta-variable application is taken over by β-reduction. This transformation will be necessary to reuse results for Algebraic Functional Systems (see Chapter 3.4), as this formalism does not have meta-variables, but does have a separate β-reduction rule.

Not all AFSMs can be transformed in this way: sometimes a system simply cannot be expressed without meta-variables taking arguments. For example, without this construction we could not express a rule like the following rule for derivation (copied from [13]):

$$\mathtt{d}(\lambda x.\mathtt{sin}(F(x))) \Rightarrow \lambda x.(\mathtt{d}(\lambda y.F(y)) \cdot x) \times \mathtt{cos}(F(x))$$

This is because for instance $\mathtt{d}(\lambda x.\mathtt{sin}(F \cdot x))$ does not match a term $\mathtt{d}(\lambda x.\mathtt{sin}(x))$; an application can only be instantiated with an application.

However, in many (or even most) common examples, this kind of matching does not occur. Such systems we *can* transform.

Definition 2.17. A meta-term s has *simple meta-applications* if all meta-variables in s occur in a form $\lambda x_1 \ldots x_n.Z(x_1, \ldots, x_n)$.

Thus, a meta-term $\mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y))$ has simple meta-applications, whereas a meta-term $\mathtt{d}(\lambda x.\mathtt{sin}(F(x)))$ does not.

Transformation 2.18 *(Flattening simple meta-applications)* Given an AFSM $(\mathcal{F}, \mathcal{R})$ such that for all rules $l \Rightarrow r \in \mathcal{R}$, the left-hand side l has simple meta-applications. For each meta-variable Z with type declaration $[\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau$, let Z' be a uniquely corresponding meta-variable with type $\sigma_1 \to \ldots \to \sigma_n \to \tau$.

The flattening function $\mathtt{flat}$ is inductively defined as follows:

$$\begin{array}{lcl}
\mathtt{flat}(x) & = & x \quad \text{if } x \in \mathcal{V} \\
\mathtt{flat}(f(s_1,\ldots,s_n)) & = & f(\mathtt{flat}(s_1),\ldots,\mathtt{flat}(s_n)) \\
\mathtt{flat}(s \cdot t) & = & \mathtt{flat}(s) \cdot \mathtt{flat}(t) \\
\mathtt{flat}(\lambda x.s) & = & \lambda x.\mathtt{flat}(s) \\
 & & \text{if } s \neq \lambda z_2 \ldots z_n.Z(x,z_2,\ldots,z_n) \\
\mathtt{flat}(\lambda x_1 \ldots x_n.Z(x_1,\ldots,x_n)) & = & Z' \\
\mathtt{flat}(Z(s_1,\ldots,s_n)) & = & Z' \cdot \mathtt{flat}(s_1) \cdots \mathtt{flat}(s_n)
\end{array}$$

Define $\mathcal{R}^{\mathtt{flat}} := \{\mathtt{flat}(l) \Rightarrow \mathtt{flat}(r) \mid l \Rightarrow r \in \mathcal{R}\}$.

Theorem 2.19. *If $(\mathcal{F}, \mathcal{R}^{\mathtt{flat}})$ is a terminating AFSM, then so is $(\mathcal{F}, \mathcal{R})$.*

Proof. This holds because $s \Rightarrow_{\mathcal{R}} t$ implies $s \Rightarrow_{\mathcal{R}^{\mathtt{flat}}} \cdot \Rightarrow_\beta^* t$. This is obvious when the reduction is a β-step, and if $s = C[l\gamma]$ and $t = C[r\gamma]$, let $\delta := [Z' := \gamma(Z) \mid Z \in dom(\gamma)]$. Then: $\mathtt{flat}(l)\delta = l\gamma$ and $\mathtt{flat}(r)\delta \Rightarrow_\beta^* r\gamma$. This holds by induction on l and r respectively, taking into account that l has simple meta-applications.

In the induction, write s for l or r. We will see that $\mathtt{flat}(s)\delta \Rightarrow_\beta^* s\gamma$ and that this is an equality if s has simple meta-applications. We may assume that $dom(\gamma)$ contains only meta-variables, and contains all meta-variables occurring in s. When s is a variable the result follows immediately, if it is a functional term, application or an abstraction not of the form $\lambda\vec{x}.Z(\vec{x})$, it follows easily from the induction hypothesis (note that the subterms of a meta-term with simple meta-applications also have simple meta-applications).

What remains are the cases when s has the form $\lambda x_1 \ldots x_n.Z(x_1,\ldots,x_n)$ or $Z(s_1,\ldots,s_n)$. In the latter case s does not have simple meta-applications, so we only have to see that $\mathtt{flat}(s)\delta \Rightarrow_\beta^* s\gamma$. Consider $\gamma(Z)$; this must have the form $\lambda y_1 \ldots y_n.t$. In the first case, then, $s\gamma = \lambda x_1 \ldots x_n.t[y_1 := x_1,\ldots,y_n := x_n] = \lambda y_1 \ldots y_n.t$ by α-conversion, which equals $\gamma(Z) = \delta(Z') = \mathtt{flat}(s)\delta$. In the latter case, $\mathtt{flat}(s)\delta = \gamma(Z) \cdot \mathtt{flat}(s_1)\delta \cdots \mathtt{flat}(s_n)\delta \Rightarrow_\beta^* (\lambda\vec{y}.t) \cdot s_1\gamma \cdots s_n\gamma$ by the induction hypothesis, and this β-reduces to $t[y_1 := s_1\gamma,\ldots,y_n := s_n\gamma] = s\gamma$. □

Note that Theorem 2.19 is a one-way result. The other direction does *not* hold, as demonstrated for example by the (terminating) system with a single rule:

$$\mathtt{f}(\lambda x.F(x)) \Rightarrow F(\mathtt{f}(\lambda x.\mathtt{0}))$$

This system is transformed into the AFSM:

$$\mathtt{f}(F) \Rightarrow F \cdot \mathtt{f}(\lambda x.\mathtt{0})$$

The result is non-terminating: $\mathtt{f}(\lambda x.\mathtt{0}) \Rightarrow (\lambda x.\mathtt{0}) \cdot \mathtt{f}(\lambda x.\mathtt{0})$, which contains the original term as a subterm. In the original AFSM, this does not happen: $\mathtt{f}(\lambda x.\mathtt{0})$ reduces only to $\mathtt{0}$. Intuitively, the created β-redex is immediately reduced.

2.3.4 Changing Types

A final transformation we will discuss is the possibility to change a base type into another type, either a different base type or a functional type.

Let $\mathcal{B}_1$ and $\mathcal{B}_2$ be sets of base types; they may have elements in common, but this is not necessary. A *type-changing function* is a function ζ which assigns to all elements of $\mathcal{B}_1$ a type built from base types in $\mathcal{B}_2$ and (possibly) also the $\to$ constructor. We extend ζ to be a function on all types over $\mathcal{B}_1$, by defining

$$\zeta(\sigma \to \tau) = \zeta(\sigma) \to \zeta(\tau)$$

And for type declarations,

$$\zeta([\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau) = [\zeta(\sigma_1) \times \ldots \times \zeta(\sigma_n)] \longrightarrow \zeta(\tau)$$

For a given AFSM $(\mathcal{F}, \mathcal{R})$, let $\mathcal{F}^\zeta = \{f : \zeta(\sigma) \mid f : \sigma \in \mathcal{F}\}$. Also, let $\mathcal{V}^\zeta = \{x : \zeta(\sigma) \mid x : \sigma \in \mathcal{V}\}$ and $\mathcal{M}^\zeta = \{Z : \zeta(\sigma) \mid Z : \sigma \in \mathcal{M}\}$. It is clear, with a trivial induction on the definition of meta-terms, that if $s : \sigma$ for meta-terms over $(\mathcal{F}, \mathcal{V}, \mathcal{M})$, then $s : \zeta(s)$ for meta-terms over $(\mathcal{F}^\zeta, \mathcal{V}^\zeta, \mathcal{M}^\zeta)$. Substitution also carries over: if $s\gamma$ is defined over $\mathcal{F}$, then it is defined over $\mathcal{F}^\zeta$, and the terms are exactly the same. Consequently, reduction is preserved: if $s \Rightarrow_\mathcal{R} t$ in the AFSM $(\mathcal{F}, \mathcal{R})$, then also $s \Rightarrow_\mathcal{R} t$ in the AFSM $(\mathcal{F}^\zeta, \mathcal{R})$.

By this reasoning, we obtain the following result:

Theorem 2.20. *For any AFSM $(\mathcal{F}, \mathcal{R})$ and type changing function ζ, if the AFSM $(\mathcal{F}^\zeta, \mathcal{R})$ is terminating, then so is the AFSM $(\mathcal{F}, \mathcal{R})$.*

Theorem 2.20 can be used to give an AFSM a form which certain termination techniques can handle better. A very standard type transformation is the function which maps all base types to some fixed base type $\mathtt{o}$. We say that this type change *collapses* the base types. However, it can also be useful to use a type changing function in exactly those cases where collapsing would lead to non-termination.

Example 2.21 (Functional Map)**.** Consider the following example from the termination problem database (`Applicative_05__Ex4MapList`):

$$\begin{array}{rcl}
\mathtt{nil} & : & \mathtt{natlist} \\
\mathtt{cons} & : & [\mathtt{nat} \times \mathtt{natlist}] \longrightarrow \mathtt{natlist} \\
\mathtt{fnil} & : & \mathtt{funclist} \\
\mathtt{fcons} & : & [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{funclist}] \longrightarrow \mathtt{funclist} \\
\mathtt{fmap} & : & [\mathtt{funclist} \times \mathtt{nat}] \longrightarrow \mathtt{natlist}
\end{array}$$

$$\begin{array}{rcl}
\mathtt{fmap}(\mathtt{fnil}, Y) & \Rightarrow & \mathtt{nil} \\
\mathtt{fmap}(\mathtt{fcons}(F, X), Y) & \Rightarrow & \mathtt{cons}(F \cdot Y, \mathtt{fmap}(X, Y))
\end{array}$$

If all types are collapsed to the same base type $\mathtt{o}$, this system is non-terminating:

$\mathtt{fmap}(\mathtt{fcons}(\lambda x.\mathtt{fmap}(x,x),y),\mathtt{fcons}(\lambda z.\mathtt{fmap}(z,z),y))$
$\Rightarrow_{\mathcal{R}} \mathtt{cons}((\lambda x.\mathtt{fmap}(x,x))\cdot\mathtt{fcons}(\lambda z.\mathtt{fmap}(z,z),y),\mathtt{fmap}(\ldots))$
$\Rightarrow_{\beta} \mathtt{cons}(\mathtt{fmap}(\mathtt{fcons}(\lambda x.\mathtt{fmap}(x,x),y),\mathtt{fcons}(\lambda z.\mathtt{fmap}(z,z),y)),\mathtt{fmap}(\ldots))$
which contains the original term as a subterm.

Consequently, termination techniques which do not distinguish between base types have trouble. However, this counterexample depends on the subterm $\mathtt{fmap}(x,x)$, which is not well-defined in the original typing. In fact, the system is terminating. We can see this using a type function where $\zeta(\mathtt{funclist}) = \mathtt{o}\to\mathtt{o}$ and $\zeta(\iota) = \mathtt{o}$ for all other base types. Termination of the resulting system will be proved in Chapter 4 (Example 4.21).

It is not obvious how to choose a good type changing function for a given system. For the theory, it is most important to know *that* they can be used, not *how* – this for instance allows termination techniques to assume that there is only one base type. In practice, the search for a type changing function may be combined with the search for a reduction ordering (these will be defined in Section 2.4).

As demonstrated by Example 2.21, we can use type changing functions to express crucial differences in base types with the type construction operator $\to$. In a limited way, this gives us some of the power created by a type ordering and inductive types (see e.g. [19]). Type changing does not have all the power of such a reasoning, but has the advantage that it is not bound to any fixed method.

In this work, type changing functions will almost exclusively be used in the simplest way only, to collapse all base types into one. However, in a possible extension to a *polymorphic* framework they might become very important. This will briefly be discussed in Chapter 9.3.

2.4 Reduction Orderings, Reduction Pairs and Rule Removal

Before diving into the depth of termination methods, we must settle on a terminology for the relations we will need. These definitions are mostly standard, or adapted in a minor way from the standard definitions to the setting of AFSMs. In Section 2.4.1 we shall consider *reduction pairs*, and see roughly how they are used. In Section 2.4.2 we will consider some transformations of reduction pairs rather than term rewriting systems.

2.4.1 Definitions

Well-founded Orderings and Quasi-Orderings. A *quasi-ordering* in mathematics is a binary relation $\geq$ on some set A which is both *transitive* (if $a \geq b$ and $b \geq c$ then $a \geq c$) and *reflexive* ($a \geq a$ for all $a \in A$). A *strict ordering* is a transitive binary relation $>$ which is *irreflexive* (not $a > a$ for any $a \in A$). Following [10], a strict ordering $>$ is *well-founded* if there is no infinite sequence $a_1 > a_2 > \ldots$ Since well-foundedness implies irreflexivity, such a relation is shortly called a *well-founded ordering*.

Comment: This definition of *well-founded*, although often used in the setting of term rewriting, is non-standard in mathematics. Assuming the axiom of choice, the standard definition is the converse of the definition used here; that is, a relation $<$ is well-founded if there is no infinite sequence $a_1, a_2, a_3, \ldots$ such that each $a_{i+1} < a_i$. The notion used here is sometimes referred to as *converse well-founded* or *Noetherian*.

Reduction Orderings and Pairs for TRSs. The traditional way to prove termination of a TRS is to use a *reduction ordering*. This is a well-founded ordering $\succ$ on the set of terms which is well-founded, transitive, stable (if $s \succ t$ then $s\gamma \succ t\gamma$ for all substitutions γ), and monotonic (if $s \succ t$ then $C[s] \succ C[t]$ for all contexts C). If we have a reduction ordering $\succ$, and we can prove that $l \succ r$ for all rules $l \Rightarrow r$, then the reduction relation $\Rightarrow_{\mathcal{R}}$ is included in $\succ$: if $s \Rightarrow_{\mathcal{R}} t$ then we can write $s = C[l\gamma]$ and $t = C[r\gamma]$; we have $l \succ r$ by assumption, $l\gamma \succ r\gamma$ by stability, and $s = C[l\gamma] \succ C[r\gamma] = t$ by monotonicity. By well-foundedness of $\succ$, there is no infinite decreasing $\succ$-chain, so $\Rightarrow_{\mathcal{R}}$ is terminating.

But we can do better! Let us consider *rule removal*. In this old technique (references go back as far as 1979 [92]), a *reduction pair* is used. Rather than proving $l \succ r$ for all rules in one go, we prove $l \succ r$ for *some* rules, $l \succsim r$ for the others, and then remove the strictly oriented rules and prove termination of the rest. To use rule removal we need a *strong reduction pair*[1]: a pair $(\succsim, \succ)$ of a quasi-ordering and a well-founded ordering such that both $\succsim$ and $\succ$ are stable and monotonic, and the two relations are *compatible*: $\succ \cdot \succsim$ is included in $\succ$.

Reduction Orderings and Pairs for AFSMs. To extend the notion of a reduction pair and the method of rule removal to AFSMs, we have to deal with β-reduction and meta-terms. Because of the pattern restriction, we will not need stability; the somewhat weaker condition of *meta-stability* suffices.

Definition 2.22. A *strong reduction pair* is a pair $(\succsim, \succ)$ of a *quasi-ordering* and a *well-founded ordering* on meta-terms, such that:

- $\succsim$ and $\succ$ are *compatible*: $\succ \cdot \succsim$ is included in $\succ$;
- $\succsim$ and $\succ$ are both *meta-stable*: if $s \succsim t$ and s is a pattern of the form $f(\vec{u}) \cdot \vec{v}$, and γ is a substitution with domain $FMV(s) \cup FMV(t)$, then $s\gamma \succsim t\gamma$ (and the same for $\succ$);
- $\succsim$ and $\succ$ are both *monotonic*: if $s \succsim t$ with s, t terms, and C is a context, then $C[s] \succsim C[t]$ (and the same for $\succ$);
- $\succsim$ contains beta: $(\lambda x.s) \cdot t \succsim s[x := t]$ if s and t are terms.

[1] In the literature, a strong reduction pair is commonly referred to as a *strongly monotonic reduction pair*. Here, we use the shorter phrasing to contrast it with a *weak reduction pair*, which is commonly just known as a *reduction pair*.

A *reduction ordering* is a well-founded ordering $\succ$ such that $(\succ^{=}, \succ)$ is a strong reduction pair (where $\succ^{=}$ is the reflexive closure of $\succ$).

Note that stability implies meta-stability; thus, this notion is a strict generalisation of the original definition to the higher-order case.

Theorem 2.23 (Rule Removal)**.** *Suppose we have a strong reduction pair $(\succsim, \succ)$ such that, for some partitioning $\mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2$ of a set of rules, we have:*

- *$l \succ r$ for all rules $l \Rightarrow r \in \mathcal{R}_1$;*
- *$l \succsim r$ for all rules $l \Rightarrow r \in \mathcal{R}_2$;*

Then the AFSM $(\mathcal{F}, \mathcal{R})$ is terminating if and only if $(\mathcal{F}, \mathcal{R}_2)$ is terminating.

Proof. One direction is obvious: if there is no infinite $\Rightarrow_{\mathcal{R}}$-reduction, then there is no infinite reduction of the subrelation $\Rightarrow_{\mathcal{R}_2}$. For the other direction, suppose that $\Rightarrow_{\mathcal{R}_2}$ is terminating. Then any infinite reduction over $\mathcal{R}$ must use a rule in $\mathcal{R}_1$ infinitely often. We will see that such an infinite reduction leads to an infinite decreasing $\succ$-chain, contradicting well-foundedness of $\succ$. So let an infinite reduction $s_0 \Rightarrow_{\mathcal{R}} s_1 \Rightarrow_{\mathcal{R}} \ldots$ be given. For each i:

- if $s_i = C[l\gamma] \Rightarrow_{\mathcal{R}_1} C[r\gamma] = s_{i+1}$, then $l \succ r$ by assumption, $l\gamma \succ r\gamma$ by meta-stability, and $s_i \succ t_i$ by monotonicity of $\succ$;
- if $s_i = C[l\gamma] \Rightarrow_{\mathcal{R}_2} C[r\gamma] = s_{i+1}$, then $l \succsim r$ by assumption, $l\gamma \succsim r\gamma$ by meta-stability, and $s_i \succsim t_i$ by monotonicity of $\succsim$;
- if $s_i = C[(\lambda x.s) \cdot t] \Rightarrow_{\beta} C[s[x := t]]$, then $s_i \succsim t_i$ because $\succsim$ contains beta and is monotonic.

Thus, always $s_i \succsim s_{i+1}$ or $s_i \succ s_{i+1}$, and the latter happens infinitely often. Since $a \succ b_1 \succsim \ldots \succsim b_n \succ c$ implies $a \succ b_n \succ c$ by compatibility, we thus obtain an infinite decreasing $\succ$ sequence. As this is impossible by well-foundedness of $\succ$, the assumption that an infinite $\Rightarrow_{\mathcal{R}}$ reduction exists must be false; $(\mathcal{F}, \mathcal{R})$ is terminating. □

If we have a suitable way of obtaining strong reduction pairs, we can successively remove rules until none remain. At that point, since β-reduction is terminating, we have proved termination of the original system. Moreover, to prove termination of $\mathcal{R}_2$, we do not need to use the same technique. We could for example remove one rule with the polynomial interpretations from Chapter 4, a second rule with the iterative path orderings from Chapter 5, then two more with polynomial interpretations, and finally pass the rest to the dependency pair framework of Chapter 6 and 7.

Weak Reduction Pairs. The *dependency pair approach*, which we will discuss at length in Chapter 6 (and extend in Chapter 7), provides an alternative way to prove termination. In this approach, the termination question is reduced to a number of sets of constraints, which can be handled with (among other methods) a *weak reduction pair*. A weak reduction pair is almost a strong reduction pair, except that $\succ$ is not required to be monotonic.

Definition 2.24. A weak reduction pair is a pair $(\succsim, \succ)$ of a quasi-ordering and a well-founded ordering on meta-terms, such that:

- $\succsim$ and $\succ$ are compatible;
- $\succsim$ and $\succ$ are both meta-stable;
- $\succsim$ is monotonic;
- $\succsim$ contains beta.

Since every strong reduction pair is also a weak reduction pair, a weak reduction pair is often simply called a *reduction pair*. We will consider weak reduction pairs and their requirements in the dependency pair approach in more detail in Chapter 6.6, but will already see some ways to define them in Chapters 4 and 5.

2.4.2 Transformations of Reduction Pairs

In later chapters we will see some examples of both weak and strong reduction pairs. Currently, we do not have much material. But we *do* have a number of ways to transform existing reduction pairs: all the transformations of AFSMs from Section 2.3 can be used for (weak or strong) reduction pairs as well.

Currying and Uncurrying. Let us start at the beginning! The *currying* transformation from Section 2.3.1 leads to a straightforward reduction pair:

Theorem 2.25. *Given a signature $\mathcal{F}$, let a reduction pair $(\succsim, \succ)$ on terms over $\mathcal{F}^{\mathtt{cur}}$ (which is defined as in Transformation 2.6) be given. For $R \in \{\succsim, \succ\}$, let the relation $R_{\mathtt{cur}}$ on (meta-)terms over $\mathcal{F}$ be given by: $s\ R_{\mathtt{cur}}\ t$ iff $\mathtt{cur}(s)\ R\ \mathtt{cur}(t)$.*

Then $(\succsim_{\mathtt{cur}}, \succ_{\mathtt{cur}})$ is a reduction pair, and $\succ_{\mathtt{cur}}$ is monotonic if $\succ$ is.

Proof. Well-foundedness, transitivity, reflexivity and compatibility are inherited from the corresponding properties on $\succsim$ and $\succ$, and monotonicity of $R_{\mathtt{cur}}$ is inherited from monotonicity of R by an easy induction on the size of the context.

Meta-stability of $R_{\mathtt{cur}}$ holds by meta-stability of R and Lemma 2.7: if l is a pattern of the right form, r a meta-variable, and γ a substitution on domain $FMV(l) \cup FMV(r)$, and if $\mathtt{cur}(l)\ R\ \mathtt{cur}(r)$, then $\mathtt{cur}(l\gamma) = \mathtt{cur}(l)\gamma^{\mathtt{cur}}\ R\ \mathtt{cur}(r)\gamma^{\mathtt{cur}} = \mathtt{cur}(r\gamma)$.

Finally, $\succsim_{\mathtt{cur}}$ contains beta because $\succsim$ does, and by Lemma 2.7: $\mathtt{cur}((\lambda x.s) \cdot t) = (\lambda x.\mathtt{cur}(s)) \cdot \mathtt{cur}(t) \succsim \mathtt{cur}(s)[x := \mathtt{cur}(t)] = \mathtt{cur}(s[x := t])$. □

Unfortunately, the transformation in the other direction (which is arguably more interesting) is trickier. We *can* still define a reduction pair which uncurries, but we have to phrase it in a careful way with regards to meta-terms.

To start, say that an applicative signature $\mathcal{F}$ "respects" an arity function ar if for all $f : \sigma_1 \to \ldots \to \sigma_n \to \iota \in \mathcal{F}$ with ι a base type, $n \geq ar(f)$. A meta-term s respects ar if any symbol f in s occurs in a context $f \cdot t_1 \cdots t_k$ with $k \geq ar(f)$.

Theorem 2.26. *Let $\mathcal{F}$ be signature where all function symbols have arity 0, and $(\succsim, \succ)$ a reduction pair on meta-terms over $\mathcal{F}$. Let ar be an arity function on $\mathcal{F}$ such that $\mathcal{F}$ respects ar, and let $\mathcal{F}^{ar}$ be the signature $\{f : [\sigma_1 \times \ldots \times \sigma_{ar(f)}] \longrightarrow \sigma_{ar(f)+1} \to \ldots \to \sigma_n \longrightarrow \iota \mid f : \sigma_1 \to \ldots \to \sigma_n \longrightarrow \iota \in \mathcal{F}\}$.*

For $R \in \{\succsim, \succ\}$, let the relation $R_{\mathtt{uncur}}$ be given by: $s \; R_{\mathtt{uncur}} \; t$ if and only if $\mathtt{uncur}(s) \; R \; \mathtt{uncur}(t)$ and either s and t are both terms or they both respect the arity function ar.

Then $(\succsim_{\mathtt{uncur}}, \succ_{\mathtt{uncur}})$ is a reduction pair, and $\succ_{\mathtt{uncur}}$ is monotonic if $\succ$ is.

Proof. Well-foundedness, transitivity, reflexivity and compatibility are all inherited from the corresponding properties of $\succsim$ and $\succ$, and monotonicity of $R_{\mathtt{uncur}}$ is inherited from monotonicity of R by a straightforward induction on the size of the context.

Meta-stability of $R_{\mathtt{uncur}}$ holds by meta-stability of R, and Lemmas 2.8 and 2.9: if l is a pattern of the right form and r a meta-variable, and γ a substitution on domain $FMV(l) \cup FMV(r)$, and if $s \; R_{\mathtt{uncur}} \; t$, then by definition of $R_{\mathtt{uncur}}$ we know that either s and t are both terms (in which case there is nothing to prove), or s and t respect ar. For such terms, it is easy to see that $s = \mathtt{cur}(\mathtt{cur}^{-1}(s))$ and the same for t. Moreover, by definition of $R_{\mathtt{uncur}}$ we know that $\mathtt{uncur}(s) \; R \; \mathtt{uncur}(t)$. Thus, $\mathtt{uncur}(s\gamma) = \mathtt{uncur}(\mathtt{cur}(\mathtt{uncur}(s))\gamma)$, which by Lemma 2.8 is equal to $\mathtt{uncur}(s)\gamma^{\mathtt{uncur}}$. By meta-stability of R (for clearly $\mathtt{uncur}(s)$ is still a pattern of the right form), this term $R \; \mathtt{uncur}(t)\gamma^{\mathtt{uncur}}$. By Lemma 2.9, and because $\succsim$ contains beta and is transitive and monotonic, this term $\succsim \; \mathtt{uncur}(t\gamma)$. Either by compatibility of $\succ$ and $\succsim$, or by transitivity of $\succsim$, we conclude that $s\gamma \; R_{\mathtt{uncur}} \; t\gamma$.

Finally, $\succsim_{\mathtt{uncur}}$ contains beta because $\succsim$ does, and by Lemma 2.9: $\mathtt{uncur}((\lambda x.s) \cdot t) = (\lambda x.\mathtt{uncur}(s)) \cdot \mathtt{uncur}(t) \succsim \mathtt{uncur}(s)[x := \mathtt{uncur}(t)] \succsim \mathtt{uncur}(s[x := t])$. □

Thus, we can use the uncurrying transformation freely, provided we do it only for meta-terms which respect the new arity function. Note that the resulting reduction pair is *not* fully stable, even if the original reduction pair $(\succsim, \succ)$ is.

In the transformation of Theorem 2.26 we had to use a special case for the meta-variables. This sets the tone to transform also the other transformations of Section 2.3.

η-expansion. The η-expansion transformation is trickier than currying or uncurrying, and we have to jump through some hoops to properly define the reduction pair. However, once defined, it is easy to use.

Theorem 2.27. *Let $(\succsim, \succ)$ be a reduction pair and define, for $R \in \{\succsim, \succ\}$, the relation $R^\uparrow$ as follows: $s\ R^\uparrow\ t$ if and only if:*

- *s and t have the same type $\sigma_1 \to \ldots \to \sigma_n \to \iota$ with ι a base type,* and
- *for all substitutions γ on domain $FMV(s) \cup FMV(t)$ and all k with $0 \leq k \leq n$ and all terms $q_1 : \sigma_1, \ldots, q_k : \sigma_k$ we have: $(s \cdot q_1 \cdots q_k)\!\uparrow^\eta\ \ R\ (t \cdot q_1 \cdots q_k)\!\uparrow^\eta$.*

Then $(\succsim^\uparrow, \succ^\uparrow)$ is also a reduction pair, and if $\succ$ is monotonic, then so is $\succ^\uparrow$.

If $s \succ t$ implies that $\lambda x.s \succ \lambda x.t$, then for all meta-terms l, r where l is a pattern of the form $f(\vec{s}) \cdot \vec{t}$ and r a meta-term of the same type, $l\ R^\uparrow\ r$ holds if $(l \cdot \vec{Z})\!\uparrow^\eta\ \ R\ (r \cdot \vec{Z})\!\uparrow^\eta$ for fresh meta-variables such that $l \cdot \vec{Z}$ has base type.

Proof. Well-foundedness, transitivity, reflexivity and compatibility are all inherited from the corresponding properties on $\succsim$ and $\succ$. For example compatibility: if $s \succ^\uparrow t \succsim^\uparrow q$, then $s,\ t$ and q all have the same type, and for all substitutions γ on domain $FMV(s) \cup FMV(t) \cup FMV(q)$ and terms $u_1, \ldots, u_k$: $(s\gamma \cdot \vec{u})\!\uparrow^\eta \succ (t\gamma \cdot \vec{u})\!\uparrow^\eta \succsim (q\gamma \cdot \vec{u})\!\uparrow^\eta$, so by compatibility of $\succ$ and $\succsim$ also $(s\gamma \cdot \vec{u})\!\uparrow^\eta \succ (q\gamma \cdot \vec{u})\!\uparrow^\eta$.

Meta-stability of both relations is evident immediately from the definition: taking into account that the result $s\gamma$ of a substitution does not include meta-variables, $s\ R^\uparrow\ t$ for meta-terms if and only if $s\gamma\ R^\uparrow\ t\gamma$ for all substitutions γ on $FMV(s) \cup FMV(t)$.

By Lemma 2.15(6) and because $\succsim$ contains beta and is monotonic and transitive, also $\succsim^\uparrow$ contains beta.

As for monotonicity, let us use induction on the form of C. Suppose R is monotonic, and $s\ R^\uparrow t$. Then certainly $s\!\uparrow^\eta\ \ R\ t\!\uparrow^\eta$ (choosing $k := 0$). Thus:

- $\lambda x.s\ R^\uparrow\ \lambda x.t$ since $(\lambda x.s)\!\uparrow^\eta = \lambda x.(s\!\uparrow^\eta)\ R\ \lambda x.(t\!\uparrow^\eta) = (\lambda x.t)\!\uparrow^\eta$ by monotonicity of R. Writing $norm(w) = \lambda x_1 \ldots x_m.w \cdot (x_1\!\uparrow^\eta) \cdots (x_m\!\uparrow^\eta)$, we have, for $k > 1$: $((\lambda x.s) \cdot q_1 \cdots q_k)\!\uparrow^\eta = norm((\lambda x.(s\!\uparrow^\eta)) \cdot (q_1\!\uparrow^\eta) \cdots (q_k\!\uparrow^\eta))$, which by monotonicity also $R\ norm((\lambda x.(t\!\uparrow^\eta)) \cdot (q_1\!\uparrow^\eta) \cdots (q_k\!\uparrow^\eta)) = ((\lambda x.t) \cdot q_1 \cdots q_k)\!\uparrow^\eta$.
- We derive that $q \cdot s\ R^\uparrow\ q \cdot t$ and $f(q_1, \ldots, s, \ldots, q_n)\ R^\uparrow f(q_1, \ldots, t, \ldots, q_n)$ in a very similar way, except that these derivations do not need a special case for $k = 0$.
- We see that $s \cdot q\ R^\uparrow\ t \cdot q$ because by definition of $s\ R^\uparrow t$ we have, for all $u_2, \ldots, u_k$, that $(s \cdot q \cdot \vec{u})\!\uparrow^\eta\ \ R\ (t \cdot q \cdot \vec{u})\!\uparrow^\eta$.

Thus, $(\succsim^\uparrow, \succ^\uparrow)$ is also a reduction pair, and $\succ^\uparrow$ is monotonic if $\succ$ is.

For the second claim, let l be a pattern of the right form and r a meta-term of the same type $\sigma_1 \to \ldots \to \sigma_n \to \iota$. To see that $l\ R^\uparrow\ r$ we must see that for all substitutions γ on domain $FMV(l) \cup FMV(r)$ and terms $s_1 : \sigma_1, \ldots, s_k : \sigma_k$ with $0 \leq k \leq n$ we have: $(l\gamma \cdot s_1 \cdots s_k)\!\uparrow^\eta\ \ R\ (r\gamma \cdot s_1 \cdots s_k)\!\uparrow^\eta$. Write $l' := l \cdot Z_1 \cdots Z_n$ and $r' := r \cdot Z_1 \cdots Z_n$ for fresh meta-variables $Z_1, \ldots, Z_n$, and suppose that $l'\!\uparrow^\eta\ R\ r'\!\uparrow^\eta$. Then l' is still a pattern of the right form, so:

$$
\begin{aligned}
&(l\gamma \cdot s_1 \cdots s_k)\uparrow^\eta \\
&= \lambda x_{k+1} \ldots x_n.(l\gamma \cdot s_1 \cdots s_k \cdot x_{k+1} \cdots x_n)\uparrow^\eta \\
&= \lambda x_{k+1} \ldots x_n.(l'\delta\uparrow^\eta) \text{ where } \delta := \gamma \cup [Z_1 := s_1, \ldots, Z_k := s_k, \\
&\qquad\qquad\qquad\qquad\qquad\qquad Z_{k+1} := x_{k+1}, \ldots, Z_n := x_n] \\
&= \lambda x_{k+1} \ldots x_n.l'\uparrow^\eta \delta^\uparrow \text{ by Lemma 2.15(4)} \\
&R\ \lambda x_{k+1} \ldots x_n.r'\uparrow^\eta \delta^\uparrow \text{ by meta-stability (and } \lambda\text{-monotonicity) of } R \\
&\gtrsim \lambda x_{k+1} \ldots x_n.(r'\delta)\uparrow^\eta \text{ by Lemma 2.15(5) and because } \gtrsim \text{ contains beta} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{and is transitive and monotonic} \\
&= \lambda x_{k+1} \ldots x_n.(r\gamma\uparrow^\eta \cdot s_1\uparrow^\eta \cdots s_k\uparrow^\eta \cdot x_{k+1}\uparrow^\eta \cdots x_n\uparrow^\eta) \\
&\gtrsim \lambda x_{k+1} \ldots x_n.(r\gamma \cdot s_1 \cdots s_k \cdot x_{k+1} \cdots x_n)\uparrow^\eta = (r\gamma \cdot s_1 \cdots s_k)\uparrow^\eta
\end{aligned}
$$

Either by compatibility of $\succ$ and $\gtrsim$, or by transitivity of $\gtrsim$, we conclude that $(l\gamma \cdot s_1 \cdots s_k)\uparrow^\eta \ R\ (r\gamma \cdot s_1 \cdots s_k)\uparrow^\eta$ as required. □

Although the definition of this reduction pair is somewhat tricky, in practice we simply use the second part of the theorem: we can freely η-expand the meta-terms under consideration before orienting them with a reduction pair.

Flattening Simple Meta-variables. Next, we move on to the transformation of Section 2.3.3. This transformation has the nice feature that it leaves terms alone; only meta-terms with meta-variables are altered.

Theorem 2.28. *Let $(\gtrsim, \succ)$ be a reduction pair, and let* `flat` *be the function from Transformation 2.18. For $R \in \{\gtrsim, \succ\}$, let $s\ R_{\texttt{flat}}\ t$ if and only if s has simple meta-applications, and $\texttt{flat}(s)\ R\ \texttt{flat}(t)$. Then $(\gtrsim_{\texttt{flat}}, \succ_{\texttt{flat}})$ is also a reduction pair, and $\succ_{\texttt{flat}}$ is monotonic if $\gtrsim_{\texttt{flat}}$ is.*

Proof. Since $\gtrsim_{\texttt{flat}}$ and $\succ_{\texttt{flat}}$ define the same relation as $\gtrsim$ and $\succ$ when restricted to terms, all properties but meta-stability are obvious. Since l has simple meta-applications whenever $l\ _{\texttt{flat}}\ r$, this is proved exactly as in Theorem 2.19. □

Type Changing. The last, and quite possibly simplest, of the transformations of Section 2.3 is the type changing transformation:

Theorem 2.29. *Let $(\gtrsim, \succ)$ be a reduction pair on (meta-)terms over $\mathcal{F}^\zeta, \mathcal{V}^\zeta, \mathcal{M}^\zeta$, and define, for $R \in \{\gtrsim, \succ\}$, the relation R_ζ on (meta-)terms over $\mathcal{F}, \mathcal{V}, \mathcal{M}$ as follows: $s\ R_\zeta t$ iff $s\ R\ t$ when s and t are seen as terms over $\mathcal{F}^\zeta, \mathcal{V}^\zeta, \mathcal{M}^\zeta$.*

Then $(\gtrsim_\zeta, \succ_\zeta)$ is also a reduction pair, and if $\succ$ is monotonic, then so is $\succ_\zeta$.

Proof. The most important observation, that these relations are well-defined, is because any (meta-)term that is well-typed in $\mathcal{F}, \mathcal{V}, \mathcal{M}$ is also well-typed in $\mathcal{F}^\zeta, \mathcal{V}^\zeta, \mathcal{M}^\zeta$. Well-foundedness, transitivity, reflexivity and compatibility are all inherited from the corresponding properties on $\gtrsim$ and $\succ$, and $\gtrsim_\zeta$ contains beta because $\gtrsim$ does. Monotonicity of either relation R is also inherited from monotonicity of R_ζ (if applicable), as is evident with a trivial induction on the context. Meta-stability is inherited because if the term $s\gamma$ is defined in the original system, then it is defined, and the same term, in the type-changed system. □

Inverse Compatibility. Finally, let us consider one more reduction pair transformation. The notion of *compatibility* used here is less general than commonly used: normally, a pair of relations $(\succsim, \succ)$ is compatible if $\succ \cdot \succsim$ is a subrelation of $\succ$, *or* $\succsim \cdot \succ$ is. Let us call the latter possibility an *inverse compatible* pair of relations. The definitions here do not permit a reduction pair where the relations are inverse compatible rather than compatible, even though the soundness proof of for instance rule removal goes through almost unmodified with this definition.

The reason for this choice is twofold: first, allowing inverse compatibility for a reduction pair necessarily complicates transformations of reduction pairs: neither the uncurrying transformation (Theorem 2.26), nor the η-expansion (Theorem 2.27 or meta-variable flattening (Theorem 2.28) transformations given in this section work if the underlying reduction pair is not compatible. These transformations could still be defined if $\succsim \cdot \succ$ is included in $\succ$, but their definition would have to be a bit more convoluted.

This is not sufficient argument to disallow this form of compatibility altogether – especially not since there are inverse compatible reduction pairs in the literature which are not reduction pairs with this definition of compatibility (see e.g. [22]). However, the second reason justifies the choice: we can transform an inverse compatible reduction pair into a normal reduction pair.

Theorem 2.30. *Let $\succsim$ be a quasi-ordering and $\succ$ a well-founded ordering on terms, such that both relations are meta-stable, and $\succsim$ contains beta and is monotonic. Suppose moreover that $\succsim \cdot \succ$ is a subrelation of $\succ$.*

Let $s \succ' t$ if for all substitutions γ on domain $FMV(s) \cup FMV(t)$: $s\gamma \succ \cdot \succsim t\gamma$. Then $(\succsim, \succ')$ is a reduction pair, and $\succ'$ is monotonic if $\succ$ is.

Proof. All required properties of $\succsim$ are already given, so we merely need to see that $\succ'$ is transitive, well-founded, meta-stable, compatible with $\succsim$ and monotonic if $\succ$ is.

For transitivity, suppose $s \succ' t \succ' q$, so for all substitution γ on domain $FMV(s) \cup FMV(t) \cup FMV(q)$ there are terms u, v such that $s\gamma \succ u \succsim t\gamma \succ v \succsim q\gamma$. By inverse compatibility of $(\succsim, \succ)$, this gives that $s\gamma \succ u \succ v \succsim q\gamma$. By transitivity of $\succ$ we thus have $s\gamma \succ v \succsim q\gamma$ for all γ, so $s \succ' q$.

For well-foundedness, suppose $s_1 \succ' s_2 \succ' \ldots$. We can safely assume that all s_i are terms, otherwise we substitute them and still have an infinite decreasing chain. By definition of $\succ'$ we can find $t_1, t_2, \ldots$ such that $s_1 \succ t_1 \succsim s_2 \succ t_2 \succsim s_3 \succ \ldots$ But then by inverse compatibility, $t_1 \succ t_2 \succ \ldots$, contradicting well-foundedness of $\succ$.

Meta-stability is immediately obvious from the definition. Monotonicity (if required) is inherited from monotonicity of $\succ$ and $\succsim$. □

Note that, if we can prove that $l \succ r$ in an inverse compatible reduction pair, and l is a pattern of the right form, then also $l \succ' r$ (by reflexivity of $\succsim$ and meta-stability). Thus, we can use an inverse compatible reduction pair just as we would use a normal reduction pair.

2.5 Overview

In this chapter, we have defined the formalism of Algebraic Functional Systems with Meta-variables. We have seen a number of transformations on this formalism, which allow us for instance to swap between notations, or use η-long forms. This gives us a lot of freedom in the use of AFSMs. We have also discussed how reduction pairs behave in the setting of AFSMs.

AFSMs are unusual in that they have both meta-variables and β-reduction, both function application and normal application. Typical applications of higher-order rewriting in practice do not need all this. However, AFSMs were not designed as a model of any particular behaviour. Rather, the AFSM formalism should only be seen as a tool to study termination of higher-order rewriting. As we will see in Chapter 3, systems in other formalisms can often be transformed to AFSMs without losing non-termination. By deriving results for AFSMs, we can obtain termination results for all these formalisms in one go.

Higher-order Formalisms

3

Or, Wait, what about all those other ways?

The first problem anyone who wishes to study higher-order term rewriting runs into, is the lack of a standard formalism. There is not one standard – there are about a dozen: CSs, CRSs, ERSs, HRSs, PRSs, AFSs, ADTSs, IDTSs, CRSXs, STRSs, STTRSs ... Some of these are strictly included in others, but most are incomparable. It is often not obvious whether a result in one formalism can be adapted to another.

Considering that one of the reasons to study higher-order term rewriting is to obtain uniform proofs, which can be used in all application areas of rewriting rather than using a different modelling in every field, it is somewhat ironic that not two papers on higher-order term rewriting seem to use exactly the same formalism.[1] In the study of confluence, Nipkow's *Higher-order Rewriting Systems* (HRSs) [101] and Klop's *Combinatory Reduction Systems* are dominant, but in termination research Jouannaud's and Okada's *Algebraic Functional Systems* (AFSs) [63] are equally popular. But, as the latter is a quite permissive formalism, many results are limited to some restriction (with some restrictions more fundamental than others), and several other formalisms have been proposed. to deal with the weaknesses of either HRSs, CRSs or AFSs.

Since the aim of this thesis is to provide general termination results, and to place existing results in a larger framework, I have elected *not* to choose any of these existing frameworks, but instead to define techniques for the new formalism of Algebraic Functional Systems with Meta-variables defined in Chapter 2.2. At first this seems counter to the wish of being general: the last thing the higher-order term rewriting community is waiting for is yet another formalism. However, as stated before, the aim is *not* to promote this formalism. Rather, the goal is to derive termination results for most of the common higher-order formalisms at once. As we will see, systems in other styles of rewriting can be transformed into an AFSM without losing non-termination. Thus, the termination methods defined in this thesis will be immediately applicable to CSs, CRSs, AFSs, PRSs, IDTSs and, probably, CRSXs as well.

[1] This is a slight exaggeration.

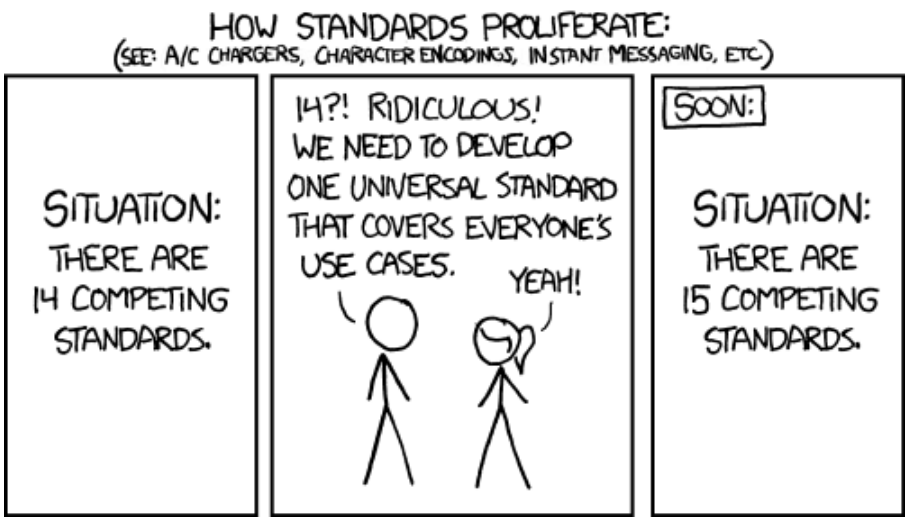


Figure 3.1: Source: http://www.xkcd.com/927/

Chapter Setup. In Section 3.1, I will attempt to give a chronological overview of the existing higher-order formalisms. The rest of the chapter is dedicated to explaining the most relevant formalisms (that is, those higher-order formalisms which have both simple types and bound variables) in some more detail: Blanqui's IDTSs (3.2), Nipkow's PRSs (3.3), Jouannaud's and Okada's AFSs (3.4) and Rose's CRSXs (3.5). We will see how to transform a system in those formalisms to an AFSM without losing non-termination. In some cases there is also a transformation in the other direction. Sections 3.6 and 3.7 treat two untyped systems (Contraction Schemes and Combinatory Reduction Systems) which can nevertheless be transformed into typed systems without affecting termination. It is very likely that most, or even all, other formalisms discussed in Section 3.1 can also be transformed into AFSMs – to some extent. For example, HRSs can *not* in general be expressed as AFSMs, but the common subclass of PRSs can be. Several other formalisms are untyped, and may well be expressible as an AFSM, but since termination is not likely to be a large issue in untyped systems (recall that the untyped λ-calculus is non-terminating!), this seems of relatively little interest.

An overview of the transformations in this chapter is given in Figure 3.2.

A solid arrow indicates that systems in the first formalism can be represented in the second without affecting termination. A dashed arrow indicates that systems in the first can be represented in the second without losing non-termination, but termination may be lost. If the arrow is marked with a star, termination of systems in the first is equivalent to a special termination question in the second (for example, termination using a beta-first reduction strategy). A dotted arrow indicates that *some* systems in the first can be represented in the second without losing non-termination. The question mark for CRSXs indicates that, although

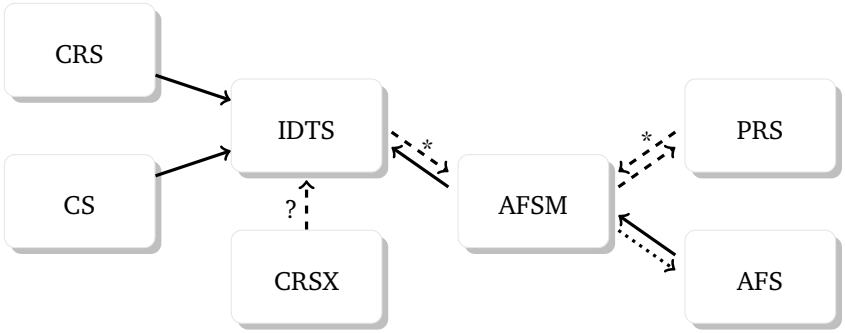


Figure 3.2: Transformations of higher-order formalisms

it is very probable that a good non-termination-preserving transformation exists, the exact formal details for the transformation are not presented in this work.

It is worth noting that every formalism is connected to AFSMs with a solid or dashed arrow. Thus, we can transform any system in one of these formalisms to an AFSM, and if the result is terminating, then so is the original.

The sections in this Chapter are only marginally inter-related. It is possible to read any of them without being familiar with the rest, although understanding of the preliminaries in Chapter 2 is recommended, and the sections on CRSXs (3.5), CSs (3.6) and CRSs (3.7) refer to the definition of IDTSs (Section 3.2).

3.1 A History of Higher-order Formalisms

Omitting logics such as the λ-calculus, the first definition of a higher-order term rewriting system is Aczel's definition of *Contraction Schemes* in 1978 [4]. This definition extends first-order term rewriting systems with binders and *meta-variables*, thus allowing higher-order functions to be defined. In a contraction scheme, abstractions are not considered terms by themselves, although they can be used as subterms. While contraction schemes are not formally equipped with a type system, the restrictions in term formation guarantee that a simple typing, as described in Section 2.1.2, can always be derived.

Extending on the ideas of Contraction Schemes, Klop defines the framework of *Combinatory Reduction Systems*, and extensively studies their properties in his 1980 thesis [70]. An alternative definition is given in 1993 [71], and has become the standard form of CRSs. In combinatory reduction systems (as in most of the later formalisms), abstractions are considered as terms, and as a consequence CRSs are not implicitly typable.

Next comes Khasidashvili's formalism of *Expression Reduction Systems*, which was published in 1990 [68]. Unlike most other higher-order systems, the syntax of expression reduction systems allows matching on variables, and rules use an explicit syntax for substitution; $f(\lambda x.X, Y) \Rightarrow X[x \leftarrow Y]$ is a typical rule (here, the variable x may occur in whatever is substituted for X). Expression reduction

systems are untyped, and there is no implicit typing. Various definitions and variations of ERSs exist, such as for instance *Context-sensitive Conditional Expression Reduction Systems* [69], which adds context-sensitive rewriting to the formalism.

The first higher-order formalisms which use types were introduced simultaneously at LICS 1991: Nipkow's *Higher-order Rewriting Systems* [101] and Jouannaud's and Okada's *Algebraic Functional Systems* [60] (although this name for the formalism did not appear in the literature until later).

In Nipkow's HRSs, terms are equivalence classes modulo $\alpha\beta\eta$. The original 1991 definition uses a *pattern* restriction on rules, which guarantees that the rewrite relation is decidable. Following Wolfram [127] this restriction was dropped in [98]; the original definition is still popular as the class of *pattern HRSs* (PRSs). HRSs use simple types.

Jouannaud's and Okada's AFSs are originally defined with a polymorphic type structure, although later variations use different typings. Later definitions also have other modifications, which solve some problems present in the original definition. AFSs are the first formalism to use both application and functional application. $\Rightarrow_\beta$ and, in some versions, $\hookrightarrow_\eta$ or its reverse are separate rule schemes added to every system. The definition of AFSs in [63], but restricted to simple types, has become the basis for the first higher-order category in the annual *termination competition* [125] in 2010.

The next few formalisms defined in the nineties do not have an explicit typing. In 1993 *Interaction Systems* are introduced by Laneve [91], to study the theory of optimal reductions as defined by Levy [93] in a setting that is more general than λ-calculus. Interaction Systems use a syntax similar to ERSs, but can be seen as a subclass of Contraction Schemes. The same year also sees Takahashi's *Conditional λ-Calculus* [116], another form of untyped rewriting which uses the explicit substitution syntax of ERSs, as well as *conditions* for rewriting. For example, the η-shortening rule (the reverse of $\hookrightarrow_\eta$) can be expressed as a CLC which has a rule $\lambda x.M \cdot x \Rightarrow M$ with condition $x \notin FV(M)$.

To deal with the vast number of higher-order formalisms, van Oostrom and van Raamsdonk introduce the framework of Higher-Order Rewrite Systems in 1994, best described in [102]. This framework aims to unify the existing formalisms by separating the notion of a *substitution calculus* from the notion of *terms*. Examples of a substitution calculus are the typed λ-calculus (for HRSs) or the untyped λ-calculus (for CRSs). The authors show the usefulness of this formalism by defining a notion of orthogonality, and obtaining various confluence results.

These HORSs are not, however, the end for other formalisms; for while the abstractness of HORSs offers an unprecedented generality, this comes at the price of easy reasoning. It is harder to express examples, and results like the recursive path ordering (as described in Chapter 5) cannot easily be extended – the substitution calculus plays too large a role in the question of termination. And it is termination which became an object of active research at the end of the 90s and in the following decade.

In 1996, Rose introduces *Combinatory Reduction Systems with eXtensions* [107], a formalism aimed at business applications, which has been significantly altered and improved over the years. Despite the name, these CRSXs are closest to Aczel's contraction schemes, and are implicitly typable. CRSXs allow matching on variables like in ERSs, and rules may introduce free variables in the right-hand side, a feature which is used for instance to simulate memory allocation. The most alien notion to other formalisms is the use of *lookup tables*, a sort of mapping where variables and strings can be assigned a (term) value. In recent definitions, CRSXs are equipped with a (polymorphic) type discipline.

Abstract Data Type Systems, defined by Jouannaud and Okada in 1997 [61], are designed to facilitate building formal specification languages, which integrate computations and proofs within a single framework. These ADTSs, also called *Inductive Data Type Systems*, define types by enumerating their constructors, for instance `Bool := True : Bool | False : Bool`. Certain restrictions on what can be defined make it easier to prove termination. However, if we look past the syntax, the underlying formalism is just the AFS framework.

In [13], Blanqui extends this formalism with CRS-like meta-variables. This formalism, still called *Inductive Data Type Systems* even though the necessity to declare types in an inductive way is dropped, is declared in a monomorphic way, and can be seen as the extension of CRSs with simple types.

As far as truly higher-order systems (including λ-abstraction) are concerned, this is where the proliferation ends. Termination results in the last decade have been designed for (variations of) the existing formalisms, in particular HRSs and AFSs, which use an explicit form of typing. This is for good reason: systems expressed in an untyped formalism tend not to be terminating. For example, the common example of a higher-order function, `map`, could be expressed by the CRS:

$$\begin{array}{rcl} \texttt{map}(\lambda x.F(x), \texttt{nil}) & \Rightarrow & \texttt{nil} \\ \texttt{map}(\lambda x.F(x), \texttt{cons}(h,t)) & \Rightarrow & \texttt{cons}(F(h), \texttt{map}(\lambda x.F(x), t)) \end{array}$$

As a CRS, this system is non-terminating: the term $\texttt{map}(\omega, \texttt{cons}(\omega, \texttt{nil}))$ admits a self-loop, where $\omega = \lambda x.\texttt{map}(x, \texttt{cons}(x, \texttt{nil}))$. However, as an IDTS (which is a CRS with typing), this example does terminate; the offending term is not typable.

Applicative Rewriting. Another style of rewriting which is sometimes referred to as higher-order rewriting is *applicative rewriting*. There are several formalisms for applicative systems. First and foremost, one might consider normal (first-order) term rewriting systems where all function symbols have arity 0 except a symbol "app" which has arity 2. Already in 1996 [67] it was known that when such a system is the curried form of a normal TRS, termination is equivalent to its uncurried form. In [56] an uncurrying termination for an arbitrary applicative TRS is given which preserves and reflects termination.

In 2001 [86] Kusakari extends these systems with simple types, which gives *Simply-typed Term Rewriting Systems* (STRSs). Later extensions also include a product type $\sigma_1 \times \ldots \times \sigma_n$ and a tuple symbol.

In a similar but unrelated line of work, Yamada defines *Simply Typed Term Rewriting Systems* (STTRSs) [129], also in 2001. STTRSs are applicative systems, but have a typing mechanism based around product types.

For termination, applicative systems in the various styles are convenient, since β-reduction significantly complicates termination proofs. Consequently, these systems are closer to first-order TRSs than the higher-order systems discussed in this section. In fact, using transformations like variable instantiation (as done in [6, 7]) applicative systems can be transformed into (many-sorted) TRSs.

As mentioned in the introduction, a sceptic might question why we should bother with λ-abstraction, and thus the whole higher-order field: would applicative TRSs not suffice? The answer depends on which question you are interested in. When we only seek to study termination of fixed terms, rather than termination of *all* terms, yes, applicative TRSs usually do suffice. Most higher-order term rewriting systems can be converted into an applicative system. For example, consider a higher-order rewriting system (an AFS) with the following rules:

$$\begin{array}{rcl} \mathtt{app}(\mathtt{abs}(F), T) & \Rightarrow & F \cdot T \\ \mathtt{start}(T) & \Rightarrow & \mathtt{app}(\mathtt{abs}(\lambda x.\mathtt{app}(x, T)), \mathtt{0}) \end{array}$$

The corresponding applicative system simulates the abstraction in the second rule by adding a new symbol `tmp`.

$$\begin{array}{rcl} \mathtt{app} \cdot (\mathtt{abs} \cdot F) \cdot T & \Rightarrow & F \cdot T \\ \mathtt{start} \cdot T & \Rightarrow & \mathtt{app} \cdot (\mathtt{abs} \cdot (\mathtt{tmp} \cdot T)) \cdot \mathtt{0} \\ \mathtt{tmp} \cdot T \cdot T' & \Rightarrow & \mathtt{app} \cdot T' \cdot T \end{array}$$

Since the term $\mathtt{start}\cdot\mathtt{0}$ is terminating in the applicative system, the term $\mathtt{start}(\mathtt{0})$ is terminating in the original. In fact, this simply-typed applicative TRS is terminating altogether. Thus, any term in the original system *which does not contain λ-abstractions* is terminating as well. If we are interested in termination of a term with binders, for instance $\mathtt{app}(\mathtt{abs}(\lambda x.\mathtt{app}(x, x)), \mathtt{abs}(\lambda x.\mathtt{app}(x, x)))$, we can still use applicative systems: we study the termination question of this term by adding a rule $\mathtt{d}\cdot x \Rightarrow \mathtt{app}(x, x)$ and consider termination of the term $\mathtt{app}\cdot(\mathtt{abs}\cdot\mathtt{d})\cdot(\mathtt{abs}\cdot\mathtt{d})$.

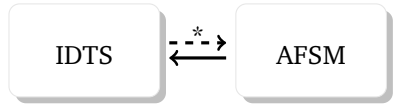
Thus, using applicative systems we can more easily study termination of specific terms, or termination of all terms without binders. This is sufficient for some purposes, but not for all. If we want to prove termination for *all* terms, not just those without binders, we would have to add infinitely many new rules, and then we are no better off than in the full higher-order system we started with.

Formal Languages. A notable omission from the list of higher-order term rewriting formalisms are the formal languages underlying functional programming languages such as Haskell and ML, or the underlying language of theorem provers such as Coq (the calculus of inductive constructions) and Isabelle (higher order logic). These formal languages typically use more sophisticated typing mechanisms, and as such probably cannot be transformed directly into an AFSM.

It is likely that these languages, which form an important application area of higher-order term rewriting, can be *partly* expressed as AFSMs. However, I have not done this study; the transformation to (or, alternatively, extension of termination methods for) AFSM from each of these formal languages seems like a good topic for dedicated research focused on those particular formal systems.

Systems of Interest. As discussed in the introduction, the focus in this thesis is on *full termination*, so termination of all terms in the rewriting formalism. Therefore, results on applicative systems are in general not reusable. Since, moreover, untyped systems are in general not terminating, the main systems of interest are HRSs, AFSs, the last version of IDTSs and CRSXs, as well as those systems where an implicit typing can be inferred. Unlike HORSs, AFSMs are syntax-wise very close to the formalisms we are interested in, and the results in this thesis should show that it is not too difficult to obtain strong termination techniques for them.

3.2 Inductive Data Type Systems



The AFSM formalism used in this work is an extension (or, depending on your point of view, a subset) of the IDTS formalism proposed by Blanqui in [13].

The name "Inductive Data Type Systems" is somewhat misleading: it refers not so much to the formalism as to the type restrictions posed in the paper where they were introduced (these restrictions were posed solely for the definition of a general schema). However, we will stick to this name as there is no alternative name available.

3.2.1 Definition

Inductive Data Type Systems use simple types, and terms are generated as in Definition 2.2, but without clause (app). Rewrite rules, too, and the rewrite relation, are defined as in AFSMs. Thus, an IDTS can be seen as an AFSM with a restriction on (meta-)term formation: only *application-free* meta-terms are permitted (and all rules are application-free). Here, a meta-term is application-free if it does not contain the application operator. In this way, AFSMs could be seen as the *extension* of IDTSs with an application operator and β-reduction. The *subset* view will be explained in Section 3.2.3.

Due to this similarity it might seem at first that an IDTS is just an AFSM where $\mathcal{F}$ and $\mathcal{R}$ have certain restrictions. However, this is not exactly the case. Since all terms, not just the rules, must be application-free, it is not in general possible to create non-variable terms of an arbitrary type. This may have a crucial impact on termination: the same system could be terminating when interpreted as an IDTS and non-terminating when interpreted as an AFSM.

Consider, for example, an IDTS with four function symbols,

$$\begin{array}{lcl} \mathtt{a} & : & \mathtt{nat} \\ \mathtt{g} & : & [\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}] \longrightarrow \mathtt{nat} \\ \mathtt{f} & : & [\mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{bool} \\ \mathtt{h} & : & [\mathtt{nat}] \longrightarrow \mathtt{bool} \end{array}$$

And a single rule:

$$\mathtt{f}(\mathtt{a}, \mathtt{g}(\lambda xy.F(x,y))) \Rightarrow \mathtt{h}(F(\mathtt{a}, \mathtt{g}(\lambda xy.F(x,y))))$$

Note that an application-free term of type `nat` cannot contain the symbol `f` anywhere and therefore cannot be reduced. Consequently, a reduction starting in an IDTS-term over $\mathcal{F}$ has at most one step, so this IDTS is terminating.

However, if we consider this same system as an AFSM, then application is allowed in term formation. In this case, there is an infinite reduction by instantiating $F(x,y)$ with $z \cdot \mathtt{f}(x,y)$ if $z : \mathtt{bool} \to \mathtt{nat}$:

$$\begin{array}{l} \mathtt{f}(\mathtt{a}, \mathtt{g}(\lambda xy.z \cdot \mathtt{f}(x,y))) \\ \Rightarrow_{\mathcal{R}} \mathtt{h}(z \cdot \underline{\mathtt{f}(\mathtt{a}, \mathtt{g}(\lambda xy.z \cdot \mathtt{f}(x,y)))}) \\ \Rightarrow_{\mathcal{R}} \mathtt{h}(z \cdot \mathtt{h}(z \cdot \underline{\mathtt{f}(\mathtt{a}, \mathtt{g}(\lambda xy.z \cdot \mathtt{f}(x,y)))})) \\ \Rightarrow_{\mathcal{R}} \ldots \end{array}$$

Even in systems with only one base type, application might make a crucial difference to termination, if the function symbols have an arity of at most one:

$$\begin{array}{lcl} \mathtt{0} & : & \mathtt{nat} \\ \mathtt{f} & : & [\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}] \longrightarrow \mathtt{nat} \\ \mathtt{g} & : & [\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}] \longrightarrow \mathtt{nat} \end{array}$$

$$\begin{array}{rcl} \mathtt{f}(\lambda xy.Z(x,y)) & \Rightarrow & Z(\mathtt{g}(\lambda xy.Z(x,y)), \mathtt{0}) \\ \mathtt{g}(\lambda xy.Z(x,y)) & \Rightarrow & Z(\mathtt{0}, \mathtt{f}(\lambda xy.Z(x,y))) \end{array}$$

Since both function symbols have arity 1, an application-free term contains at most one variable (bound or free), which is not the case when application is allowed. Seen as an IDTS, this system is terminating. Intuitively, if s does not contain any bound variables (except maybe in binders λx), then any rewrite step decreases the size of the term. If s *does* contain a bound variable x, then s may have two forms which allow a non-decreasing step:

- $s = C[f(\lambda xy.s')] \Rightarrow_{\mathcal{R}} C[s'[x := g(\lambda xy.s')]]$
- $s = C[g(\lambda yx.s')] \Rightarrow_{\mathcal{R}} C[s'[x := f(\lambda yx.s')]]$

Here, y is ignored because it cannot occur in s' when s' already contains x. After this step, the occurrence of x in s' is still the only leaf in the term, but now s

does not have either of the forms above, nor reduces to it. Thus, any further step decreases the size of the term, so reductions must terminate.

However, if we allow the application operator, then there is a counterexample for termination: instantiate $Z(x, y)$ by $z \cdot x \cdot y$. Then:

$$\begin{aligned}
&\underline{f(\lambda xy.z \cdot x \cdot y)} \\
&\Rightarrow_{\mathcal{R}} z \cdot g(\lambda xy.z \cdot x \cdot y) \cdot 0 \\
&\Rightarrow_{\mathcal{R}} z \cdot (z \cdot 0 \cdot \underline{f(\lambda xy.z \cdot x \cdot y)}) \cdot 0 \\
&\Rightarrow_{\mathcal{R}} \cdots
\end{aligned}$$

3.2.2 From IDTS to AFSM

The examples above demonstrate that IDTSs and AFSMs are not quite the same. However, termination of an AFSM does imply its termination on the set of application-free terms. Thus, termination results on AFSMs transfer to IDTSs:

Theorem 3.1. *An IDTS $(\mathcal{F}, \mathcal{R})$ is terminating if and only if the corresponding AFSM $(\mathcal{F}, \mathcal{R})$ is terminating on application-free terms.*

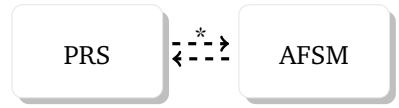
In this work, we will not really consider techniques which take the added information that terms are application-free into account. The main place where it might matter is in the dependency graph of Chapter 7.4: if we are only considering termination of application-free terms, then we may be able to omit some edges from the dependency graph.

3.2.3 From AFSM to IDTS

For the other direction, we change viewpoints. We could see an AFSM as a special kind of IDTS by considering IDTSs which contain, for all types σ, τ, a symbol $@^{\sigma\to\tau} : [\sigma \to \tau \times \sigma] \longrightarrow \tau$ and rules $@^{\sigma\to\tau}(\lambda x.F(x), Z) \Rightarrow F(Z)$. Thus, application and β-reduction are simulated.

Any AFSM can be seen as an IDTS of this form and thus, termination methods on IDTSs immediately extend. True, this IDTS is infinite, but the infinity comes from a countable set of very similar rules. When using techniques like the iterative path ordering discussed in Chapter 5, the constraints from such rules can easily be eliminated.

3.3 Pattern Higher-order Rewrite Systems



Another important style of higher-order rewriting, used in particular in confluence research, are *Nipkow's HRSs* [101], and the subclass of *PRSs*. This formalism uses simply-typed terms, where term equality is defined modulo β and η.

3.3.1 Definition

An HRS directly extends λ-terms with typed function symbols and rules. Formally, assume given a set $\mathcal{F}$ of simply-typed function symbols and a set $\mathcal{V}$ of simply-typed variables which contains countably many variables of each type. *Pre-terms* over $\mathcal{F}, \mathcal{V}$ are those expressions s for which we can derive $s : \sigma$ using the clauses:

$$\begin{array}{lll} a : \sigma & \text{if} & a : \sigma \in \mathcal{V} \cup \mathcal{F} \\ s \cdot t : \tau & \text{if} & s : \sigma \to \tau \text{ and } t : \sigma \\ \lambda x.s : \sigma \to \tau & \text{if} & x : \sigma \in \mathcal{V} \text{ and } s : \tau \end{array}$$

Note that this only differs from the definition of simply-typed λ-terms by including typed function symbols. As in the λ-calculus, we consider application as left-associative, and omit unnecessary parentheses. A *term* is a pre-term in η-long β-normal form. Every pre-term s corresponds to a unique term $s{\updownarrow}^{\eta}_{\beta}$.[2]

Using the normal definition of substitution, a type-respecting mapping $[x_1 := s_1, \ldots, x_n := s_n]$, the result of applying a substitution γ on a term t is the *pre-term* t with all occurrences of some x_i replaced by s_i. A context is a *term* with a single occurrence of a typed symbol $\Box_\sigma$; if σ is a base type then the pre-term $C[s]$ obtained by replacing $\Box_\sigma$ by some term of the same type is also a term.

A rewrite rule is a pair $l \Rightarrow r$ of terms which have the same base type, such that $FV(r) \subseteq FV(l)$ and l has the form $f \cdot l_1 \cdots l_n$. The rewrite relation on terms is given by:

$$C[l\gamma{\updownarrow}^{\eta}_{\beta}] \Rightarrow_{\mathcal{R}} C[r\gamma{\updownarrow}^{\eta}_{\beta}] \quad (l \Rightarrow r \in \mathcal{R},\ \gamma \text{ a substitution},\ C[] \text{ a context})$$

Unfortunately, this rewrite relation is in general not decidable. Hence attention is usually restricted to *pattern* HRSs, where the left-hand sides of rules are "patterns". A term l is a pattern in an HRS if for every subterm $x \cdot t_1 \cdots t_n$ with x free in l and $n > 0$, the t_i are the η-long forms of distinct bound variables. Patterns are defined by Miller [99], who proves that unification (and hence matching) modulo β is decidable for patterns. This restriction is very similar to the patterns used in AFSMs (and in CRSs and IDTSs). We will refer to Pattern HRSs as *PRSs*.

Example 3.2. The typed λ-calculus can be encoded as an infinite PRS with symbols $@^{\sigma\to\tau} : (\sigma \to \tau) \to \sigma \to \tau$ for all types σ, τ, and corresponding rules $@^{\sigma\to\tau} \cdot (\lambda x.y \cdot x) \cdot z \Rightarrow y \cdot z$. Note that $\lambda x.y \cdot x$ in the left-hand side also matches on, for instance, $\lambda x.g(x, x)$, since this can be written as $\lambda x.((\lambda z.g(z, z)) \cdot x){\updownarrow}^{\eta}_{\beta}$.

Example 3.3. We can represent the system `map` from Example 2.3 by a PRS with symbols $\texttt{nil} : \texttt{list}$, $\texttt{cons} : \texttt{nat} \to \texttt{list} \to \texttt{list}$, $\texttt{map} : (\texttt{nat} \to \texttt{nat}) \to \texttt{list} \to \texttt{list}$, $\texttt{0} : \texttt{nat}$, $\texttt{s} : \texttt{nat} \to \texttt{nat}$, and rules:

$$\begin{array}{lcl} \texttt{map} \cdot (\lambda x.F \cdot x) \cdot \texttt{nil} & \Rightarrow & \texttt{nil} \\ \texttt{map} \cdot (\lambda x.F \cdot x) \cdot (\texttt{cons} \cdot y \cdot z) & \Rightarrow & \texttt{cons} \cdot (F \cdot y) \cdot (\texttt{map} \cdot (\lambda x.F \cdot x) \cdot z) \end{array}$$

[2] Alternatively, terms might be defined as equivalence classes of pre-terms modulo β/η-equality. These definitions are equivalent (the "terms" in the definition used here being the standard representative in the alternative definition).

Note that the F in the map rules is a variable and that $\lambda x.F \cdot x$ can be instantiated by any term $\lambda x.s : \mathtt{nat} \to \mathtt{nat}$. An example reduction:

$$\begin{array}{ll} & \mathtt{map} \cdot (\lambda x.x) \cdot (\mathtt{cons} \cdot (\mathtt{s} \cdot \mathtt{0}) \cdot \mathtt{nil}) \\ \Rightarrow_{\mathcal{R}} & (\mathtt{cons} \cdot ((\lambda x.x) \cdot (\mathtt{s} \cdot \mathtt{0})) \cdot (\mathtt{map} \cdot (\lambda x.x) \cdot \mathtt{nil}))\updownarrow^{\eta}_{\beta} \\ = & \mathtt{cons} \cdot (\mathtt{s} \cdot \mathtt{0}) \cdot (\mathtt{map} \cdot (\lambda x.x) \cdot \mathtt{nil}) \\ \Rightarrow_{\mathcal{R}} & \mathtt{cons} \cdot (\mathtt{s} \cdot \mathtt{0}) \cdot (\mathtt{nil}\updownarrow^{\eta}_{\beta}) \\ = & \mathtt{cons} \cdot (\mathtt{s} \cdot \mathtt{0}) \cdot \mathtt{nil} \end{array}$$

3.3.2 From PRS to AFSM

The definition of terms as a special kind of pre-terms is straightforward enough; in practice, we merely have the convenient assumption that all terms we can reduce are β-normal and η-long, and should remember to β-normalise a term after reducing it.

Transformation 3.4 *(Transforming a PRS into an AFSM)* For any simple type $\sigma := \sigma_1 \to \ldots \to \sigma_n \to \iota$ with $\iota \in \mathcal{B}$, let σ' be the type declaration $[\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \iota$. Given a PRS $(\mathcal{F}, \mathcal{R})$, let $\mathcal{F}^{\mathtt{PA}} = \{f' : \sigma' \mid f : \sigma \in \mathcal{F}\}$, and choose for all variables $x : \sigma$ which occur freely in some rule, a uniquely corresponding AFSM-meta-variable $Z^x : \sigma'$. Now, given a set S of variables, define a function φ_S which maps PRS-terms to AFSM-(meta-) terms as follows:

$$\begin{array}{lcll} \varphi_S(\lambda x.s) & = & \lambda x.\varphi_S(s) & \\ \varphi_S(f \cdot s_1 \cdots s_n) & = & f'(\varphi_S(s_1), \ldots, \varphi_S(s_n)) & f \in \mathcal{F} \\ \varphi_S(x \cdot s_1 \cdots s_n) & = & x \cdot \varphi_S(s_1) \cdots \varphi_S(s_n)) & x \in \mathcal{V} \setminus S \\ \varphi_S(x \cdot y_1\!\uparrow^{\eta} \cdots y_n\!\uparrow^{\eta}) & = & Z^x(y_1, \ldots, y_n) & x \in S,\ y_1, \ldots, y_n \in \mathcal{V} \setminus S \\ \varphi_S(x \cdot s_1 \cdots s_n) & = & Z^x(\psi_S(s_1), \ldots, \psi_S(s_n)) & x \in S, \text{ otherwise} \end{array}$$

We shortly denote $\varphi(s) := \varphi_{\emptyset}(s)$.

Let $\mathcal{R}^{\mathtt{PA}} = \{\varphi_{FV(l)}(l) \Rightarrow \varphi_{FV(l)}(r) \mid l \Rightarrow r \in \mathcal{R}\}$; this is a well-defined set of AFSM-rules because a PRS-pattern is always mapped to an AFSM-pattern, as a simple induction on the definition of φ_S shows.

Thus, the PRS-rule

$$\mathtt{map} \cdot (\lambda x.F \cdot x) \cdot (\mathtt{cons} \cdot y \cdot z) \Rightarrow \mathtt{cons} \cdot (F \cdot y) \cdot (\mathtt{map} \cdot (\lambda x.F \cdot x) \cdot z)$$

is mapped to an AFSM-rule

$$\mathtt{map}'(\lambda x.F(x), \mathtt{cons}'(Y, Z)) \Rightarrow \mathtt{cons}'(F(Y), \mathtt{map}'(\lambda x.F(x), Z))$$

The role of reasoning modulo β is taken over by the use of meta-variables. At least, to some extent; there are still AFSM-terms over $\mathcal{F}^{\mathtt{PA}}$ which only correspond with *pre-terms* in the original PRS (for instance a term $(\lambda x.x) \cdot \mathtt{0}$). Consequently,

Transformation 3.4 does not necessarily preserve termination. Consider for example the following third-order PRS:

$$\begin{array}{rcll} \mathtt{h} \cdot \mathtt{0} & \Rightarrow & \mathtt{g} \cdot (\lambda xy.x \cdot (\mathtt{h} \cdot y)) & (x : \mathtt{nat} \to \mathtt{nat}) \\ \mathtt{g} \cdot (\lambda xy.Z \cdot (\lambda z.(x \cdot z)) \cdot y) & \Rightarrow & Z \cdot (\lambda z.\mathtt{0}) \cdot \mathtt{0} & \end{array}$$

This PRS is terminating: $\mathtt{h} \cdot \mathtt{0}$ reduces to $\mathtt{g} \cdot (\lambda xy.x \cdot (\mathtt{h} \cdot y))$, which reduces to $(\lambda xy.x \cdot (\mathtt{h} \cdot y)) \cdot (\lambda z.\mathtt{0}) \cdot \mathtt{0} \updownarrow^{\eta}_{\beta} = \mathtt{0}$. This PRS is transformed into the following AFSM:

$$\begin{array}{rcl} \mathtt{h}'(\mathtt{0}') & \Rightarrow & \mathtt{g}'(\lambda xy.x \cdot \mathtt{h}'(y)) \\ \mathtt{g}'(\lambda xy.Z(x,y)) & \Rightarrow & Z(\lambda z.\mathtt{0}', \mathtt{0}'), \end{array}$$

We have an infinite loop: $\mathtt{h}'(\mathtt{0}') \Rightarrow \mathtt{g}'(\lambda xy.x \cdot \mathtt{h}'(y)) \Rightarrow (\lambda z.\mathtt{0}') \cdot \mathtt{h}'(\mathtt{0}') \Rightarrow \ldots$

This is the case because the application symbol $\cdot$ in a PRS has a slightly different nature than the same symbol in an AFSM: every appearance of a β-redex is instantly reduced. Even in a second-order system the difference may be significant. Consider for example the following AFSM, obtained as a translation from a PRS[3]:

$$\begin{array}{rcl} \mathtt{f}_1(X) & \Rightarrow & \mathtt{f}_2(X, X) \\ \mathtt{f}_2(\mathtt{a}, X) & \Rightarrow & \mathtt{f}_3(X) \\ \mathtt{g}_1(\lambda x.\mathtt{f}_3(F(x))) & \Rightarrow & F(\mathtt{hide}(\lambda x.\mathtt{f}_3(F(x)))) \\ \mathtt{unhide}(\mathtt{hide}(\lambda x.F(x))) & \Rightarrow & \mathtt{g}_2(\lambda x.F(x)) \\ \mathtt{g}_2(\lambda x.\mathtt{f}_3(F(x))) & \Rightarrow & \mathtt{g}_1(\lambda x.\mathtt{f}_1(F(x))) \end{array}$$

This (admittedly highly artificial) second-order system has the property that any term that is β-normal can only be reduced to other β-normal terms. Yet non-termination is caused by the possibility of non-β-normal terms. Let $\chi[y] := (\lambda x.\mathtt{a}) \cdot \mathtt{unhide}(y)$. Then:

$$\begin{array}{rl} & \mathtt{g}_1(\lambda y.\mathtt{f}_1(\chi[y])) \\ \Rightarrow_{\mathcal{R}} & \mathtt{g}_1(\lambda y.\mathtt{f}_2(\chi[y], \chi[y])) \\ \Rightarrow_{\beta} & \mathtt{g}_1(\lambda y.\mathtt{f}_2(\mathtt{a}, \chi[y])) \\ \Rightarrow_{\mathcal{R}} & \mathtt{g}_1(\lambda y.\mathtt{f}_3(\chi[y])) \\ \Rightarrow_{\mathcal{R}} & \chi[\mathtt{hide}(\lambda y.\mathtt{f}_3(\chi[y]))] \\ = & (\lambda x.\mathtt{a}) \cdot \mathtt{unhide}(\mathtt{hide}(\lambda y.\mathtt{f}_3(\chi[y]))) \\ \Rightarrow_{\mathcal{R}} & (\lambda x.\mathtt{a}) \cdot \mathtt{g}_2(\lambda y.\mathtt{f}_3(\chi[y])) \\ \Rightarrow_{\mathcal{R}} & (\lambda x.\mathtt{a}) \cdot \underline{\mathtt{g}_1(\lambda y.\mathtt{f}_1(\chi[y]))} \end{array}$$

The crux of the example is that, because in an AFSM we do not work with β-normal terms, there is a term that reduces to $\mathtt{a}$ but also has a subterm $\mathtt{unhide}(y)$. In the original PRS, no counterpart of this reduction exists.

Thus we see: Transformation 3.4 does not preserve termination. However, non-termination is preserved, and termination is preserved if we use a beta-first reduction strategy.

[3]The PRS in question is omitted, because it is utterly non-interesting: it looks exactly the same, except that it is curried. The rules are already in η-long form.

Theorem 3.5. *The PRS $(\mathcal{F}, \mathcal{R})$ is terminating if and only if the AFSM $(\mathcal{F}^{\mathtt{PA}}, \mathcal{R}^{\mathtt{PA}})$ is terminating using a reduction strategy where β-reduction is preferred to other steps.*

Proof. Let l, r, s be PRS-terms, S a set of variables, and γ a substitution on domain S. Let δ_γ be $[Z^x := \varphi(\gamma(x)) \mid x \in S]$. We can make the following observations:

1. If $l = x \cdot y_1 \uparrow^\eta \cdots y_n \uparrow^\eta$ with $x \in S$ and all y_i distinct variables not in S, then $\varphi(l\gamma\updownarrow^\eta_\beta) = \varphi_S(l)\delta_\gamma$.

   Proof: Using α-conversion, we can write $\gamma(x) = \lambda y_1 \dots y_n.t$, so $l\gamma \updownarrow^\eta_\beta = t[y_1 := y_1 \uparrow^\eta, \dots, y_n := y_n \uparrow^\eta] \updownarrow^\eta_\beta = t \updownarrow^\eta_\beta = t$. Therefore $\varphi(l\gamma \updownarrow^\eta_\beta) = \varphi(t)$. On the other hand $\delta_\gamma(Z^x) = \varphi(\gamma(x)) = \lambda y_1 \dots y_n.\varphi(t)$, so $\varphi_S(l)\delta_\gamma = Z^x(y_1, \dots, y_n)\delta_\gamma = \varphi(t)[y_1 := \varphi(y_1)\delta_\gamma, \dots, y_n := \varphi(y_n)\delta_\gamma] = \varphi(t)[y_1 := y_1, \dots, y_n := y_n] = \varphi(t)$ as required.

2. If for all subterms $x \cdot s_1 \cdots s_n$ of l with $x \in S$ all s_i are (the η-long forms of) distinct variables not in S, then $\varphi(l\gamma\updownarrow^\eta_\beta) = \varphi_S(l)\delta_\gamma$.

   Proof: This follows by a straightforward induction on the size of l, the base case being (1) and the induction cases ($l = \lambda x.l_0$, $l = f \cdot l_1 \cdots l_n$ with $f \in \mathcal{F}$ or $l = x \cdot l_1 \cdots l_n$ with x not occurring in S) all simple.

3. $\varphi(s)[x := \varphi(\gamma(x)) \mid x \in S] \Rightarrow^*_\beta \varphi(s\gamma\updownarrow^\eta_\beta)$.

   Proof: By induction on the pre-term $s\gamma$, ordered with $\Rightarrow_\beta \cup \rhd$ (which is terminating because $\Rightarrow_\beta$ is). The cases where $s = \lambda x.s_0$, $s = f \cdot s_1 \cdots s_n$ or $s = x \cdot s_1 \cdots s_n$ with $x \in \mathcal{V} \setminus S$ are all immediate with the $\rhd$ part of the induction hypothesis. What remains is when $s = x \cdot s_1 \cdots s_n$ with $x \in S$.

   Let $\gamma(x) = \lambda y_1 \dots y_n.t$. Write $\delta := [x := \varphi(\gamma(x)) \mid x \in S]$. Then $\varphi(s)\delta = \delta(x) \cdot \varphi(s_1)\delta \cdots \varphi(s_n)\delta \Rightarrow^*_\beta \delta(x) \cdot \varphi(s_1\gamma\updownarrow^\eta_\beta) \cdots \varphi(s_n\gamma\updownarrow^\eta_\beta)$ by the $\rhd$ part of the induction hypothesis. Since $\delta(x) = \lambda y_1 \dots y_n.\varphi(t)$, this term β-reduces to $\varphi(t)[y_1 := \varphi(s_1\gamma\updownarrow^\eta_\beta), \dots, y_n := \varphi(s_n\gamma\updownarrow^\eta_\beta)]$. Using the $\Rightarrow_\beta$ part of the induction hypothesis, this term β-reduces to $\varphi(t[y_1 := s_1\gamma\updownarrow^\eta_\beta, \dots, y_n := s_n\gamma\updownarrow^\eta_\beta]\updownarrow^\eta_\beta) = \varphi(t[y_1 := s_1\gamma, \dots, y_n := s_n\gamma]\updownarrow^\eta_\beta) = \varphi(s\gamma\updownarrow^\eta_\beta)$.

4. $\varphi_S(r)\delta_\gamma \Rightarrow^*_\beta \varphi(r\gamma\updownarrow^\eta_\beta)$.

   Proof: By induction on the form of r. The cases where r is one of $\lambda x.r_0$ or $f \cdot r_1 \cdots r_n$ or $x \cdot r_1 \cdots r_n$ with $x \notin S$ are immediate with the induction hypothesis. So suppose $r = x \cdot r_1 \cdots r_n$ with $x \in S$. If the r_i are all (the η-long forms of) different variables not in S then (1) gives the required (in-)equality. Otherwise let $\gamma(x) = \lambda y_1 \dots y_n.t$. Then $\varphi_S(r)\delta_\gamma = Z^x(\varphi_S(r_1), \dots, \varphi_S(r_n))\delta_\gamma = \varphi(t)[y_1 := \varphi_S(r_1)\delta_\gamma, \dots, y_n := \varphi_S(r_n)\delta_\gamma]$, and by the induction hypothesis on the r_i this term β-reduces to $\varphi(t)[y_1 := \varphi(r_1\gamma\updownarrow^\eta_\beta), \dots, y_n := \varphi(r_n\gamma\updownarrow^\eta_\beta)]$. By (3) this term $\Rightarrow^*_\beta \varphi(t[y_1 := r_1\gamma\updownarrow^\eta_\beta, \dots, y_n := r_n\gamma\updownarrow^\eta_\beta]\updownarrow^\eta_\beta) = \varphi(t[y_1 := r_1\gamma, \dots, y_n := r_n\gamma]\updownarrow^\eta_\beta) = \varphi(r\gamma\updownarrow^\eta_\beta)$ as required.

5. $\varphi(C[s]) = \varphi(C)[\varphi(s)]$ if s has base type and C is a context.

   Proof: By a completely straightforward induction on the form of C (note that no meta-variables are introduced because we consider $\varphi_\emptyset$).

6. If $s \Rightarrow_\mathcal{R} t$, then $\varphi(s) \Rightarrow_\mathcal{R} \cdot \Rightarrow_\beta^* t$.

   Proof: If $s = C[l\gamma\updownarrow_\beta^\eta]$ and $t = C[r\gamma\updownarrow_\beta^\eta]$, then $\varphi(s) = \varphi(C)[\varphi(l\gamma\updownarrow_\beta^\eta)]$ by (5), $= \varphi(C)[\varphi_{FV(l)}(l)\delta_\gamma]$ by (2), $\Rightarrow_{\mathcal{R}^{\mathtt{PA}}} \varphi(C)[\varphi_{FV(l)}(r)\delta_\gamma] \Rightarrow_\beta^* \varphi(C)[\varphi(r\delta\updownarrow_\beta^\eta)]$ by (4), $= \varphi(C[r\delta\updownarrow_\beta^\eta])$ by (5).

Fact (6) provides the *only if* part of the Theorem: if there is an infinite reduction in the PRS $(\mathcal{F}, \mathcal{R})$, we can also find an infinite reduction in $(\mathcal{F}^{\mathtt{PA}}, \mathcal{R}^{\mathtt{PA}})$.

For the *if* part, consider an "inverse" φ^{-1} of φ, defined only on AFSM-terms in η-long β-normal form:

$$\begin{array}{rcl} \varphi^{-1}(\lambda x.s) & = & \lambda x.\varphi^{-1}(s) \\ \varphi^{-1}(x \cdot s_1 \cdots s_n) & = & x \cdot \varphi^{-1}(s_1) \cdots \varphi^{-1}(s_n) \\ \varphi^{-1}(f'(s_1, \ldots, s_n)) & = & f \cdot \varphi^{-1}(s_1) \cdots \varphi^{-1}(s_n) \end{array}$$

It is evident that $\varphi^{-1}(\varphi(s)) = s$ for all PRS-terms s, and also that $\varphi(\varphi^{-1}(t)) = t$ for AFSM-terms t in η-long β-normal form. We also have:

7. $\varphi^{-1}(C[s]) = \varphi^{-1}(C)[\varphi^{-1}(s)]$ if C is an AFSM-context and s a base-type AFSM-term, both in η-long β-normal form.

   Proof: $\varphi^{-1}(C[s]) = \varphi^{-1}(\varphi(\varphi^{-1}(C))[\varphi(\varphi^{-1}(s))])$, which by (5) equals $\varphi^{-1}(\varphi(\varphi^{-1}(C)[\varphi^{-1}(s)])) = \varphi^{-1}(C)[\varphi^{-1}(s)])$.

Now suppose there is an infinite beta-first reduction over $(\mathcal{F}^{\mathtt{PA}}, \mathcal{R}^{\mathtt{PA}})$, say $s_i \Rightarrow_\beta^* s_i\downarrow_\beta \Rightarrow_{\mathcal{R}^{\mathtt{PA}}} s_{i+1}$ for all $i \in \mathbb{N}$. Recalling Lemma 2.15(7) we can assume that these s_i all have η-long form. Since the reduction $s_i\downarrow_\beta \Rightarrow_{\mathcal{R}^{\mathtt{PA}}} s_{i+1}$ cannot be a β-step, we may write $s_i\downarrow_\beta = C[\varphi_{FV(l)}(l)\chi]$ and $s_{i+1} = C[\varphi_{FV(l)}(r)\chi]$ for some PRS-rule $l \Rightarrow r$ and substitution χ on domain $\{Z^x \mid x \in FV(l)\}$. Choosing $\gamma := [x := \varphi^{-1}(\chi(Z^x)) \mid x \in FV(l)]$ we have $\chi = \delta_\gamma$, so by (7) $\varphi^{-1}(s_i) = \varphi^{-1}(C)[\varphi^{-1}(\varphi_{FV(l)}(l)\delta_\gamma)]$, which by (2) equals $\varphi^{-1}(C)[l\gamma\updownarrow_\beta^\eta]$. This term reduces to $\varphi^{-1}(C)[r\gamma\updownarrow_\beta^\eta]$, which is in β-normal form, and therefore by (4) is equal to $\varphi^{-1}(C)[\varphi^{-1}(\varphi_{FV(l)}(r)\delta_\gamma\downarrow_\beta)] = \varphi^{-1}(C[\varphi_{FV(l)}(r)\chi]\downarrow_\beta) = \varphi^{-1}(s_{i+1}\downarrow_\beta)$.

We obtain an infinite PRS-reduction $\varphi^{-1}(s_1\downarrow_\beta) \Rightarrow_\mathcal{R} \varphi^{-1}(s_2\downarrow_\beta) \Rightarrow_\mathcal{R} \ldots$. □

Thus, a PRS is terminating if the corresponding AFSM is terminating. Theorem 3.5 gives us more: we don't have to prove full termination, but merely termination under a particular reduction strategy. In this work, however, this property will not be exploited, as the focus is on full termination.

3.3.3 From AFSM To PRS

Alternatively, to use the existing termination techniques for PRSs to derive termination of an AFSM, we will need an embedding of AFSMs into PRSs. The trick is simply to consider application as an explicit function symbol, and β-reduction as an infinite set of rules.

Transformation 3.6 *(Transforming an AFSM into a PRS)* Recall the definition $\mathtt{cur}([\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau) = \sigma_1 \to \ldots \to \sigma_n \to \tau$ of "currying" type declarations used in Transformation 2.6.

Given an AFSM $(\mathcal{F}, \mathcal{R})$, let $\mathcal{F}^{\mathtt{AP}} = \{f' : \mathtt{cur}(\sigma) \mid f : \sigma \in \mathcal{F}\} \cup \{@^\sigma : \sigma \to \sigma \mid \sigma \in \mathcal{T} \mid \sigma \text{ functional}\}$. Choose, for all meta-variables $Z : \sigma$, a matching variable x_Z with type $\mathtt{cur}(\sigma)$. Now let $\varphi(s)$ be inductively defined as follows:

$$\begin{array}{lcll} \varphi(x) & = & x{\uparrow^\eta} & (x \in \mathcal{V}) \\ \varphi(f(s_1 \cdots s_n)) & = & f' \cdot \varphi(s_1) \cdots \varphi(s_n){\uparrow^\eta} & (f \in \mathcal{F}) \\ \varphi(Z(s_1, \ldots, s_n)) & = & x_Z \cdot \varphi(s_1) \cdots \varphi(s_n){\uparrow^\eta} & (Z \in \mathcal{M}) \\ \varphi(s \cdot t) & = & @^\sigma \cdot \varphi(s) \cdot \varphi(t){\uparrow^\eta} & (s : \sigma) \\ \varphi(\lambda x.s) & = & \lambda x.\varphi(s) & \end{array}$$

$$\begin{array}{rl} \text{Let } R := & \{\varphi(l) \Rightarrow \varphi(r) \mid l \Rightarrow r \in \mathcal{R}\} \\ & \cup \{(@^\sigma \cdot x \cdot y){\uparrow^\eta} \Rightarrow (x \cdot y){\uparrow^\eta} \mid \text{all functional types } \sigma\} \\ \text{and } \mathcal{R}^{\mathtt{AP}} := & \{l \Rightarrow r \in R \mid \lambda\vec{x}.l \Rightarrow \lambda\vec{x}.r \in R \mid l, r \text{ having base type}\}. \end{array}$$

It is easy to see that φ maps every meta-term in the AFSM $(\mathcal{F}, \mathcal{R})$ to a term in the PRS $(\mathcal{F}^{\mathtt{AP}}, \mathcal{R}^{\mathtt{AP}})$. Some further study reveals that every AFSM-rule is mapped to a PRS-rule: $\varphi(l)$ is a PRS-pattern if l is an AFSM-pattern. Moreover, reductions, and therefore non-termination, are preserved:

Theorem 3.7. *The AFSM $(\mathcal{F}, \mathcal{R})$ is terminating if the PRS $(\mathcal{F}^{\mathtt{AP}}, \mathcal{R}^{\mathtt{AP}})$ is.*

Proof. We will see that $s \Rightarrow_{\mathcal{R}} t$ implies that $\varphi(s) \Rightarrow_{\mathcal{R}^{\mathtt{AP}}} \varphi(t)$. We observe:

(**) $\varphi(s\gamma) =_{\beta/\eta} \varphi(s)\gamma^\varphi$ for all s, γ if $dom(\gamma)$ contains all meta-variables in s.

This holds by a straightforward induction, first on the number of meta-variables occurring in s, second on its size. The cases where s is an abstraction, functional term or application, are immediate with the second induction hypothesis. The case for s a variable is also clear, whether $s \in dom(\gamma)$ (both sides are η-equal to $\varphi(\gamma(s))$), or $s \notin dom(\gamma)$ (both sides are $s{\uparrow^\eta}$). The only remaining case, $s = Z(s_1, \ldots, s_n)$ with $\gamma(Z) = \lambda x_1 \ldots x_n.t$, uses the first induction hypothesis: $\varphi(s\gamma) = \varphi(t[\vec{x} := \vec{s}\gamma]) =_{\beta/\eta}$ (IH1) $\varphi(t)[\vec{x} := \varphi(\vec{s}\gamma)] =_{\beta/\eta}$ (IH2) $\varphi(t)[\vec{x} := \varphi(\vec{s})\gamma^\varphi] =_\beta \varphi(\gamma^\varphi(x_Z)) \cdot (\varphi(s_1)\gamma^\varphi) \cdots (\varphi(s_n)\gamma^\varphi) = \varphi(s)\gamma^\varphi$.

Using (**), we can see that if $s \Rightarrow_{\mathcal{R}} t$ in the original AFSM, then $\varphi(s) \Rightarrow_{\mathcal{R}} \varphi(t)$ in the resulting PRS; by induction on the size of s. The induction cases (where s is a functional term, application or abstraction, and the reduction happens in an

immediate subterm) are all straightforward (taking into account that every $q\uparrow^\eta$ in the definition of φ can be read as $\lambda\vec{x}.q\cdot(x_1\uparrow^\eta)\cdots(x_n\uparrow^\eta)$). Only the base case of a topmost step remains.

First suppose $s = l\gamma \Rightarrow_{\mathcal{R}} r\gamma = t$. Noting that φ maps to terms, which are already in long β/η-normal form, (**) provides that $\varphi(s) = \varphi(l)\gamma^\varphi \updownarrow^\eta_\beta \Rightarrow_{\mathcal{R}^{\text{AP}}} \varphi(r)\gamma^\varphi\updownarrow^\eta_\beta = \varphi(t)$ as required.

Alternatively, if $s = (\lambda x.q)\cdot u$ and $t = q[x := u]$, then $\varphi(s) = @^\sigma\cdot(\lambda x.\varphi(q))\cdot\varphi(u) \Rightarrow_{\mathcal{R}^{\text{AP}}} (\lambda x.\varphi(q))\cdot\varphi(u)$, and $(\lambda x.\varphi(q))\cdot\varphi(u)\updownarrow^\eta_\beta = \varphi(q)[x := \varphi(u)]\updownarrow^\eta_\beta$, which by (**) equals $\varphi(t)\updownarrow^\eta_\beta = \varphi(t)$. □

Theorem 3.7 is not an equivalence. Consider for example the terminating AFSM with $\mathtt{f} : [\mathtt{o}\to\mathtt{o}]\longrightarrow\mathtt{o}$ and $\mathtt{g} : \mathtt{o}\to\mathtt{o}$, and a single rule

$$\mathtt{f}(\mathtt{g}) \Rightarrow \mathtt{f}(\lambda y.(\mathtt{g}\cdot y))$$

The translation to a PRS,

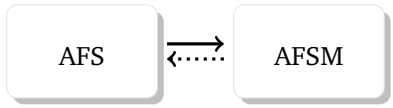
$$\mathtt{f}'\cdot(\lambda y.(\mathtt{g}\cdot y)) \Rightarrow \mathtt{f}'\cdot(\lambda y.(@^{\mathtt{o}\to\mathtt{o}}\cdot(\lambda z.(\mathtt{g}\cdot z))\cdot y))$$

is not terminating, as demonstrated by the looping reduction:

$$\begin{array}{ll} & \mathtt{f}'\cdot(\lambda y.(\mathtt{g}\cdot y)) \\ \Rightarrow & \mathtt{f}'\cdot(\lambda y.@^{\mathtt{o}\to\mathtt{o}}(\lambda z.(\mathtt{g}\cdot z), y)) \\ \Rightarrow & \mathtt{f}'\cdot(\lambda y.((\lambda z.(\mathtt{g}\cdot z))\cdot y)\updownarrow^\eta_\beta) \\ = & \mathtt{f}'\cdot(\lambda y.(\mathtt{g}\cdot y)) \end{array}$$

In addition, Theorem 3.7 creates an infinite PRS, while the original AFSM may be finite. However, this is not likely to be much of a problem: these infinitely many rules all have the same form, and definitions of e.g. the higher-order recursive path ordering and weakly monotonic algebras can typically orient these rules easily (we shall study variations of these techniques for the class of AFSMs in the following chapters).

3.4 Algebraic Functional Systems



An *Algebraic Functional System*, as defined in [63] (based on a definition in [60]), extends the simply-typed λ-calculus with function symbols and rewrite rules. Consequently, β is a separate reduction step. As suggested by the name, AFSs and AFSMs are very similar, but AFSs use normal variables instead of meta-variables in the rules (which gives more limited matching functionality). There are several variations of AFSs in the literature: using different type systems, with or without an η-reduction rule, sometimes adding pairing rules... Here, we will stick with the definitions of [63], limited to simple types. This variation of the formalism also appears in [118, Chapter 11], and is used in the higher-order category of the annual termination competition [125].

3.4.1 Definition

Terms in Algebraic Functional Systems are exactly AFSM-terms, defined using clauses (`var`), (`fun`), (`abs`) and (`app`) in Definition 2.2. There are no meta-terms. Substitutions and contexts are defined as before. A *rule* is a pair of terms $l \Rightarrow r$ such that l and r have the same type and all variables in r also occur in l. The rewrite relation $\Rightarrow_{\mathcal{R}}$ is generated by:

$$\begin{array}{rcll} C[l\gamma] & \Rightarrow_{\mathcal{R}} & C[r\gamma] & \text{if } l \Rightarrow r \in \mathcal{R},\ C \text{ a context},\ \gamma \\ & & & \text{a substitution},\ dom(\gamma) = FV(l) \\ C[(\lambda x.s) \cdot t] & \Rightarrow_{\mathcal{R}} & C[s[x := t]] & \end{array}$$

An AFS is a pair $(\mathcal{F}, \mathcal{R})$ of a signature and a set of AFS-rules over this signature.

3.4.2 From AFS to AFSM

Given an AFS with rules $\mathcal{R}$, we might replace the free variables in the rules by meta-variables, which gives an AFSM with the same reduction relation (and thus equivalent termination) as the original AFS – *if* the left-hand side of the resulting rule scheme is a pattern of the form $f(l_1, \ldots, l_m) \cdot l_{m+1} \cdots l_n$. In an AFS, there is no pattern restriction. It is perfectly allowed to match on an application (using rules like $x \cdot y \Rightarrow y$ or $f((\lambda x.y) \cdot \mathtt{0}) \Rightarrow y$), or on an abstraction (using a rule $\lambda x.l \Rightarrow r$). Nevertheless, the following result is obviously true.

Theorem 3.8. *An AFS where all the rules have the form $f(l_1, \ldots, l_m) \cdot l_{m+1} \cdots l_n$, and left-hand sides have no subterms $s \cdot t$ with s an abstraction or free variable, is terminating if and only if the corresponding AFSM is.*

In Section 3.4.4 we will see that an AFS can always be transformed to satisfy these restrictions, without losing either termination or non-termination. This closely follows the reasoning in [75]. Consequently, for every AFS we can find an AFSM which is terminating if and only if the original AFS is.

3.4.3 From AFSM to AFS

For the other direction, we can use Transformation 2.18. An AFSM with simple meta-applications is terminating if the corresponding AFSM with flattened meta-applications is terminating, and such an AFSM corresponds exactly to an AFS.

Example 3.9. Recall the AFSM from Example 2.3:

$$\begin{array}{rcl} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y)) \end{array}$$

This AFSM has simple meta-applications, and thus is terminating if the following AFS is terminating:

$$\begin{array}{rcl} \mathtt{map}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{map}(F, \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F \cdot X, \mathtt{map}(F, Y)) \end{array}$$

For AFSMs with non-simple meta-variables, such as the one we saw in Section 2.3.3, there is no obvious transformation to an AFS:

$$\mathtt{d}(\lambda x.\mathtt{sin}(F(x))) \Rightarrow \lambda x.(\mathtt{d}(\lambda y.F(y)) \cdot x) \times \mathtt{cos}(F(x))$$

3.4.4 Simplifying AFSs

To obtain the inclusion in the diagram, we have yet to see that an AFS can always be transformed such that for all rules $l \Rightarrow r$, the left-hand side l:

- is β-normal;
- has no subterms $x \cdot s$ with x free in s;
- is not an abstraction.

With these properties, l corresponds to a pattern (if free variables are replaced by meta-variables with arity 0), and by elimination of alternatives, l must have the form $f(l_1, \ldots, l_m) \cdot l_{m+1} \cdots l_n$, or be a single variable. Since a rule $x \Rightarrow_{\mathcal{R}} r$ is obviously non-terminating, we can safely assume such rules do not occur.

Example 3.10. In the rest of this section, we will go through a transformation of an AFS with the following function symbols:

$$\begin{array}{rcl}
\mathtt{0} & : & \mathtt{nat} \\
\mathtt{s} & : & [\mathtt{nat}] \longrightarrow \mathtt{nat} \\
\mathtt{nil} & : & \mathtt{list} \\
\mathtt{cons} & : & [\mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list} \\
\mathtt{map} & : & [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{list}] \longrightarrow \mathtt{list} \\
\mathtt{op} & : & [(\mathtt{nat} \to \mathtt{nat}) \times (\mathtt{nat} \to \mathtt{nat})] \longrightarrow \mathtt{nat} \to \mathtt{nat} \\
\mathtt{pow} & : & [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat}] \longrightarrow \mathtt{nat} \to \mathtt{nat}
\end{array}$$

And rules $\mathcal{R}_{\mathtt{pow}}$ consisting of:

$$\begin{array}{rcl}
\mathtt{map}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{map}(F, \mathtt{cons}(x, y)) & \Rightarrow & \mathtt{cons}(F \cdot x, \mathtt{map}(F, y)) \\
\mathtt{pow}(F, \mathtt{0}) & \Rightarrow & \lambda x.x \\
\mathtt{pow}(F, \mathtt{s}(x)) & \Rightarrow & \mathtt{op}(F, \mathtt{pow}(F, x)) \\
\mathtt{op}(F, G) \cdot x & \Rightarrow & F \cdot (G \cdot x) \\
\lambda x.F \cdot x & \Rightarrow & F \quad (\text{where } F : \mathtt{nat} \to \mathtt{nat})
\end{array}$$

Removing Leading Free Variables. The first step is to get rid of subterms $x \cdot s$ in the left-hand side of a rule. To do so, we will first instantiate headmost variables with functional terms: for any rule $l = C[x \cdot s] \Rightarrow r$ all possible rules $l[x := f(\vec{y}) \cdot \vec{z}] \Rightarrow r[x := f(\vec{y}) \cdot \vec{z}]$ are added. Now when a rule with leading variables is used, we can assume these variables are not instantiated with a functional term.

Second, we introduce a number of fresh symbols $@^\sigma$ and replace occurrences $s \cdot t$ in any rule by $@^\sigma(s,t)$ if $s : \sigma$ and s is not a functional term, and σ corresponds to the type of a leading variable in any left-hand side. We add rules $@^\sigma(x,y) \Rightarrow x \cdot y$ only for those $@^\sigma$ occurring in the changed rules. With this transformation, the applicative `map` rule

$$\mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot x \cdot y) \Rightarrow \mathtt{cons} \cdot (F \cdot x) \cdot (\mathtt{map} \cdot F \cdot y)$$

either stays unchanged if there are no rules with a leading variable of type $\mathtt{nat} \to \mathtt{nat}$ in the left-hand side, or, if there are, becomes:

$$\mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot x \cdot y) \Rightarrow \mathtt{cons} \cdot @^{\mathtt{nat}\to\mathtt{nat}}(F,x) \cdot (\mathtt{map} \cdot F \cdot y)$$

We will make this transformation formal, and more general, in Transformations 3.12–3.18. To start, we choose for every function symbol f an *output arity*, denoted $oa(f)$. Intuitively, this is a number k such that all applications of the form $f(s_1,\ldots,s_m) \cdot s_{m+1} \cdots s_n$ with $n \leq k$ are "protected": no $@^\sigma$ symbols will be introduced at the head of such a term. There are various reasonable choices for $oa(f)$:

I if $f : [\sigma_1 \times \ldots \times \sigma_m] \longrightarrow \tau_1 \to \ldots \tau_n \to \iota$, an obvious choice is n (so *all* terms headed by a functional term are protected – this is the choice that was assumed in the idea sketch above);

II another reasonable choice, which avoids unnecessary introduction of new rules, is to take for $oa(f)$ the highest number k such that $f(s_1,\ldots,s_m) \cdot t_1 \cdots t_k$ appears in any rule;

III alternatively, we might choose $oa(f) = 0$ for all f; in that case, Transformations 3.12 and 3.15 have no effect, but we may end up introducing more symbols $@^\sigma$ than necessary.

Example 3.11. Let us follow guideline II, so $oa(f)$ is the highest number m such that $f(\vec{s}) \cdot t_1 \cdots t_m$ occurs in a rule. In the system $\mathcal{R}_{\mathtt{pow}}$ from Example 3.10 this guideline gives output arity 0 for all symbols except `op`, to which we assign output arity 1.

Let a term be *limited functional* if it has the form $f(\vec{s}) \cdot t_1 \cdots t_n$ and $n < oa(f)$. Note that if $oa(f) = 0$, then a term $f(\vec{s})$ is *not* limited functional: we could think of a limited functional term as a term which expects additional arguments.

Following the idea sketch, we will first instantiate leading variables. Let $HV(s)$ be the set of free head variables of s, that is, the set of those $x \in FV(s)$ where x occurs at the head of an application in s ($s = C[x \cdot t]$ for some context C and term t). For every rule $l = C[x \cdot t] \Rightarrow r$ we will add a number of rules where a limited functional term $f(\vec{y}) \cdot \vec{z}$ is substituted for x.

Transformation 3.12 *(Filling in head variables)* For every rule $l \Rightarrow r$ in $\mathcal{R}$, every $x : \sigma \in HV(l)$, every function symbol $f : \tau \in \mathcal{F}$ such that $\tau = [\tau_1 \times \ldots \times \tau_k] \longrightarrow \rho_1 \to \ldots \rho_n \to \sigma$ for some n with $0 \leq n < oa(f)$, let $\delta := [x := f(y_1, \ldots, y_k) \cdot z_1 \cdots z_n]$ and add a new rule $l\delta \Rightarrow r\delta$ (with $y_1, \ldots, y_k, z_1, \ldots, z_n$ fresh variables). Repeat this for the newly added rule schemes.

If $\mathcal{R}$ is finite, this process terminates (because in every step the maximum number of leading variables in a rule is lowered) and the result, $\mathcal{R}^{fill}$, is also finite. Otherwise define $\mathcal{R}^{fill}$ as the limit of the procedure.

Example 3.13. Following on Examples 3.10 and 3.11, the only rule with a leading variable in the left-hand side is the η-reduction rule $\lambda x.F \cdot x \Rightarrow F$. Consequently, Transformation 3.12 completes after one step, with a single new rule; $\mathcal{R}^{fill}$ contains:

$$\begin{array}{rcl}
\mathtt{map}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{map}(F, \mathtt{cons}(x, y)) & \Rightarrow & \mathtt{cons}(F \cdot x, \mathtt{map}(F, y)) \\
\mathtt{pow}(F, \mathtt{0}) & \Rightarrow & \lambda x.x \\
\mathtt{pow}(F, \mathtt{s}(x)) & \Rightarrow & \mathtt{op}(F, \mathtt{pow}(F, x)) \\
\mathtt{op}(F, G) \cdot x & \Rightarrow & F \cdot (G \cdot x) \\
\lambda x.F \cdot x & \Rightarrow & F \\
\lambda x.\mathtt{op}(F, G) \cdot x & \Rightarrow & \mathtt{op}(F, G)
\end{array}$$

It is not hard to see that $\mathcal{R}$ and $\mathcal{R}^{fill}$ generate the same relation (since $\mathcal{R} \subseteq \mathcal{R}^{fill}$, and all rules in $\mathcal{R}^{fill}$ are instances of a rule in $\mathcal{R}$). Moreover:

Lemma 3.14. *If $s \Rightarrow_{\mathcal{R}^{fill}} t$ with a topmost step, then there are $l \Rightarrow r \in \mathcal{R}^{fill}$ and a substitution γ such that $s = l\gamma$, $t = r\gamma$ and $\gamma(x)$ is not limited functional for any $x \in HV(l)$.*

Proof. By definition of topmost step, there exist l, r, γ such that $s = l\gamma$ and $t = r\gamma$. We use induction on the size of $A := \{x \mid x \in HV(l) \mid \gamma(x) \text{ is limited functional}\}$. If $A = \emptyset$ we are done, so assume there is some $x : \sigma \in A$. Then $\gamma(x)$ has the form $f(\vec{q}) \cdot u_1 \cdots u_n$ with $f : \tau \in \mathcal{F}$ and $n < oa(f)$. Since substitutions respect types, we can write $\tau = [\tau_1 \times \ldots \times \tau_k] \longrightarrow \rho_1 \to \ldots \to \rho_n \to \sigma$. Let $\delta := [x := f(y_1, \ldots, y_k) \cdot z_1 \cdots z_n]$. By the procedure of Transformation 3.12, $\mathcal{R}^{fill}$ contains a rule $l' := l\delta \Rightarrow r\delta =: r'$.

Let γ' be the substitution $[a := \gamma(a) \mid a \in dom(\gamma) \setminus \{x\}] \cup [y_1 := q_1, \ldots, y_k := q_k, z_1 := u_1, \ldots, z_n := u_n]$. Then it is clear that $l'\gamma' = s$ and $r'\gamma' = t$. Also $HV(l') \subset HV(l)$, and the new rule instantiates one less head variable with a limited functional term. We complete with the induction hypothesis. □

Before we move on to the introduction of symbols $@^\sigma$, it turns out to be a good idea to add a number of rules.

Transformation 3.15 *(Respecting Output Arity)* Let $\mathcal{R}^{res}$ be the set which contains all rules in $\mathcal{R}^{fill}$, and moreover for every rule $l \Rightarrow r \in \mathcal{R}^{fill}$ with l limited functional, all rules $l \cdot x_1 \Rightarrow r \cdot x_1, \ldots, l \cdot x_1 \cdots x_{k+1} \Rightarrow r \cdot x_1 \cdots x_{k+1}$ where $x_1, \ldots, x_{k+1}$ are fresh variables of a suitable type, and $l \cdot x_1 \cdots x_k$ is limited functional.

Thus, for all rules where the left-hand side is limited functional, we add variations of a smaller type. This is done because in a rule $l \Rightarrow r$ with $l = f(\vec{s}) \cdot \vec{t}$ limited functional, an application of the form $f(\vec{s}) \cdot \vec{t} \cdot q$ will be "protected" while $r \cdot q$ may not be. Note that if $\mathcal{R}^{fill}$ is finite, and $oa(f)$ is bounded (something which we can easily enforce), then also $\mathcal{R}^{res}$ is finite. It is clear that $\mathcal{R}^{fill}$ and $\mathcal{R}^{res}$ define the same rewrite relation.

Example 3.16. Since none of the left-hand sides of $\mathcal{R}_{\mathtt{pow}}$ are limited functional with the chosen output arity, Transformation 3.15 has no effect. If we had chosen option I, then $\mathtt{pow}(s,t)$ would be limited functional, so we would add two rules:

$$\begin{array}{lcl} \mathtt{pow}(F, \mathtt{0}) \cdot y & \Rightarrow & (\lambda x.x) \cdot y \\ \mathtt{pow}(F, \mathtt{s}(x)) \cdot y & \Rightarrow & \mathtt{op}(F, \mathtt{pow}(F, x)) \cdot y \end{array}$$

$\mathcal{R}^{res}$ has a nice property that we will need later, in the proof of Theorem 3.19.

Lemma 3.17. *If $s \cdot q \Rightarrow_{\mathcal{R}^{res}} t \cdot q$, and s is limited functional while t is not, then we can assume this reduction uses a topmost step: $s \cdot q = l\gamma,\ t \cdot q = r\gamma$, and $\gamma(x)$ is not limited functional for any $x \in HV(l)$.*

Proof. Since $\mathcal{R}^{fill}$ and $\mathcal{R}^{res}$ define the same rewrite relation, we know that $s \cdot q \Rightarrow_{\mathcal{R}^{fill}} t \cdot q$. If this reduction uses a topmost step, then we are done by Lemma 3.14, since $\mathcal{R}^{fill} \subseteq \mathcal{R}^{res}$. If not, the reduction must still use a headmost step, otherwise t would also be limited functional. Write $s = (l\gamma) \cdot u_1 \cdots u_k$ and $t = (r\gamma) \cdot \vec{u}$; by Lemma 3.14 we can assume that $\gamma(x)$ is not limited functional for any $x \in HV(l)$. Since also l is not a variable itself, it follows that l is not headed by a variable, so l must be limited functional too. Thus, we can write $l = f(l_1, \ldots, l_n) \cdot v_1 \cdots v_m$, and $s = f(l_1\gamma, \ldots, l_n\gamma) \cdot (v_1\gamma) \cdots (v_m\gamma) \cdot u_1 \cdots u_k$.

Let $x_1, \ldots, x_{k+1}$ be fresh variables. Since s is limited functional, we conclude that $m+k < oa(f)$, so $l \cdot x_1 \cdots x_k$ is limited functional as well. Thus $\mathcal{R}^{res}$ contains a rule $l \cdot x_1 \cdots x_{k+1} \Rightarrow r \cdot x_1 \cdots x_{k+1}$. Writing $\delta := \gamma \cup [x_1 := u_1, \ldots, x_k := u_k, x_{k+1} := q]$, we have $s \cdot q = l\delta \Rightarrow_{\mathcal{R}^{res}} r\delta = t \cdot q$, a topmost reduction. □

Finally, we have all the preparations for the main transformation. As suggested in the sketch, we should introduce new symbols $@^\sigma$ only for those σ where it is necessary. Formally, let S be a set of functional types which contains all types σ where $x : \sigma \in HV(l)$ for some variable x and $l \Rightarrow r \in \mathcal{R}^{res}$. For every type

$\sigma \to \tau \in S$, introduce a new symbol $@^{\sigma\to\tau} : [(\sigma \to \tau) \times \sigma] \longrightarrow \tau$. For all terms s define $\mathtt{exp}(s)$ as follows:

$$\begin{array}{lcl}
\mathtt{exp}(f(s_1,\ldots,s_n)) & = & f(\mathtt{exp}(s_1),\ldots,\mathtt{exp}(s_n)) \\
\mathtt{exp}(x) & = & x \ (x \text{ a variable}) \\
\mathtt{exp}(\lambda x.s) & = & \lambda x.\mathtt{exp}(s) \\
\mathtt{exp}(s\cdot t) & = & \begin{cases} @^{\sigma}(\mathtt{exp}(s),\mathtt{exp}(t)) \text{ if } s:\sigma \text{ and } \sigma \in S \text{ and} \\ \qquad s \text{ is not limited functional} \\ \mathtt{exp}(s)\cdot\mathtt{exp}(t) \text{ otherwise} \end{cases}
\end{array}$$

That is, subterms $s \cdot t$ are replaced by $@^{\sigma}(s,t)$, provided the split does not occur in a "protected" functional term, and s has a "dangerous" type.

Transformation 3.18 *(Making application explicit)* Let $\mathcal{R}^{noapp} = \{\mathtt{exp}(l) \Rightarrow \mathtt{exp}(r) \mid l \Rightarrow r \in \mathcal{R}^{res}\} \cup \{@^{\sigma}(x,y) \Rightarrow x \cdot y \mid \sigma \in S\}$.

We have reached our aim: Transformations 3.12–3.18 preserve finiteness, yet $\mathcal{R}^{noapp}$ will not have leading (free) variables. We pose the main theorem of this first step to simplify AFSs (*Removing Leading Free Variables*):

Theorem 3.19. *The rewrite relation $\Rightarrow_{\mathcal{R}^{noapp}}$ generated by $\mathcal{R}^{noapp}$ is terminating if and only if $\Rightarrow_{\mathcal{R}}$ is.*

Proof. For one direction, if $s \Rightarrow_{\mathcal{R}^{noapp}} t$ then also $s' \Rightarrow^{=}_{\mathcal{R}^{res}} t'$, where s', t' are s, t with occurrences of $@^{\sigma}(q,u)$ replaced by $q \cdot u$. Equality only occurs if s has fewer $@^{\sigma}$ symbols than t, so any infinite $\Rightarrow_{\mathcal{R}^{noapp}}$ reduction leads to an infinite $\Rightarrow_{\mathcal{R}^{res}}$ reduction. As we saw before, $\mathcal{R}^{res}$ defines the same rewrite relation as $\mathcal{R}^{fill}$ and $\mathcal{R}$, so if $\Rightarrow_{\mathcal{R}^{noapp}}$ is non-terminating, then so is $\Rightarrow_{\mathcal{R}}$.

For the other direction, suppose we can see that whenever $s \Rightarrow_{\mathcal{R}} t$ we also have $\mathtt{exp}(s) \Rightarrow^{+}_{\mathcal{R}^{noapp}} \mathtt{exp}(t)$. Then any $\Rightarrow_{\mathcal{R}}$ reduction leads to a $\Rightarrow_{\mathcal{R}^{noapp}}$ reduction of at least equal length, so if $\Rightarrow_{\mathcal{R}}$ is non-terminating, then so is $\Rightarrow_{\mathcal{R}^{noapp}}$.

It remains to be seen that $s \Rightarrow_{\mathcal{R}} t$ implies $\mathtt{exp}(s) \Rightarrow^{+}_{\mathcal{R}^{noapp}} \mathtt{exp}(t)$. We use induction on the size of s. For any term $q : \sigma$, let $\mathit{typeof}(q)$ denote the type σ.

If s is a functional term $f(s_1,\ldots,s_n)$ or an abstraction $\lambda x.s_0$, and the reduction takes place in one of the s_i, we immediately conclude with the induction hypothesis. Similarly if $s = q \cdot u \Rightarrow_{\mathcal{R}} q \cdot v$ with a reduction in the right-hand side of an application.

If the reduction takes place in the left-hand side of an application, $s = q \cdot u \Rightarrow_{\mathcal{R}} q' \cdot u = t$, we can still use the induction hypothesis if either $\mathit{typeof}(q) \notin S$, or both q and q' are limited functional, or both q and q' are not limited functional. If q is not limited functional and q' is, then $\mathtt{exp}(s) = @^{\sigma}(\mathtt{exp}(q),\mathtt{exp}(u)) \Rightarrow_{\mathcal{R}^{noapp}} \mathtt{exp}(q)\cdot\mathtt{exp}(u) \Rightarrow^{+}_{\mathcal{R}^{noapp}} \mathtt{exp}(q')\cdot\mathtt{exp}(u) = \mathtt{exp}(t)$ by the $@$-rules and the induction hypothesis.

What remains are two cases: either $s \Rightarrow_{\mathcal{R}} t$ by a topmost step, or $s = q \cdot u \Rightarrow_{\mathcal{R}} q' \cdot u = t$, where q is limited functional and q' is not. In the first case,

Lemma 3.14 provides a rule $l \Rightarrow r \in \mathcal{R}^{fill} \subseteq \mathcal{R}^{res}$ and substitution γ such that $s = l\gamma,\ t = r\gamma$ and $\gamma(x)$ is not limited functional for any $x \in HV(l)$. In the second case, Lemma 3.17 provides a rule $l \Rightarrow r \in \mathcal{R}^{res}$ and substitution γ which satisfies the same property. Using a separate induction on the definition of $\mathtt{exp}$ we find: $\mathtt{exp}(l\gamma) = \mathtt{exp}(l)\gamma^{\mathtt{exp}}$ and $\mathtt{exp}(r)\gamma^{\mathtt{exp}} \Rightarrow^*_{\mathcal{R}^{noapp}} \mathtt{exp}(r\gamma)$, where $\gamma^{\mathtt{exp}} = [x := \mathtt{exp}(\gamma(x)) \mid x \in dom(\gamma)]$. This provides $\mathtt{exp}(s) \Rightarrow^+_{\mathcal{R}^{noapp}} \mathtt{exp}(t)$ as required. The only non-trivial part of this induction is if l or r is an application:

- $l = s \cdot t$: by the assumption on γ, s is limited functional if and only if $s\gamma$ is (as we can see by a case analysis on the form of s); since also $typeof(s) \in S$ if and only if $typeof(s\gamma) \in S$, the induction hypothesis on $\mathtt{exp}(s\gamma)$ and $\mathtt{exp}(t\gamma)$ provides that $\mathtt{exp}(l\gamma)$ and $\mathtt{exp}(l)\gamma^{\mathtt{exp}}$ are the same.
- $r = s \cdot t$: of course $typeof(s) \in S$ if and only if $typeof(s\gamma) \in S$, and clearly if s is limited functional then so is $s\gamma$. Therefore either:

$$\begin{aligned}
&\mathtt{exp}(r)\gamma^{\mathtt{exp}}\\
&= (\mathtt{exp}(s)\gamma^{\mathtt{exp}}) \cdot (\mathtt{exp}(t)\gamma^{\mathtt{exp}})\\
&\Rightarrow^*_{\mathcal{R}^{noapp}} \mathtt{exp}(s\gamma^{\mathtt{exp}}) \cdot \mathtt{exp}(t\gamma^{\mathtt{exp}}) \text{ (induction hypothesis)}\\
&= \mathtt{exp}(r\gamma)
\end{aligned}$$

  or:

$$\begin{aligned}
&\mathtt{exp}(r)\gamma^{\mathtt{exp}}\\
&= @^\sigma(\mathtt{exp}(s)\gamma^{\mathtt{exp}}, \mathtt{exp}(t)\gamma^{\mathtt{exp}})\\
&\Rightarrow^*_{\mathcal{R}^{noapp}} @^\sigma(\mathtt{exp}(s\gamma), \mathtt{exp}(t\gamma)) \text{ (induction hypothesis)}\\
&= \mathtt{exp}(r\gamma)
\end{aligned}$$

  or:

$$\begin{aligned}
&\mathtt{exp}(r)\gamma^{\mathtt{exp}}\\
&= @(\mathtt{exp}(s)\gamma^{\mathtt{exp}}, \mathtt{exp}(t)\gamma^{\mathtt{exp}})\\
&\Rightarrow_{\mathcal{R}^{noapp}} (\mathtt{exp}(s)\gamma^{\mathtt{exp}}) \cdot (\mathtt{exp}(t)\gamma^{\mathtt{exp}})\\
&\Rightarrow^*_{\mathcal{R}^{noapp}} \mathtt{exp}(s\gamma^{\mathtt{exp}}) \cdot \mathtt{exp}(t\gamma^{\mathtt{exp}}) \text{ (induction hypothesis)}\\
&= \mathtt{exp}(r\gamma)
\end{aligned}$$

If s top-reduces to t with a β-step, we use that $\mathtt{exp}(q)[x := \mathtt{exp}(u)] \Rightarrow^*_{\mathcal{R}^{noapp}} \mathtt{exp}(q[x := u])$ as we just derived. □

Example 3.20. Choosing $S = \{\mathtt{nat} \to \mathtt{nat}\}$, $\mathcal{R}^{noapp}$ consists of:

$$\begin{aligned}
\mathtt{map}(F, \mathtt{nil}) &\Rightarrow \mathtt{nil}\\
\mathtt{map}(F, \mathtt{cons}(x, y)) &\Rightarrow \mathtt{cons}(@(F, x), \mathtt{map}(F, y))\\
\mathtt{pow}(F, \mathtt{0}) &\Rightarrow \lambda x.x\\
\mathtt{pow}(F, \mathtt{s}(x)) &\Rightarrow \mathtt{op}(F, \mathtt{pow}(F, x))\\
\mathtt{op}(F, G) \cdot x &\Rightarrow @(F, @(G, x))\\
\lambda x.@(F, x) &\Rightarrow F\\
\lambda x.\mathtt{op}(F, G) \cdot x &\Rightarrow \mathtt{op}(F, G)\\
@(F, x) &\Rightarrow F \cdot x
\end{aligned}$$

Removing Problematic Abstractions. Having guaranteed that the AFS we consider has no more leading free variables in the left-hand sides of rules, there are two more problems to deal with before we can transform AFSs into AFSMs: left-hand sides which are abstractions, and non-β-normal left-hand sides. Both issues can be solved in one go by "escaping" the responsible abstraction.

The solution is very similar to the one for leading variables: we identify the types of all abstractions which cause a problem, and replace abstractions $\lambda x.s$ of such a type σ by $\Lambda_\sigma(\lambda x.s)$, where Λ_σ is a new function symbol.

Formally, let Q be a set of types which contains all types σ such that $\mathcal{R}$ either contains a rule $\lambda x.s \Rightarrow r$, or a rule $C[(\lambda x.s) \cdot t]$ with $\lambda x.s : \sigma$. For every $\sigma \in Q$ introduce a new symbol $\Lambda^\sigma : [\sigma] \longrightarrow \sigma$. For all terms s, define $\mathtt{expL}(s)$ as follows:

$$\begin{array}{lcl}
\mathtt{expL}(f(s_1,\ldots,s_n)) & = & f(\mathtt{expL}(s_1),\ldots,\mathtt{expL}(s_n)) \\
\mathtt{expL}(s \cdot t) & = & \mathtt{expL}(s) \cdot \mathtt{expL}(t) \\
\mathtt{expL}(x) & = & x \ (x \text{ a variable}) \\
\mathtt{expL}(\lambda x.s) & = & \begin{cases} \Lambda^\sigma(\lambda x.\mathtt{expL}(s)) & \text{if } \lambda x.s : \sigma \text{ and } \sigma \in Q \\ \lambda x.\mathtt{expL}(s) & \text{otherwise} \end{cases}
\end{array}$$

Transformation 3.21 *(Marking Abstractions)* Let $\mathcal{R}^\Lambda := \{\mathtt{expL}(l) \Rightarrow \mathtt{expL}(r) \mid l \Rightarrow r \in \mathcal{R}^{noapp}\} \cup \{\Lambda^\sigma(x) \Rightarrow x \mid \sigma \in Q\}$

It is evident that $\mathcal{R}^\Lambda$ has no rule schemes of the form $\lambda x.l \Rightarrow r$ and its left-hand sides are β-normal. Moreover, the termination of this AFSM is equivalent to termination of the original system.

Theorem 3.22. *$\Rightarrow_{\mathcal{R}^\Lambda}$ is terminating if and only if $\Rightarrow_{\mathcal{R}}$ is.*

Proof. Defining s', t' as s, t with occurrences of any Λ_σ erased, it is easily seen that $s \Rightarrow_{\mathcal{R}^\Lambda} t$ implies $s' \Rightarrow^{=}_{\mathcal{R}^{noapp}} t'$, with equality only if the former was Λ-erasing. Thus, if $\Rightarrow_{\mathcal{R}^\Lambda}$ is non-terminating, then so is $\Rightarrow_{\mathcal{R}^{noapp}}$, and by Theorem 3.19 this implies non-termination of $\Rightarrow_{\mathcal{R}}$.

For the other direction, we will derive that $s \Rightarrow_{\mathcal{R}^{noapp}} t$ implies $\mathtt{expL}(s) \Rightarrow^{+}_{\mathcal{R}^\Lambda} \mathtt{expL}(t)$. Since non-termination of $\Rightarrow_{\mathcal{R}}$ implies non-termination of $\Rightarrow_{\mathcal{R}^\Lambda}$ by Theorem 3.19, this completes the proof.

To this end we make two observations:

1. $\mathtt{expL}(q\gamma) = \mathtt{expL}(q)\gamma^{\mathtt{expL}}$, where $\gamma^{\mathtt{expL}} = [x := \mathtt{expL}(\gamma(x)) \mid x \in dom(\gamma)]$;
2. $\mathtt{expL}(C[q]) = \mathtt{expL}(C)[\mathtt{expL}(q)]$.

Both follow easily with induction (on the form of q and C respectively).

We can write $s = C[q] \Rightarrow_{\mathcal{R}^{noapp}} C[u] = t$ and $q \Rightarrow_{\mathcal{R}^{noapp}} u$ by a topmost step; by (2) it suffices to show that $\mathtt{expL}(q) \Rightarrow_{\mathcal{R}^\Lambda} \mathtt{expL}(u)$.

Consider first a β-step. If $q = (\lambda x.v) \cdot w \Rightarrow_\beta v[x := w] = u$, then either $\mathtt{expL}(q) = \Lambda(\lambda x.\mathtt{expL}(v)) \cdot \mathtt{expL}(w) \Rightarrow_{\mathcal{R}^\Lambda} (\lambda x.\mathtt{expL}(v)) \cdot \mathtt{expL}(w)$, or $\mathtt{expL}(s) = (\lambda x.\mathtt{expL}(v)) \cdot \mathtt{expL}(w)$. Either way, $\mathtt{expL}(q) \Rightarrow^{=}_{\mathcal{R}^\Lambda} \cdot \Rightarrow_\beta \mathtt{expL}(v)[x := \mathtt{expL}(w)] = \mathtt{expL}(v[x := w]) = \mathtt{expL}(u)$ by (1).

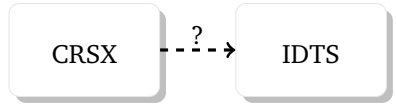
If q reduces to u by a rule step, let $q = l\gamma$ and $t = r\gamma$. By (1), $\mathtt{expL}(q) = \mathtt{expL}(l\gamma) = \mathtt{expL}(l)\gamma \Rightarrow_{\mathcal{R}^\Lambda} \mathtt{expL}(r)\gamma = \mathtt{expL}(u)$. □

Example 3.23. Continuing the transformation of $\mathcal{R}_{\mathtt{map}}$, we choose $Q = \{\mathtt{nat} \to \mathtt{nat}\}$. Since all abstractions in the rules from Example 3.20 have a type $\mathtt{nat} \to \mathtt{nat}$, Transformation 3.21 introduces the new symbol Λ around all abstractions:

$$\begin{array}{rcl}
\mathtt{map}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{map}(F, \mathtt{cons}(x, y)) & \Rightarrow & \mathtt{cons}(@(F, x), \mathtt{map}(F, y)) \\
\mathtt{pow}(F, \mathtt{0}) & \Rightarrow & \Lambda(\lambda x.x) \\
\mathtt{pow}(F, \mathtt{s}(x)) & \Rightarrow & \mathtt{op}(F, \mathtt{pow}(F, x)) \\
\mathtt{op}(F, G) \cdot x & \Rightarrow & @(F, @(G, x)) \\
\Lambda(\lambda x.@(F, x)) & \Rightarrow & F \\
\Lambda(\lambda x.\mathtt{op}(F, G) \cdot x) & \Rightarrow & \mathtt{op}(F, G) \\
@(F, x) & \Rightarrow & F \cdot x
\end{array}$$

Thus we have seen that the rule "left-hand sides of rules l must be β-normal terms of the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$, not containing any subterms headed by a variables which is free in l" can indeed be assumed to hold in any AFS.

3.5 Combinatory Reduction Systems with Extensions



The last higher-order rewriting system with types that we shall study (to a lesser extent than the others) is that of *Combinatory Reduction Systems with Extensions*, originally defined by Rose in [107]. CRSXs have seen major changes in recent years, and are almost not recognisable from the original definition. As a programming language (with features like syntactic sugar, pre-defined types, and many restrictions to allow for optimisations) the formalism is currently in use for specifying compilers at *International Business Machines* (IBM).

The most recent updates to the language and its underlying formalism, as well as their theoretical properties, are still very much a work in progress (see also the documentation at `http://crsx.sf.net/`). Therefore, I will not give a detailed overview, but rather discuss some of the formalism's peculiarities. In addition, I will mention some ideas for a transformation from CRSXs to IDTSs, but again, will not go into details (to formally describe the transformation would require a solid treatment of the underlying formalism first!).

3.5.1 Definition

The basis of Combinatory Reduction Systems with Extensions is very similar to that of IDTSs; that is, a combinatory reduction system is simply an AFSM without application. In addition, the output types of function symbols and meta-variable applications are always base types, so terms have a functional type if and only if they are abstractions. There are three special features:

- CRSXs are natively second-order systems, in a strong meaning of the word:
  - the type declaration of each meta-variable has order ≤ 2;
  - the type declaration of each function symbol has order ≤ 3.
- CRSX-rules can match on variables, provided they have a certain type, and the right-hand sides of rules may contain fresh variables.
- Some types carry *lookup tables*, a mapping of key/value pairs which can be checked for presence or absence of keys.

To sketch the idea, let us look at the example of a CRSX which encodes the typed λ-calculus, and provides functions to find the type of a closed λ-term.

Let $\mathcal{B}$ be the set $\{\texttt{term}, \texttt{type}, \texttt{string}\}$, and let $\mathcal{B}_v$, the set of *base types with syntactic variables*, consist of the base type `term`. We will use the following function symbols:

$$\mathcal{F} = \left\{ \begin{array}{rcl} \texttt{base} & : & [\texttt{string}] \longrightarrow \texttt{type} \\ \texttt{arrow} & : & [\texttt{type} \times \texttt{type}] \longrightarrow \texttt{type} \\ \texttt{app} & : & [\texttt{term} \times \texttt{term}] \longrightarrow \texttt{term} \\ \texttt{lam} & : & [\texttt{type} \times (\texttt{term} \to \texttt{term})] \longrightarrow \texttt{term} \\ \texttt{gettype} & : & [\texttt{term}] \longrightarrow \texttt{type} \\ \texttt{output} & : & [\texttt{type}] \longrightarrow \texttt{type} \end{array} \right\}$$

In addition, there are infinitely many function symbols of type string (in the CRSX-language, `string` is a pre-defined type).

The idea is that for example a λ-term $\lambda x.(x \cdot y)$ with $x : \texttt{a} \to \texttt{b}$ is represented by the CRSX-term $\texttt{lam}(\texttt{arrow}(\texttt{base}(\text{“a”}), \texttt{base}(\text{“b”})), \lambda x.\texttt{app}(x, y))$, with variables in the λ-term being represented by variables in its CRSX-representation. The function `gettype` will be used to map a term to its type.

We assign $\texttt{typemap}(\texttt{type}) = \{\texttt{term} : \texttt{type}\}$, which means that terms with type `type` have a lookup table which maps terms of type `term` to terms with type `type`. The `gettype` function determines the type of a term, using the following rules:

$$\begin{array}{rcl} \texttt{output}(\texttt{arrow}(X, Y)) & \Rightarrow & Y \\ \{G\}\ \texttt{gettype}(\texttt{app}(X, Y)) & \Rightarrow & \texttt{output}(\{G\}\ \texttt{gettype}(X)) \\ \{G\}\ \texttt{gettype}(\texttt{lam}(X, \lambda x.Y(x))) & \Rightarrow & \texttt{arrow}(X, \{G;\ z : X\}\ \texttt{gettype}(Y(z))) \\ \{G \mid x : X\}\ \texttt{gettype}(x) & \Rightarrow & X \\ \{G \mid \neg x\}\ \texttt{gettype}(x) & \Rightarrow & \texttt{base}(\text{“unknown”}) \end{array}$$

This example demonstrates a number of interesting things, which are explained below. First of all, the use of lookup tables. The lookup table is associated, implicitly or explicitly, with every occurrence of a function symbol with output type `type`. In fact, the first rule is short-hand notation for:

$$\{G\}\ \mathtt{output}(\{H\}\ \mathtt{arrow}(X, Y)) \Rightarrow Y$$

Unused lookup tables in the left-hand side of rules are generally omitted, as are empty lookup tables $\{\}$ in the right-hand sides of rules. In fact, this has been done in the second, third and fifth rules; for example the second rule should actually read:

$$\{G\}\ \mathtt{gettype}(\mathtt{app}(X, Y)) \Rightarrow \{\}\ \mathtt{output}(\{G\}\ \mathtt{gettype}(X))$$

In the first rule, the lookup tables are ignored; the rule simply selects an argument. This rule computes the output type of a given type. The second rule computes the type of an application $\mathtt{app}(s, t)$. Here, the lookup table is passed on unmodified to the recursive `gettype` call.

In the third rule, things get a little more interesting. Here, two things happen: the introduction of a fresh variable z in the right-hand side of the rule, and the addition of a key/value pair to a lookup table. The notation $\{G; z : X\}$ indicates that the lookup table G is updated with the key/value pair $z : Z$. The introduction of z is allowed only because z has type $\mathtt{term} \in \mathcal{B}_v$, so a type where variables have a special meaning. *Syntactic variables*, that is, variables with a type in $\mathcal{B}_v$, are also the only variables which may occur free in the left-hand sides of a rule. This would cause problems with confluence, if not for the following clause: *syntactic variables may only ever be substituted by other variables*. That is, if $Z(\ldots, s, \ldots)$ occurs in any rule and $s : \iota \in \mathcal{B}_v$, then s is a variable. We see that this is indeed satisfied in the given rules.

The variable z occurs as a new key in the lookup table $\{G; z : X\}$ in the third rule. The keys of a lookup table must be terms of a key-type (in this case `term`); either syntactic variables, or function symbols of arity 0, provided there is no rule which reduces this symbol. A very common type for lookup table keys is `string`.

The fourth rule assigns a type to a variable. Here we see how the lookup tables are used for the first time: the construction $\{G \mid z : X\}$ matches a lookup table which must contain a key z (where z is the variable that occurs as the argument of `gettype`), and assigns the value associated to z to the meta-variable X. Essentially, this construction does a lookup for the value of z in G. The fifth rule deals with the case that the key is not present: the construction $\{G \mid \neg z\}$ matches a lookup table which does not contain the key z.

3.5.2 From CRSX to AFSM

To represent a CRSX as an IDTS or AFSM, and thus be able to use the termination results of this thesis, we will have to deal both with syntactic variables, and with lookup tables. The easiest way to do this is to abstract them away almost entirely: for example, represent lookup tables as a list of values, and use non-determinism to choose a value. This is, however, a very strong abstraction, which may well lose termination. Instead, let us consider an embedding using a *constraints*. We could also see this as a very specific *reduction strategy* (which a termination tool might ignore to a lesser or greater extent).

A lookup table $\{k_1 : v_1, \ldots, k_n : v_n\}$ is represented by a list:

$$\texttt{table}(k_1, v_1, \texttt{table}(k_2, v_2, \ldots, \texttt{table}(k_n, v_n, \texttt{emptytable})\ldots))$$

We add some additional rules for looking up values:

$$\begin{array}{rcl} \texttt{lookup}(key, \texttt{emptytable}) & \Rightarrow & \texttt{absent} \\ \texttt{lookup}(key, \texttt{table}(key, v, rest)) & \Rightarrow & \texttt{present}(v) \\ \texttt{lookup}(key, \texttt{table}(k, v, rest)) & \Rightarrow & \texttt{lookup}(key, rest) \end{array}$$

And we consider the following constraints on the rewrite relation:

- a term of the form $\texttt{lookup}(s, \texttt{table}(t, q, u))$ may only be reduced at the top if s and t are both either a syntactic variable, or a function symbol which cannot be reduced;
- the third `lookup` rule may only be applied if the second is not applicable on a given term;
- if a term contains a subterm $\texttt{lookup}(\ldots)$, then it may not be reduced by any other rule than the three rules above.

Now the existing rules can be altered to use lookup tables as terms instead of special constructions. A rule

$$\{G \mid x : X\}\ \texttt{gettype}(x) \Rightarrow X$$

is for instance replaced by the following rules:

$$\begin{array}{rcl} \texttt{termwithtable}(G, \texttt{gettype}(x)) & \Rightarrow & \texttt{test}(\texttt{lookup}(G), \texttt{gettype}(x)) \\ \texttt{test}(\texttt{present}(X), \texttt{gettype}(x)) & \Rightarrow & X \end{array}$$

Similarly, a rule

$$\{G\}\ \texttt{gettype}(\texttt{lam}(X, \lambda x.Y(x))) \Rightarrow \texttt{arrow}(X, \{G;\ z : X\}\ \texttt{gettype}(Y(z)))$$

is replaced by the rule:

$$\begin{array}{l} \texttt{termwithtable}(G, \texttt{gettype}(\texttt{lam}(X, \lambda x.Y(x))) \Rightarrow \\ \quad \texttt{termwithtable}(\texttt{emptytable}, \\ \quad\quad \texttt{arrow}(X, \texttt{termwithtable}(\texttt{table}(z, X, G), \texttt{gettype}(Y(z))))) \end{array}$$

Of course this transformation creates the possibility for reductions which were previously not possible, and it may well lose termination. However, since every reduction in the old system can be simulated by a reduction in the altered CRSX, we at least preserve non-termination, and due to the reduction strategy we preserve the determinism of the original.

The other oddity in CRSXs, which AFSMs cannot deal with, is the use of syntactic variables. Let us consider their peculiarities:

- only syntactic variables may appear freely in either side of a rewrite rule;
- only syntactic variables may occur as keys in lookup tables (now that we have effectively removed lookup tables, this is not very relevant anymore);
- the rules are defined in such a way that a syntactic variable is never substituted by anything other than a variable.

The restriction on substitution in particular suggests that syntactic variables are closer to function symbols than to normal variables. Typically, syntactic variables are used to simulate computer addresses; the main function of them being variables is that in rewriting we already have the notion of a fresh variables, and not so much the notion of a fresh symbol.

To get rid of syntactic variables, consider a new base type `index`, and countably many symbols $\mathtt{1}, \mathtt{2}, \mathtt{3}, \ldots$ of type `index`. For every base type ι in $\mathcal{B}_v$, let $\mathtt{var}_\iota : [\mathtt{indexes}] \longrightarrow \iota$ be a function symbol. Intuitively, we can replace syntactic variables x in terms by $\mathtt{var}(x)$, and match on the occurrence of the symbol `var` rather than the variable itself. This avoids the problem of free variables on the left-hand sides; the fresh variables in the right-hand sides we can deal with later!

For the rules, this requires the following transformation:

- in the reduction strategy for the `lookup` rules, a term $\mathtt{var}(s)$ with $s : \mathtt{index}$ is considered a valid key;
- for all syntactic variables x which occur free in the left-hand side of a rule:
  - let Z_x be a uniquely corresponding meta-variable of type `index`;
  - if the right-hand side of the rule has any subterms $F(\ldots, x, \ldots)$, replace these by $F(\ldots, Z_x, \ldots)$ (the left-hand side is a pattern, so does not have such subterms);
  - wherever x still occurs in either side of the rules, replace it by $\mathtt{var}(Z_x)$;
- for all remaining syntactic variables x which occur free or bound in either side of a rule:
  - let x' be a uniquely corresponding variable of type `index`;
  - replace subterms $F(\ldots, x, \ldots)$ in either side by $F(\ldots, x', \ldots)$;
  - replace any remaining occurrences of x by $\mathtt{var}(x')$.

Having done this, the rules do not match on variables anymore, and it seems plausible (depending on the exact definition of the rewrite relation) that if $s \Rightarrow_{\mathcal{R}} t$ in the original system, then $\varphi(s) \Rightarrow_{\mathcal{R}'} \varphi(t)$ in the new system, where φ is the function which replaces all syntactic variables x by $\mathtt{var}(x')$.

This leaves the issue of fresh variables. As mentioned, AFSMs (and most other higher-order rewriting systems) do not really have a way to deal with this. Note that syntactic variables cannot just be substituted, so they are essentially harmless, far more so than bound variables. Therefore, let us consider a solution with a reduction strategy, which treats them as "harmless" symbols.

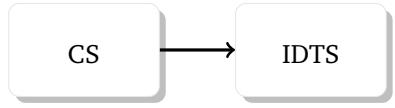
For every rule with fresh variables in the right-hand side, add countably many rules, each of which replaces the free variable by a different element of `index` (if there are multiple fresh variables, replace them all by distinct indexes). Consider the reduction strategy: *You may only use a rule with indexes in the right-hand side if the newly created indexes are not used anywhere in the term.* With this strategy, we effectively simulate the freshness constraint, but without introducing variables. The price is having infinitely many rules. However, these rules are all very similar, and an automatic tool could deal with them all at once by putting a special marker in place of the "fresh variables".

3.5.3 Overview

As described above, we can transform a CRSX into an IDTS without losing non-termination. Of course, this transformation is not very formal, and misses a number of important cases (for example, how to deal with lookup tables which have more than one type of keys). To properly study a transformation, we would first need to formally define the exact syntax and semantics of CRSXs. This is work in progress, but at present not done.

CRSXs also use a limited form of polymorphism, which we have not discussed here. This polymorphism is all but harmless, because type variables are treated much like base types. We could consider a polymorphic CRSX as generating a (possibly infinite) monomorphic CRSX (see also the discussion in Chapter 9.3). Or we could consider a type-changing function which collapses all base types, and also type variables, to the same base type.

3.6 Contraction Schemes



Although Aczel's Contraction Schemes are defined without types, the formalism uses a restriction on term formation which makes it possible to derive a typing. We will see how contraction schemes can be seen as IDTSs (that is, AFSMs where term formation does not use the application $\cdot$) with a single base type. This result may have some bearing on other formalisms with an arity restriction on term formation as well.

3.6.1 Definition

Terms in a Contraction Scheme are built from an infinite set of variables, and a signature $\mathcal{F}$ of *forms* $f : [k_1, \ldots, k_n]$ with all k_i natural numbers, according to the following clauses:

1. a variable is a term;
2. if $f : [k_1, \ldots, k_n] \in \mathcal{F}$ and $s_1, \ldots, s_n$ are terms, then also
$$f(\lambda x_{1,1}, \ldots, x_{1,k_1}.s_1, \ldots, \lambda x_{n,1}, \ldots, x_{n,k_n}.s_n)$$
is a term.

Meta-terms are built from variables, function symbols and an infinite set of meta-variables, each with a fixed arity. This uses the previous clauses (substituting "meta-term" for "term"), and in addition:

3. if Z is a meta-variable of arity n and $s_1, \ldots, s_n$ are meta-terms, then also $Z(s_1, \ldots, s_n)$ is a meta-term.

A *rewrite rule* is a pair $l \Rightarrow r$ of meta-terms such that all meta-variables in r also occur in l. In addition, the left-hand side l of a rule must satisfy the following restrictions:

1. l is closed (that is, all variables occur in the scope of a λ), although r does not need to be;
2. l is linear, that is, every meta-variable occurs at most once in l;
3. l is a fully extended pattern;
4. l has a depth of 1 or 2 (where $Z(\vec{x})$ has depth 0 and $f(\lambda\vec{x}_1.s_1, \ldots, \lambda\vec{x}_n.s_n)$ has depth $\max(\{\mathrm{depth}(s_i) \mid 1 \leq i \leq n\}) + 1$).

The rewrite relation generated by a set of rewrite rules $\mathcal{R}$, is the smallest relation $\Rightarrow_{\mathcal{R}}$ such that always $l\gamma \Rightarrow_{\mathcal{R}} r\gamma$ for $l \Rightarrow r \in \mathcal{R}$ and γ a substitution on domain $FMV(l)$.

Example 3.24. We could represent `map` from Example 2.3 as the following Contraction Scheme:

$$\begin{array}{rcl} \mathcal{F} & = & \{\mathtt{nil} : [],\ \mathtt{cons} : [0,0],\ \mathtt{map} : [1,0]\} \\ \mathcal{R} & = & \left\{ \begin{array}{l} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) \Rightarrow \mathtt{nil}, \\ \mathtt{map}(\lambda x.F(x), \mathtt{cons}(H,T)) \Rightarrow \mathtt{cons}(F(H), \mathtt{map}(\lambda x.F(x), T)) \end{array} \right\} \end{array}$$

3.6.2 From CS to AFSM

Since there are, to my knowledge, no termination results on contraction schemes, and the depth restriction is not present in other formalisms, I will only demonstrate how contraction schemes can be seen as a form of inductive data type systems.

An observation when comparing rules in a contraction scheme to IDTS-rules is that the former allows free variables in the right-hand sides while the latter does not. However, unlike CRSXs, contraction schemes have no way to *match* on a variable. Nor does it matter *which* variable is used, since the rules are required to be left-linear. Consequently, if we introduce a fresh function symbol v with form [], and replace free variables in the right-hand sides of rules by v, termination is not affected.

Lemma 3.25. *A CS is terminating if and only if the corresponding CS with all free variables in the rules replaced by the fresh symbol v, is terminating.*

Proof. Let $\mathcal{R}$ be the original set of CS-rules, and $\mathcal{R}'$ be the variation where fresh variables are replaced by v. Let φ_X be the function which maps all free variables in a (meta-)term, except those in X, to v. Then $\mathcal{R}'$ consists of rules $l \Rightarrow \varphi_\emptyset(r)$ with $l \Rightarrow r \in \mathcal{R}$, and we have $\varphi_\emptyset(l) = l$. It is easy to see that:

(**) $\varphi_X(s\gamma) = \varphi_X(s)[Z := \varphi_X(\gamma(Z)) \mid Z \in dom(\gamma)]$.

For one direction, we note that $s \Rightarrow_\mathcal{R} t$ implies $\varphi_\emptyset(s) \Rightarrow_{\mathcal{R}'} \varphi_\emptyset(t)$, which is trivial with induction on the size of s (the base case, a topmost step, uses (**)).

For the other direction, we note that if $\varphi_X(s) \Rightarrow_{\mathcal{R}'} q$, then we can find some t such that $s \Rightarrow_\mathcal{R} t$ and $q = \varphi_X(t)$. This holds by induction on the size of s. The base case, a topmost step, uses (**) and the observation that if $\varphi_X(s) = l\gamma$ for some linear pattern l and a substitution on domain $FMV(l)$, then $s = l\gamma'$ for some substitution such that $\gamma(Z) = \varphi_X(\gamma(Z))$ for all $Z \in dom(\gamma)$. This observation holds with a separate induction on the size of l, using linearity. □

Having made this modification, let $\mathtt{o}$ be a 0-ary type constructor and write σ_n for the type $\mathtt{o} \to \ldots \to \mathtt{o} \to \mathtt{o}$ with in total n arrows (so $n+1$ occurrences of $\mathtt{o}$). Let τ_n be the type declaration $[\mathtt{o} \times \ldots \times \mathtt{o}] \longrightarrow \mathtt{o}$ with $n+1$ occurrences of $\mathtt{o}$. A form $f : [k_1, \ldots, k_n]$ in $\mathcal{F}$ is translated to a function symbol $f : [\sigma_{k_1} \times \ldots \times \sigma_{k_n}] \longrightarrow \mathtt{o}$ in the set of IDTS-symbols $\mathcal{F}^{\mathtt{SI}}$. All variables are assigned type $\mathtt{o}$. Now every CS-style (meta-)term can be seen as an IDTS-style (meta-)term as well. Moreover, contraction scheme rules map to IDTS-rules; let $\mathcal{R}^{\mathtt{SI}}$ be the "translation" of $\mathcal{R}$. It is not hard see that every term in the IDTS over $\mathcal{F}^{\mathtt{SI}}$ which is not an abstraction uniquely corresponds to some CS-term. We can derive the following facts:

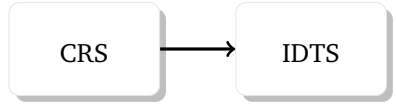
Lemma 3.26. *Let s, t be CS-terms, and s', t' be the corresponding IDTS-terms.*

1. *If $s \Rightarrow_\mathcal{R} t$, then $s' \Rightarrow_{\mathcal{R}^{\mathtt{SI}}} t'$.*
2. *If $s' \Rightarrow_{\mathcal{R}^{\mathtt{SI}}} t'$, then $s \Rightarrow_\mathcal{R} t$.*

Proof. Each case holds by a simple induction on the size of s. For the base case, a topmost step, we either have $s = l\gamma$ or $s' = l'\gamma'$, and use induction on the size of l or l'. □

Thus we have: if the original contraction scheme is non-terminating, then the resulting IDTS is non-terminating by Lemma 3.26. In addition, if the IDTS $(\mathcal{F}^{\mathtt{SI}}, \mathcal{R}^{\mathtt{SI}})$ is non-terminating, then there must be an infinite reduction not starting in an abstraction. Since every IDTS-term which is not an abstraction is also a CS-term, Lemma 3.26(2) provides that also the CS $(\mathcal{F}, \mathcal{R})$ is non-terminating. This gives the required solid arrow, that is, the equivalence of the natural embedding of contraction schemes into IDTSs.

3.7 Combinatory Reduction Systems



Finally, let us consider a truly untyped system: Klop's Combinatory Reduction Systems. CRSs were originally defined in 1980 [70], but the definition relayed here is the more commonly used version of [71].

3.7.1 Definition

Combinatory Reduction Systems extend first-order term rewriting with untyped λ-abstraction and meta-variables; essentially, they are IDTSs without type restrictions.[4] Formally, given a set $\mathcal{V}$ of variables, a set $\mathcal{M}$ of meta-variables and a signature $\mathcal{F}$ of function symbols, where the meta-variables and function symbols are equipped with an arity, the set of meta-terms is given by the grammar:

$$\mathbb{T} ::= x \mid \lambda x.\mathbb{T} \mid f(\mathbb{T}^n) \mid Z(\mathbb{T}^m) \qquad x \in \mathcal{V},\ f \in \mathcal{F},\ Z \in \mathcal{M},\ ar(f) = n,\ ar(Z) = m$$

Terms are meta-terms without meta-variables and are not subject to any type restrictions. *Rewrite rules* are pairs $l \Rightarrow r$ of closed meta-terms such that all meta-variables from r also occur in l, and moreover l is a CRS-pattern: all meta-variable occurrences in l have the form $Z(x_1, \ldots, x_n)$ with the x_i distinct variables. A substitution may either map variables to terms, or n-ary meta-variables to n-ary substitutes $\lambda x_1 \ldots x_n.t$ and works as in AFSMs. The rewrite relation $\Rightarrow_{\mathcal{R}}$ is the monotonic relation on CRS-terms generated by: $l\gamma \Rightarrow_{\mathcal{R}} r\gamma$ for $l \Rightarrow r \in \mathcal{R}$ and γ a substitution whose domain consists of the meta-variables in l.

Example 3.27. We could represent the system `map` from Example 2.3 as the CRS:

$$\begin{array}{lcl} \mathcal{F} & = & \{\mathtt{nil},\ \mathtt{cons},\ \mathtt{map}\},\ ar(\mathtt{nil}) = 0,\ ar(\mathtt{cons}) = 2,\ ar(\mathtt{map}) = 2 \\ \mathcal{R} & = & \left\{ \begin{array}{l} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) \Rightarrow \mathtt{nil}, \\ \mathtt{map}(\lambda x.Z(x), \mathtt{cons}(X, Y)) \Rightarrow \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y)) \end{array} \right\} \end{array}$$

[4] Although, considering CRSs predate IDTSs by twenty years, it would be more fair to say that IDTSs are CRSs with an additional type restriction!

Due to the untyped nature of CRSs, this system is non-terminating. This is demonstrated by the term $\Omega := \mathtt{map}(\omega, \mathtt{cons}(\omega, \mathtt{nil}))$, where the subterm ω is given by $\lambda x.\mathtt{map}(x, \mathtt{cons}(x, \mathtt{nil}))$ (note that this example closely corresponds to the example for termination of the untyped λ-calculus in Chapter 2.1.3). Using simple types as in an AFSM or IDTS, Ω cannot be typed, but it is a legal CRS-term. Using the second rule, Ω reduces to $\mathtt{cons}(\mathtt{map}(\omega, \mathtt{cons}(\omega, \mathtt{nil})), \mathtt{map}(\omega, \mathtt{nil})) = \mathtt{cons}(\Omega, \mathtt{map}(\omega, \mathtt{nil}))$, which contains the original term.

3.7.2 From CRS to AFSM

Their untyped nature makes CRSs less interesting for full termination results: as Example 3.27 demonstrates, many interesting and seemingly harmless CRSs are non-terminating. Nevertheless, CRSs can be embedded into typed systems without affecting termination. The transformation used here is very similar to the transformation from CRSs to HRSs used in [118, Ch.11.4.1]. Recall that an IDTS is an AFSM where terms are formed without the $\cdot$ operator for application.

Transformation 3.28 *(Transforming a CRS into an IDTS)* Given a CRS $(\mathcal{F}, \mathcal{R})$, let $\mathcal{B} = \{\mathtt{o}\}$ and define type declarations $\tau_n = [\mathtt{o} \times \ldots \times \mathtt{o}] \longrightarrow \mathtt{o}$ with n occurrences of $\mathtt{o}$ before the $\longrightarrow$. Let $\mathcal{F}^{\mathtt{CA}} = \{\Lambda : [\mathtt{o} \to \mathtt{o}] \longrightarrow \mathtt{o}\} \cup \{f' : \tau_n \mid f \in \Sigma,\ ar(f) = n\}$ and assume every CRS-variable x corresponds with an IDTS-variable x' of type $\mathtt{o}$, and every meta-variable Z of arity n with an IDTS-meta-variable Z' with type declaration τ_n. Let φ be the function mapping CRS-style (meta-)terms to IDTS-style (meta-)terms as follows:

$$\begin{array}{lcl}
\varphi(x) & = & x' \ (x \text{ a variable}) \\
\varphi(f(s_1, \ldots, s_n)) & = & f'(\varphi(s_1), \ldots, \varphi(s_n)) \ (f \in \mathcal{F}, ar(f) = n) \\
\varphi(Z(s_1, \ldots, s_n)) & = & Z'(\varphi(s_1), \ldots, \varphi(s_n)) \ (Z \in \mathcal{M}, ar(Z) = n) \\
\varphi(\lambda x.s) & = & \Lambda(\lambda x'.\varphi(s))
\end{array}$$

Let $\mathcal{R}^{\mathtt{CA}} := \{\varphi(l) \Rightarrow \varphi(r) \mid l \Rightarrow r \in \mathcal{R}\}$.

Theorem 3.29. *The CRS $(\mathcal{F}, \mathcal{R})$ is terminating if and only if the IDTS $(\mathcal{F}^{\mathtt{CA}}, \mathcal{R}^{\mathtt{CA}})$ is terminating.*

Proof. Simple inductions on the size of s show:

1. if s is a CRS-term and γ a substitution, then $\varphi(s\gamma) = \varphi(s)\gamma^{\varphi}$, where $\gamma^{\varphi} = [x' := \varphi(\gamma(x)) \mid x \in dom(\gamma)]$;
2. if s is a CRS-meta-term and δ a substitution whose domain contains all meta-variables in s (and no variables), then $\varphi(s\delta) = \varphi(s)\delta^{\varphi}$, where $\delta^{\varphi}(Z') = \lambda\vec{x}.\varphi(t)$ if $\delta(Z) = \lambda\vec{x}.t$ and $ar(Z) = n$; the case where $s = Z(\vec{s})$ uses (1);
3. if $s \Rightarrow_{\mathcal{R}} t$ then $\varphi(s) \Rightarrow_{\mathcal{R}^{\mathtt{CA}}} \varphi(t)$; the case where $s = l\delta$ and $t = r\delta$ for some CRS-rule $l \Rightarrow r$ uses (2);

By (3) every infinite reduction in the original CRS leads to an infinite reduction in the IDTS $(\mathcal{F}^{\mathtt{CA}}, \mathcal{R}^{\mathtt{CA}})$, which provides one direction.

For the other direction, an "inverse" of φ is needed. For every IDTS-variable x, choose a CRS-variable y_x (note that not every IDTS-variable has the form z' for some CRS-variable z, since only base type variables can have this form). Now define ψ as the function which, essentially, drops types from an IDTS-term:

$$\begin{array}{rcl} \psi(x) & = & y_x \\ \psi(\lambda x.s) & = & \lambda y_x.\psi(s) \\ \psi(f'(s_1,\ldots,s_n)) & = & f(\psi(s_1),\ldots,\psi(s_n)) \\ \psi(\Lambda(s)) & = & \psi(s) \end{array}$$

Now a simple induction on the size of s shows:

4. If s is an IDTS-term and γ a substitution, then $\psi(s\gamma) = \psi(s)\gamma^\psi$, where $\gamma^\psi = [y_x := \psi(\gamma(x)) \mid x \in dom(\gamma)]$.

5. If s is a CRS-meta-term and δ an IDTS-substitution whose domain contains all meta-variables in $\varphi(s)$ and no variables, then $\psi(\varphi(s)\delta) = s\delta^\psi$, where $\delta^\psi(Z) = \lambda y_{x^1} \ldots y_{x^n}.\psi(t)$ if $\delta(Z') = \lambda x^1 \ldots x^n.t$; the case where $s = Z(s_1,\ldots,s_n)$ (so $\psi(\varphi(s)\delta) = \psi(t[x^1 := \varphi(s_1)\delta,\ldots,x^n := \varphi(s_n)\delta])$ and $s\delta^\psi = \psi(t)[y_x^1 := s_1\delta^\psi,\ldots,y_x^n := s_n\delta^\psi]$) can be handled with the induction hypothesis and (4).

6. If s is an IDTS-term and $s \Rightarrow_{\mathcal{R}^{\mathtt{CA}}} t$, then $\psi(s) \Rightarrow_{\mathcal{R}} \psi(t)$; the case where $s = \varphi(l)\delta \Rightarrow_{\mathcal{R}^{\mathtt{CA}}} \varphi(r)\delta = t$ uses (5).

By (6) every infinite reduction in the IDTS $(\mathcal{F}^{\mathtt{CA}}, \mathcal{R}^{\mathtt{CA}})$ leads to an infinite reduction in the CRS $(\mathcal{F}, \mathcal{R})$, which completes the proof. □

3.7.3 Confluence

Since the focus of this work is on termination, we have so far not discussed whether confluence is preserved by any of the transformations. However, to have some results available, it is worthwhile to make some brief observations on the relation between CRSs and AFSMs.

Every AFSM can be seen as a CRS, if we drop the types, view the application operator as a binary infix function symbol, and add a single rule $(\lambda x.F(x)) \cdot X \Rightarrow F(X)$. If the original AFSM is $(\mathcal{F}, \mathcal{R})$, then let $(\mathcal{F}^{\mathtt{AC}}, \mathcal{R}^{\mathtt{AC}})$ be the resulting CRS. Of course, this CRS is utterly non-terminating (it implements the untyped λ-calculus), but it has the following properties:

- every AFSM-meta-term over $\mathcal{F}$ is a CRS-meta-term over $\mathcal{F}^{\mathtt{AC}}$;
- if $s \Rightarrow_{\mathcal{R}} t$ in the original AFSM, then $s \Rightarrow_{\mathcal{R}^{\mathtt{AC}}} t$ in the resulting CRS;
- if s is an AFSM-term over $\mathcal{F}$, and $s \Rightarrow_{\mathcal{R}^{\mathtt{AC}}} t$, then t is also an AFSM-term over $\mathcal{F}$, and $s \Rightarrow_{\mathcal{R}} t$.

Thus, if the CRS $(\mathcal{F}^{\mathtt{AC}}, \mathcal{R}^{\mathtt{AC}})$ is confluent, then so is the AFSM $(\mathcal{F}, \mathcal{R})$. Knowing this we immediately obtain all available confluence results which are available for CRSs, such as the fact that an orthogonal CRS is confluent [71]. We will use this in Chapter 7.7.

Definition 3.30 (Orthogonality)**.** An AFSM is *non-overlapping* if for every pair of rules $l \Rightarrow r, u \Rightarrow v$, and subterm l' of u we have: if there are substitutions γ, δ such that $l'\gamma = u\delta$, then either l' has the form $Z(x_1, \ldots, x_n)$ for some meta-variable Z and variables $x_1, \ldots, x_n$, or $l' = l = u$ and $r = v$.

An AFSM with rules $\mathcal{R}$ is *orthogonal* if it is left-linear and non-overlapping.

Theorem 3.31. *An orthogonal AFSM is confluent.*

Proof. If an AFSM is orthogonal, then so is the underlying CRS: orthogonality for CRSs is defined in exactly the same way, and due to the pattern restriction the left-hand side of a rule does not overlap with the redex $@(\lambda x.F(x), X)$ for the β-rule. Since orthogonality implies confluence for CRSs, this implies confluence for the original AFSM as well. □

3.8 Overview

This chapter discusses a variety of higher-order formalisms, in particular *Pattern Higher-order Rewriting Systems*, *Algebraic Functional Systems* and *Inductive Data Type Systems*, all of which have both λ-abstraction and use simple types. For each of these formalisms we have found an embedding into AFSMs, which preserves at least non-termination. For the formalism of *Combinatory Reduction Systems with Extensions* we have sketched a similar transformation. For demonstration purposes, embeddings for two untyped systems into AFSMs have also been given. Consequently, to prove termination for a system in any of these formalisms, it suffices to translate the system into an AFSM and use the techniques from this thesis to show termination of the result.

For those systems where termination techniques have been defined (PRSs, AFSs and IDTSs) we also have a non-termination-preserving embedding from AFSMs to any of these systems. As such, existing techniques can be translated to the AFSM formalism, as is done at several places in this work. Moreover, following the arrows in Figure 3.2, these transformations make it possible to transfer termination results between existing formalisms.

Although confluence is of relatively little interest in this work, we have also seen that confluence results for CRSs naturally extend to AFSMs.

Polynomial Interpretations

4

Or, Can we think of "Plus" as "+"?

One of the most prominent techniques in termination proofs for first-order term rewriting is the use of *polynomial interpretations*. In this method, which dates back to the seventies [92], terms are mapped to polynomial functions over some well-founded set, such as the natural numbers. For example, to prove termination of the TRS

$$\begin{array}{rcl} \mathtt{plus}(\mathtt{0}, y) & \Rightarrow & y \\ \mathtt{plus}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{s}(\mathtt{plus}(x, y)) \end{array}$$

we can use an interpretation in the natural numbers, assigning to the symbol $\mathtt{0}$ the natural number 0, to $\mathtt{s}$ the function $\boldsymbol{\lambda} n.n + 1$ and to $\mathtt{plus}$ the function $\boldsymbol{\lambda} nm.2 \cdot n + m + 1$. Then the interpretations of the rules are strictly decreasing:

$$\begin{array}{rcccccl} [\![\mathtt{plus}(\mathtt{0}, y)]\!] & = & y + 1 & > & y & = & [\![y]\!] \\ [\![\mathtt{plus}(\mathtt{s}(x), y)]\!] & = & 2 \cdot x + y + 3 & > & 2 \cdot x + y + 2 & = & [\![\mathtt{s}(\mathtt{plus}(x, y))]\!] \end{array}$$

Consequently, the interpretation of a term decreases when the term is rewritten, and therefore the TRS is terminating.

Polynomial interpretations often give very intuitive (hand-written) termination proofs, since a TRS is usually written with a meaning in mind, which may be modelled by the interpretation – to some extent. In the example above, the successor $\mathtt{s}$ is modelled by $\boldsymbol{\lambda} n.n + 1$. However, the naive interpretation for $\mathtt{plus}$, $\boldsymbol{\lambda} nm.n + m$, does not work: with this interpretation both sides of the rule are mapped to the same number. Apart from hand-written proofs, the method is also automatable, and has been implemented in various automatic tools, such as AProVE [45], $\mathrm{T_TT_2}$ [83] and Jambox [34].

Polynomial interpretations are an instance of the *monotonic algebra* approach, which also includes for instance *matrix interpretations* [35].

In his 1996 thesis [104], van de Pol proposes an extension of monotonic algebras to higher-order term rewriting. More precisely, to the HRS formalism described in Chapter 3.3 (without a pattern restriction). In this extension, terms with a functional type, such as $\lambda x.\mathtt{0}$, are mapped to *weakly monotonic functions*. Surprisingly, the method has seen little interest in the literature since, while methods

based on a *computability* reasoning, such as HORPO (see Chapter 5.1.3), have flourished.

In this chapter, I will first present the basic definitions and results for the class of weakly monotonic functionals, as defined for HRSs in [104], and extend these results to AFSMs using a transformation (Section 4.1). This gives only a weak reduction pair, which cannot be used directly for termination proofs, but we will then consider strongly monotonic functionals, and see how these can be used to prove termination of an AFSM in Section 4.2. This result is a simplification of the *strict functionals* used in the original method. In Section 4.3 we will study the class of *higher-order polynomials*. This class of weakly monotonic functionals can be used as an interpretation domain for the function symbols of an AFSM.

A proof-of-concept implementation of the method is discussed in Chapter 8.5.

This chapter is based on [42], where polynomial interpretations for the class of AFSs are introduced. The paper also discusses the implementation, which is detailed in Chapter 8.5.

4.1 Weakly Monotonic Functionals

Background. In the first-order definition of monotonic algebras [35], terms are mapped to elements of a well-founded target domain $(A, >, \geq)$. This is done by choosing an interpretation function $\mathcal{J}(f)$ for all function symbols f in a given signature, and a valuation $\alpha(x)$ for all variables. These interpretations are extended homomorphically to an interpretation $[\![\cdot]\!]_{\mathcal{J},\alpha}$ of terms. The interpretation function must be monotonic w.r.t. $>$ and $\geq$. In the definition of *polynomial interpretations*, $\mathcal{J}(f)$ is always a polynomial.

If $[\![l]\!]_{\mathcal{J},\alpha} > [\![r]\!]_{\mathcal{J},\alpha}$ for all valuations α of the free variables of l and r, then $[\![C[l\gamma]]\!]_{\mathcal{J},\alpha} > [\![C[r\gamma]]\!]_{\mathcal{J},\alpha}$ for all contexts C, substitutions γ and valuations α. Thus, if there is an infinite reduction over $\Rightarrow_{\mathcal{R}}$, then there is also an infinite decreasing $>$-chain, contradicting well-foundedness of $>$. More precisely, the relations $>$ and $\geq$ on A induce a strong reduction pair on terms.

Type Interpretations. In higher-order term rewriting we have to deal with infinitely many types (due to the type constructor $\to$), a complication which is not present in first-order term rewriting. As a consequence, it is not very practical to map all terms to the same target set. A more natural interpretation would be, for instance, to map a functional term $\lambda x.s : \mathsf{o} \to \mathsf{o}$ to an element of the function space $\mathbb{N} \to \mathbb{N}$. However, this choice has problems of its own, since we then have to reason about functions that we have no information about. For example, if we do not know anything about F, we can only be sure that a constraint $F(a) \geq F(b)$ holds if $a = b$, which is very restrictive.

Instead, the target domain for interpreting terms, as proposed by van de Pol in [104], is the class of *weakly monotonic functionals*. To be precise, to each type σ we assign a set $\mathcal{WM}_\sigma$ and two relations: a well-founded ordering $\sqsupset_\sigma$ and a quasi-

ordering $\sqsupseteq_\sigma$ (see Section 2.4.1 for definitions of these concepts). Intuitively, the elements of $\mathcal{WM}_{\sigma\to\tau}$ are functions which preserve the quasi-ordering $\sqsupseteq$, but not necessarily the strict ordering $\sqsupset$.

We consider the following definition from [104, Def.4.1.1]:

Definition 4.1 (Weakly Monotonic Functionals)**.** We assume given a *well-founded set*: a triple $\mathcal{A} = (A, >, \geq)$ of a non-empty set, a well-founded strict ordering on that set and a quasi-ordering that is compatible with it.[1,2]

To each type σ we associate a set $\mathcal{WM}_\sigma$ of *weakly monotonic functionals of type* σ and two relations $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$, defined inductively as follows.

For a base type ι:

- $\mathcal{WM}_\iota = A$;
- $\sqsupset_\iota = >$, and $\sqsupseteq_\iota = \geq$.

For a functional type $\sigma\to\tau$:

- $\mathcal{WM}_{\sigma\to\tau}$ consists of the functions f from $\mathcal{WM}_\sigma$ to $\mathcal{WM}_\tau$, such that $\sqsupseteq$ is preserved: if $x \sqsupseteq_\sigma y$ then $f(x) \sqsupseteq_\tau f(y)$;
- $f \sqsupset_{\sigma\to\tau} g$ iff $f(x) \sqsupset_\tau g(x)$ for all $x \in \mathcal{WM}_\sigma$;
- $f \sqsupseteq_{\sigma\to\tau} g$ iff $f(x) \sqsupseteq_\tau g(x)$ for all $x \in \mathcal{WM}_\sigma$.

Thus, $\mathcal{WM}_{\sigma\to\tau}$ is a subset of the function space $\mathcal{WM}_\sigma \to \mathcal{WM}_\tau$, consisting of functions which preserve the $\sqsupseteq$ relation. Note that both the $\mathcal{WM}_\sigma$ and the relations $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ should be considered as parametrised with $\mathcal{A}$; the complete notation would be $\mathcal{WM}^{\mathcal{A}}_\sigma, \sqsupset^{\mathcal{A}}_\sigma, \sqsupseteq^{\mathcal{A}}_\sigma$. However, for readability, $\mathcal{A}$ will normally be omitted, as will the type denotations for the various $\sqsupset_\sigma$ and $\sqsupseteq_\sigma$ relations; if $x, y \in \mathcal{WM}_\sigma$, then $x \sqsupseteq y$ should be read as $x \sqsupseteq_\sigma y$. The phrase "f is weakly monotonic" means that $f \in \mathcal{WM}_\sigma$ for some σ.

It is worth noting that we use the mathematical definition of a function as a set of pairs; a function is specified entirely by its domain and values. Thus, if F and G are both functions in $\mathcal{WM}_\sigma$ for some σ, and $F(x) = G(x)$ for all x in their domain, then $F = G$. We will use the (extensional) notation $\boldsymbol{\lambda} x.P(x)$ for a function that takes one argument x, and returns $P(x)$.

It is not hard to see that an element $\boldsymbol{\lambda} x_1 \ldots x_n.P(x_1, \ldots, x_n)$ of the function space $\mathcal{WM}_{\sigma_1} \to \ldots \to \mathcal{WM}_{\sigma_n} \to A$ is weakly monotonic if and only if:

$$\forall N_1, M_1 \in \mathcal{WM}_{\sigma_1}, \ldots, N_n, M_n \in \mathcal{WM}_{\sigma_n} :$$
$$\text{if each } N_i \sqsupseteq M_i \text{ then } P(N_1, \ldots, N_n) \sqsupseteq P(M_1, \ldots, M_n)$$

[1] In [104], van de Pol defines $\geq$ as the reflexive closure of $>$, but in recent definitions of monotonic algebras it is common to separate $>$ and $\geq$. This is needed for example for interpretations in the rational numbers.

[2] Several definitions of *well-founded set* appear in the literature, but the basis is a pair of a set A and a well-founded ordering $>$. The definition used here is a generalisation of this notion; if $\geq$ is the reflexive closure of $>$, then the two definitions coincide.

Since all base types are interpreted by the same set A, the sets $\mathcal{WM}_\sigma$ and $\mathcal{WM}_\tau$ are the same if σ and τ are equal modulo collapsing of base types.

> *Comment:* The definition in [104] actually assigns a different well-founded set $\mathcal{A}_\iota$ to each base type ι (although there must be an addition operator $+_{\iota,\kappa,\iota}$ for every pair of base types). Here we use the same set for all base types, as this leads to a simpler definition, and it is not obvious whether using different sets gives a stronger technique. We could for instance choose $\mathcal{A}$ as the disjoint union of $\mathcal{A}_\iota$ and $\mathcal{A}_\kappa$ instead of using two different base sets. Moreover, all examples in [104] use a single "base" set, as do the polynomial interpretations of Section 4.3.
> Also, in [104] $\mathcal{WM}_{\sigma\to\tau}$ consists of functions f in a larger function space $\mathcal{I}_\sigma \to \mathcal{I}_\tau$ such that $f(x) \in \mathcal{WM}_\tau$ if $x \in \mathcal{WM}_\sigma$ and f preserves $\sqsupseteq$. Here, $\mathcal{I}_\iota = \mathcal{A}_\iota$ if $\iota \in \mathcal{B}$, and $\mathcal{I}_{\sigma\to\tau}$ is the full function space $\mathcal{I}_\sigma \to \mathcal{I}_\tau$. The definition here is simpler to present, but essentially equivalent; every function in $\mathcal{WM}_\sigma \to \mathcal{WM}_\tau$ can be extended to a function in $\mathcal{I}_\sigma \to \mathcal{I}_\tau$.

By [104, Lemma 4.1.4] the relation $\sqsupset$ is a well-founded ordering, the relation $\sqsupseteq$ is a quasi-ordering, and $\sqsupseteq$ includes $\sqsupset$. Since here we did not take $\geq$ as the reflexive closure of $>$, the last statement does not always hold anymore, but we can derive a compatibility result instead:

Lemma 4.2. *For all types σ the following statements hold:*

- $\sqsupset_\sigma$ *is well-founded;*
- $\sqsupset_\sigma$ *and* $\sqsupseteq_\sigma$ *are both transitive;*
- $\sqsupseteq_\sigma$ *is reflexive;*
- $\sqsupset_\sigma$ *and* $\sqsupseteq_\sigma$ *are compatible;*
- $\mathcal{WM}_\sigma$ *is non-empty.*

Proof. We prove the lemma with induction on the type σ. Assume (IH) that for all strict subtypes τ of σ, $\sqsupset_\tau$ is well-founded, both $\sqsupset_\tau$ and $\sqsupseteq_\tau$ are transitive, $\sqsupseteq_\tau$ is reflexive, $\sqsupset_\tau$ and $\sqsupseteq_\tau$ are compatible and $\mathcal{WM}_\tau$ is non-empty.

For σ a base type, we immediately have well-foundedness, transitivity, reflexivity, compatibility and non-emptiness, by the assumptions on $>$, $\geq$ and A. For $\sigma = \tau \to \rho$, we obtain:

$\sqsupset_\sigma$ **is well-founded:** Suppose, towards a contradiction, that $f_1 \sqsupset_\sigma f_2 \sqsupset_\sigma f_3 \sqsupset_\sigma \ldots$. Let $a \in \mathcal{WM}_\tau$ (such a exists by IH). By definition of $\sqsupset_\sigma$, also $f_1(a) \sqsupset_\rho f_2(a) \sqsupset_\rho f_3(a) \sqsupset_\rho \ldots$, contradicting well-foundedness of $\sqsupset_\rho$.

$\sqsupseteq_\sigma$ **is transitive:** Suppose $f \sqsupseteq_\sigma g \sqsupseteq_\sigma h$. Then for all $x \in \mathcal{WM}_\tau$ we have $f(x) \sqsupseteq_\rho g(x) \sqsupseteq_\rho h(x)$ by definition of $\sqsupseteq_\sigma$, so by the induction hypothesis $f(x) \sqsupseteq_\rho h(x)$ for all $x \in \mathcal{WM}_\tau$. This exactly means that $f \sqsupseteq_\sigma h$.

$\sqsupset_\sigma$ **is transitive:** Same as for $\sqsupseteq_\sigma$.

$\sqsupseteq_\sigma$ **is reflexive:** Let $f \in \mathcal{WM}_\sigma$. Then $f \sqsupseteq_\sigma f$ iff for all $x \in \mathcal{WM}_\tau$: $f(x) \sqsupseteq_\rho f(x)$. But this holds by reflexivity of $\sqsupseteq_\rho$ (IH).

$\sqsupset_\sigma$ **and** $\sqsupseteq_\sigma$ **are compatible:** Let $f, g, h \in \mathcal{WM}_\sigma$ and suppose $f \sqsupset_\sigma g \sqsupseteq_\sigma h$. Then for all $x \in \mathcal{WM}_\sigma$ we have $f(x) \sqsupset_\rho g(x) \sqsupseteq_\rho h(x)$, so $f(x) \sqsupset_\rho h(x)$ by (IH), and therefore $f \sqsupset_\sigma h$.

$\mathcal{WM}_\sigma$ **is non-empty:** Let $a \in \mathcal{WM}_\rho$; the function $g := \boldsymbol{\lambda} n : \mathcal{WM}_\tau . a$ is in $\mathcal{WM}_\sigma$ because if $x \sqsupseteq_\tau y$, then $g(x) = a \sqsupseteq_\rho a = g(y)$ by reflexivity of $\sqsupseteq_\rho$ (IH). □

Note that $n \sqsupseteq m$ does not in general imply that $n \sqsupset m$ or $n = m$, even if $\geq$ is the reflexive closure of $>$. For example, using functions over the natural numbers, let $f := \boldsymbol{\lambda} x.x$ (that is, the function which takes some argument n and returns it), and $g := \boldsymbol{\lambda} x.0$. Then $f \sqsupseteq g$ (because for all n we have $f(n) = n \geq 0 = g(n)$), but not $f \sqsupset g$ (because not $f(0) > g(0)$), and also not $f = g$ (because $f(1) \neq g(1)$).

Term Interpretations for λ-terms. The goal is to associate to every term of some type σ an element of $\mathcal{WM}_\sigma$; thus, terms of functional type will be mapped to functions. The original definition from [104] targets HRSs rather than our AFSMs, but the ideas translate easily to AFSMs. Moreover, we will not have to redo any proofs; we can transpose the original results using a transformation. All that we will really need from [104] are the results related in Lemma 4.4.

Definition 4.3 (Interpreting a λ-term as a Weakly Monotonic Functional)**.** Given a well-founded set $\mathcal{A} = (A, >, \geq)$, a simply-typed λ-term s and a *valuation* α – a function which assigns to all variables $x : \sigma$ in $FV(s)$ an element of $\mathcal{WM}_\sigma$ –, let $[s]_\alpha$ be defined by the following recursive clauses:

$$\begin{array}{lcll} [x]_\alpha & = & \alpha(x) & \text{if } x \in \mathcal{V} \\ [s \cdot t]_\alpha & = & [s]_\alpha([t]_\alpha) & \\ [\lambda x.s]_\alpha & = & \boldsymbol{\lambda} n.[s]_{\alpha \cup \{x \mapsto n\}} & \end{array}$$

Definition 4.3 is an instance of a definition in [104], which defines term interpretations for HRS-pre-terms. The definition for λ-terms suffices to transpose the proof of the relevant results to AFSMs. From [104] we obtain:

Lemma 4.4 (Facts about Algebra Interpretations)**.**

1. (Respecting β and η) *If* $s \updownarrow_\beta^\eta = t \updownarrow_\beta^\eta$ *then also* $[s]_\alpha = [t]_\alpha$ *for all* α.
2. (Substitution Lemma) *Given a substitution* $\gamma = [x_1 := s_1, \ldots, x_n := s_n]$ *and a valuation* α *whose domain does not include the* x_i*, we have:* $[s\gamma]_\alpha = [s]_{\alpha \circ \gamma}$ *(where* $\alpha \circ \gamma$ *is the valuation* $\alpha \cup \{x_1 \mapsto [s_1]_\alpha, \ldots, x_n \mapsto [s_n]_\alpha\}$*).*
3. (Weak Monotonicity of Interpretations) *If* $s : \sigma$ *is a simply-typed* λ*-term, then* $[s]_\alpha \in \mathcal{WM}_\sigma$ *for all valuations* α.

Proof. By Proposition 3.2.2, Lemma 3.2.1 and Proposition 4.1.5(1) in [104] respectively. Note that we here use a slightly different definition of $\mathcal{WM}$ than van de Pol does, in that we do not require that $\geq$ is the reflexive closure of $>$ (we also choose the same base set for all base types, but this was permitted by the definitions in [104] as well). This is not a problem, however: the first two statements do not depend on the definition of $\mathcal{WM}$, and Proposition 4.1.5(1) only uses reflexivity of $\sqsupseteq_\sigma$, and the following facts:

- if $f \in \mathcal{WM}_{\sigma\to\tau}$ and $x \in \mathcal{WM}_\sigma$, then $f(x) \in \mathcal{WM}_\tau$;
- if $f \sqsupseteq_{\sigma\to\tau} g$ and $x \in \mathcal{WM}_\sigma$, then $f(x) \sqsupseteq g(x)$;
- if $f \sqsupset_{\sigma\to\tau} g$ and $x \in \mathcal{WM}_\sigma$, then $f(x) \sqsupset g(x)$;
- if $f \in \mathcal{WM}_{\sigma\to\tau}$ and $x, y \in \mathcal{WM}_\sigma$ and $x \sqsupseteq y$ then $f(x) \sqsupseteq f(y)$.

All of these still hold. □

Lemma 4.4(3) provides a way to find weakly monotonic functionals, for example:

Example 4.5 (Weakly Monotonic Functionals)**.**

1. For all $n \in A$ and types $\sigma = \sigma_1 \to \ldots \to \sigma_k \to \iota$, let $n_\sigma := \boldsymbol{\lambda}\vec{x}.n_\sigma$. This *constant function* is in $\mathcal{WM}_\sigma$, because it is $[\lambda x_1 \ldots x_k.y]_{\{y\mapsto n\}}$.
2. Suppose A is equipped with a symbol 0 which is a minimal element for the ordering $>$. Then for any type $\sigma = \sigma_1 \to \ldots \to \sigma_n \to \iota$, the *lowest value* function, $\boldsymbol{\lambda}f.f(\vec{0})$, which maps a functional $f \in \mathcal{WM}_\sigma$ to the constant $f(0_{\sigma_1}, \ldots, 0_{\sigma_n}) \in A$, is a weakly monotonic functional in $\mathcal{WM}_{\sigma\to\iota}$ because it is $[\lambda f x_1 \ldots x_n.f \cdot x_1 \cdots x_n]_{\{x_1\mapsto 0_{\sigma_1},\ldots,x_n\mapsto 0_{\sigma_n}\}}$.
3. *Maximum Function:* In the natural numbers, the function $\max$ which assigns to any two numbers the highest of the two is weakly monotonic, since $\max(a,b) \geq \max(a',b')$ if $a \geq a'$ and $b \geq b'$. For any type $\tau = \tau_1 \to \ldots \to \tau_k \to \iota$ (with $\iota \in \mathcal{B}$) let $\max_\tau(f,m) = \boldsymbol{\lambda}x_1 \ldots x_k.\max(f(x_1,\ldots,x_k),m)$. This function is in $\mathcal{WM}_{\tau\to\iota\to\tau}$ because it is $[\lambda x_1 \ldots x_n.a \cdot (y \cdot x_1 \cdots x_k) \cdot z]_\alpha$, where $\alpha = [a \mapsto \max, y \mapsto f, z \mapsto m]$.

Term Interpretations for AFSMs. The interpretation of Definition 4.3 is not entirely suitable for our purposes, mostly because of the way application is handled: the definition has been designed in such a way that $[s]_\alpha = [t]_\alpha$ if $s =_\beta t$. While this is useful for HRSs (where terms are equivalence classes modulo $\alpha\beta\eta$), it is not very convenient for AFSMs if for instance $[\![(\lambda x.\mathsf{0}) \cdot t]\!] = [\![(\lambda x.\mathsf{0}) \cdot q]\!]$ regardless of the values of t and q. But, recalling Transformation 3.6, we could think of application as a sort of function symbol. Or, if we use a transformation to simply-typed λ-terms, as a special variable.

Definition 4.6 (Weakly Monotonic Algebras for AFSMs)**.** For any type declaration $\sigma = [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau$, let $\mathtt{cur}(\sigma)$ denote the type $\sigma_1 \to \ldots \to \sigma_n \to \tau$.

A *weakly monotonic algebra* for an AFSM with function symbols $\mathcal{F}$ consists of a well-founded set $\mathcal{A} = (A, >, \geq)$ and an *interpretation function* $\mathcal{J}$ which assigns to all $f : \sigma \in \mathcal{F}$ an element of $\mathcal{WM}_{\mathtt{cur}(\sigma)}$, and which also assigns a value in $\mathcal{WM}_{\sigma\to\sigma}$ to the fresh symbol $@^\sigma$ for all functional types σ.

Given a weakly monotonic algebra $(\mathcal{A}, \mathcal{J})$, a meta-term s over $\mathcal{F}$ and a *valuation* α which assigns to all variables $x : \sigma$ in $FV(s)$ an element of $\mathcal{WM}_\sigma$, and to all meta-variables $Z : \sigma$ in $FMV(s)$ an element of $\mathcal{WM}_{\mathtt{cur}(\sigma)}$, let $[\![s]\!]_{\mathcal{J},\alpha}$ be defined recursively by the following clauses:

$$\begin{array}{lcll}
[\![x]\!]_{\mathcal{J},\alpha} & = & \alpha(x) & \text{if } x \in \mathcal{V} \\
[\![Z(s_1,\ldots,s_n)]\!]_{\mathcal{J},\alpha} & = & \alpha(Z)([\![s_1]\!]_{\mathcal{J},\alpha},\ldots,[\![s_n]\!]_{\mathcal{J},\alpha}) & \text{if } Z \in \mathcal{M} \\
[\![f(s_1,\ldots,s_n)]\!]_{\mathcal{J},\alpha} & = & \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha},\ldots,[\![s_n]\!]_{\mathcal{J},\alpha}) & \text{if } f \in \mathcal{F} \\
[\![s \cdot t]\!]_{\mathcal{J},\alpha} & = & \mathcal{J}(@^\sigma)([\![s]\!]_{\mathcal{J},\alpha},[\![t]\!]_{\mathcal{J},\alpha}) & \text{if } s : \sigma \\
[\![\lambda x.s]\!]_{\mathcal{J},\alpha} & = & \boldsymbol{\lambda} n.[\![s]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}} & \text{if } x \notin dom(\alpha)
\end{array}$$

This definition assigns to every function symbol, variable and meta-variable a weakly monotonic functional, and calculates the value of the meta-term accordingly. For the purposes of the interpretation, application is treated as a function symbol $@^\sigma$. The interpretation function $\mathcal{J}$ for the function symbols is separate from the valuation α of the variables and meta-variables because the former is normally fixed, while we will quantify over the latter. Definition 4.6 roughly follows the ideas in [104], where also an interpretation function $\mathcal{J}$ and separate valuation are used. It conservatively extends the first-order definitions in [35].

Example 4.7. Consider our usual `map` signature, and let $\mathcal{A} = (\mathbb{N}, >, \geq)$, where $>$ is the usual greater than relation in the natural numbers and $\geq$ its reflexive closure. Choose:

$$\begin{array}{rcl}
\mathcal{J}(\mathtt{cons}) & = & \boldsymbol{\lambda} nm.n + m \\
\mathcal{J}(\mathtt{map}) & = & \boldsymbol{\lambda} Fn.F(n) \\
\mathcal{J}(\mathtt{s}) & = & \boldsymbol{\lambda} n.n + 2 \\
\mathcal{J}(\mathtt{0}) & = & 1 \\
\alpha(z) & = & 37
\end{array}$$

Then $[\![\mathtt{map}(\lambda x.\mathtt{s}(x), \mathtt{cons}(\mathtt{s}(\mathtt{0}), z))]\!]_{\mathcal{J},\alpha} = [\![F(n)]\!]_{\mathcal{J},\{F\mapsto\boldsymbol{\lambda} m.m+2, n\mapsto 40\}} = 42$.

Theorem 4.8 (Weakly Monotonic Algebras for AFSMs)**.** *Let $\mathcal{J}$ be an interpretation function, s, t meta-terms and γ a substitution. For all valuations α, the following statements hold:*

1. $[\![s]\!]_{\mathcal{J},\alpha} \in \mathcal{WM}_\sigma$ *if* $s : \sigma$.
2. $[\![s]\!]_{\mathcal{J},\alpha\circ\gamma} = [\![s\gamma]\!]_{\mathcal{J},\alpha}$ *(where* $\alpha \circ \gamma = \alpha \cup \{x \mapsto [\![\gamma(x)]\!]_{\mathcal{J},\alpha} \mid x \in dom(\gamma)\}$*).*
3. *If* $[\![s]\!]_{\mathcal{J},\theta} \sqsupseteq [\![t]\!]_{\mathcal{J},\theta}$ *for all valuations* θ*, then* $[\![s\gamma]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![t\gamma]\!]_{\mathcal{J},\alpha}$.
   If $[\![s]\!]_{\mathcal{J},\theta} \sqsupset [\![t]\!]_{\mathcal{J},\theta}$ *for all valuations* θ*, then* $[\![s\gamma]\!]_{\mathcal{J},\alpha} \sqsupset [\![t\gamma]\!]_{\mathcal{J},\alpha}$.

Proof. The proof proceeds by translating AFSM-meta-terms to simply-typed λ-terms, and then reusing the original result. Interpretation of function symbols ($\mathcal{J}$) is translated to assignment of variables (α), and application is treated as a function symbol. We use the following transformation:

$$\begin{array}{rcll} \varphi(x) & = & x & (x \in \mathcal{V}) \\ \varphi(Z(s_1,\ldots,s_n)) & = & x_Z \cdot \varphi(s_1) \cdots \varphi(s_n) & (Z \in \mathcal{M}) \\ \varphi(f(s_1,\ldots,s_n)) & = & x_f \cdot \varphi(s_1) \cdots \varphi(s_n) & (f \in \mathcal{F}) \\ \varphi(s \cdot t) & = & x_{@^\sigma} \cdot \varphi(s) \cdot \varphi(t)) & (s : \sigma) \\ \varphi(\lambda x.s) & = & \lambda x.\varphi(s) & \end{array}$$

Here, the x_f is a new variable of type $\mathtt{cur}(\sigma)$ for $f : \sigma \in \mathcal{F} \cup \{@^\sigma \mid \sigma \in \mathcal{T},\ \sigma \text{ functional}\}$, and x_Z is a special variable of type $\mathtt{cur}(\sigma)$ for a given meta-variable $Z : \sigma$. Note that this transformation is almost exactly Transformation 3.6, only function symbols are mapped to variables instead of other function symbols, and we don't bother η-expanding since $[\![\ ]\!]_{\mathcal{J},\alpha}$ is defined on pre-terms anyway.

For any substitution γ, let γ^φ denote the substitution which sends x to $\varphi(\gamma(x))$ for $x \in FV(s)$, and x_Z to $\varphi(\gamma(Z))$ for $Z \in FMV(s)$ (the x_f and $x_{@^\sigma}$ are left alone). We make the following observation:

(I) $\varphi(s\gamma) =_\beta \varphi(s)\gamma^\varphi$ for all substitutions γ.

This holds by induction first on the number of meta-variables in $dom(\gamma)$, second on the form of s. The only non-trivial case is when $s = Z(s_1,\ldots,s_n)$. In that case, let $\gamma(Z) = \lambda x_1 \ldots x_n.q$. On the one hand, $\varphi(s\gamma) = \varphi(q[x_1 := s_1\gamma,\ldots,x_n := s_n\gamma])$. On the other, $\varphi(s)\gamma^\varphi = \varphi(Z) \cdot \varphi(s_1)\gamma^\varphi \cdots \varphi(s_n)\gamma^\varphi$, which by the second induction hypothesis and definition of φ is equal modulo β to $(\lambda\vec{x}.\varphi(q)) \cdot \varphi(s_1\gamma) \cdots \varphi(s_n\gamma) \Rightarrow_\beta^* \varphi(q)[x_1 := \varphi(s_1\gamma),\ldots,x_n := \varphi(s_n\gamma)]$. Thus, using the first induction hypothesis (the substitution $[\vec{x} := \vec{s\gamma}]$ has no meta-variables in its domain, while γ does), both sides are equal modulo β.

In addition, note that:

(II) $[\![s]\!]_{\mathcal{J},\alpha} = [\varphi(s)]_\chi$, if $\chi(x) = \alpha(x)$ for $x \in FV(s)$, $\chi(x_Z) = \alpha(Z)$ for $Z \in FMV(s)$ and $\chi(x_f) = \mathcal{J}(f)$ for all x_f occurring in $\varphi(s)$.

This holds by induction on the definition of φ; all cases are obvious.

Now we can prove the statements in the theorem.

(1) holds by (II) and Lemma 4.4(3).

(2) holds by the Substitution Lemma and the two observations above: $[\![s\gamma]\!]_{\mathcal{J},\alpha} = [\varphi(s\gamma)]_\chi$ by (II), $= [\varphi(s)\gamma^\varphi]_\chi$ by (I) and Lemma 4.4(1), $= [\varphi(s)]_{\chi\circ\gamma^\varphi}$ by Lemma 4.4(2), which is exactly $[\![s]\!]_{\mathcal{J},\alpha\circ\gamma}$ by (II).

(3) holds by (2): $[\![s\gamma]\!]_{\mathcal{J},\alpha} = [\![s]\!]_{\mathcal{J},\alpha\circ\gamma}$ by (2), $\sqsupseteq [\![t]\!]_{\mathcal{J},\alpha\circ\gamma}$ because the statement speaks of *all* valuations, $= [\![t\gamma]\!]_{\mathcal{J},\alpha}$. The case with $\sqsupset$ is exactly the same. □

For the time being, we can ignore the rest of the theory of [104] and continue only with Definition 4.6 and Theorem 4.8, in addition to the results about the class of weakly monotonic functionals. We will need one more result, which does not have a counterpart in [104] because van de Pol did not consider situations like rule removal, where $\sqsupseteq$ must induce a monotonic quasi-ordering.

Lemma 4.9. *Let $(\mathcal{A}, \mathcal{J})$ be a weakly monotonic algebra for an AFSM $(\mathcal{F}, \mathcal{R})$, and s, t terms in this AFSM. If $[\![s]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α, then $[\![C[s]]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![C[t]]\!]_{\mathcal{J},\alpha}$ for all valuations α and contexts C.*

Proof. This is a straightforward induction on the form of C. The base case ($C = \Box_\sigma$) is evident, otherwise suppose (IH) $[\![D[s]]\!]_{\mathcal{J},\chi} \sqsupset [\![D[t]]\!]_{\mathcal{J},\chi}$ for all valuations χ, and let α be an arbitrary valuation. Consider the form of C.

If $C[] = \lambda x.D[]$, an abstraction, then $[\![C[s]]\!]_{\mathcal{J},\alpha} = \boldsymbol{\lambda} n.[\![D[s]]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}}$ and we are done because, by (IH) and the definition of $\sqsupseteq_\sigma$ for functional types, $\boldsymbol{\lambda} n.[\![D[s]]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}} \sqsupseteq \boldsymbol{\lambda} n.[\![D[t]]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}} = [\![C[t]]\!]_{\mathcal{J},\alpha}$.

If $C[] = f(s_1, \ldots, D[], \ldots, s_n)$, a functional term, then $[\![C[s]]\!]_{\mathcal{J},\alpha} = \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![D[s]]\!]_{\mathcal{J},\alpha}, \ldots, [\![s_n]\!]_{\mathcal{J},\alpha})$. Weak monotonicity of $\mathcal{J}(f)$ implies that if any of the arguments $\sqsupseteq$-decreases, then so does the result. Thus, by (IH) also $[\![C[s]]\!]_{\mathcal{J},\alpha} \sqsupseteq \mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![D[t]]\!]_{\mathcal{J},\alpha}, \ldots, [\![t_1]\!]_{\mathcal{J},\alpha}) = [\![C[t]]\!]_{\mathcal{J},\alpha}$.

The cases where $C[] = D[] \cdot q$ or $C[] = q \cdot D[]$ are very similar. □

With the theory so far we can create a *weak reduction pair* (see Definition 2.24).

Theorem 4.10. *Let a weakly monotonic algebra $(\mathcal{A}, \mathcal{J})$ be given such that always $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} fn.f(n)$, and define the pair $(\succsim, \succ)$ by: $s \succsim t$ if $[\![s]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α, and $s \succ t$ if $[\![s]\!]_{\mathcal{J},\alpha} \sqsupset [\![t]\!]_{\mathcal{J},\alpha}$ for all α. Then $(\succsim, \succ)$ is a weak reduction pair.*

Proof. $(\succsim, \succ)$ is a compatible combination of a quasi-ordering and a well-founded ordering by Lemma 4.4, both relations are stable (so certainly meta-stable) by Theorem 4.8(3) and $\succsim$ is monotonic by Lemma 4.9. Also, $\succsim$ contains `beta`: for all valuations α, $[\![(\lambda x.s) \cdot t]\!]_{\mathcal{J},\alpha} = \mathcal{J}(@^\sigma)([\![\lambda x.s]\!]_{\mathcal{J},\alpha}, [\![t]\!]_{\mathcal{J},\alpha}) \sqsupseteq [\![\lambda x.s]\!]_{\mathcal{J},\alpha}([\![t]\!]_{\mathcal{J},\alpha})$ by assumption, which equals $[\![s]\!]_{\mathcal{J},\alpha\cup\{x\mapsto [\![t]\!]_{\mathcal{J},\alpha}\}} = [\![s]\!]_{\mathcal{J},\alpha\circ[x:=t]}$, and this equals $[\![s[x := t]]\!]_{\mathcal{J},\alpha}$ by Theorem 4.8(2). □

> *Comment:* if we choose $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn.f(n)$, we have a system very similar to the one used for simply-typed λ-calculus (and HRSs). By not fixing the interpretation of $@^\sigma$ we have a choice, which will be essential to do rule removal, and also for the dependency pair approach of Chapter 6.

Example 4.11. We consider an interpretation in $\mathbb{N}$ for the AFSM `map` from Example 2.3.

$$\begin{array}{rcl} \mathcal{J}(\texttt{nil}) & = & 1 \\ \mathcal{J}(\texttt{cons}) & = & \boldsymbol{\lambda} nm.n + m + 1 \\ \mathcal{J}(\texttt{map}) & = & \boldsymbol{\lambda} Fn.F(0) + 2 \cdot n + n \cdot F(n) \\ \mathcal{J}(@^{\sigma\to\tau}) & = & \boldsymbol{\lambda} Fn.F(n) + n \text{ for all types } \sigma, \tau \end{array}$$

It is not hard to check that each of these functions is in $\mathcal{WM}$. Moreover, taking $\alpha = \{F \mapsto f,\ X \mapsto n,\ Y \mapsto m\}$, we have:

- $[\![\texttt{map}(\lambda x.F(x), \texttt{nil})]\!]_{\mathcal{J},\alpha} = f(0) + f(1) + 2 > 1 = [\![\texttt{nil}]\!]_{\mathcal{J},\alpha}$;

- $[\![\mathtt{map}(\lambda x.F(x), \mathtt{cons}(X,Y))]\!]_{\mathcal{J},\alpha}$
  $= f(0)+2\cdot n+2\cdot m+2+n\cdot f(n+m+1)+m\cdot f(n+m+1)+f(n+m+1)$
  $= (1+f(n+m+1)+f(0)+2\cdot m+m\cdot f(n+m+1))+2\cdot n+1+$
  $n\cdot f(n+m+1)$
  $> 1+f(n)+f(0)+2\cdot m+m\cdot f(m)$
  $= [\![\mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y))]\!]_{\mathcal{J},\alpha}$.

The last inequality holds because $f(n+m+1) \geq f(n)$ and $f(n+m+1) \geq m$ for all weakly monotonic f. Here we see the advantage of considering interpretations in $\mathcal{WM}$ rather than interpreting higher-order terms with arbitrary functions. Reasonings of the form "$f(a) \geq f(b)$ because $a \geq b$ and f is weakly monotonic" are very prevalent in termination proofs using weakly monotonic algebras, and will return in Chapter 8.5.

However, we won't be able to do much with a weak reduction pair until Chapter 6. In Example 4.11 we found a reduction pair $(\succsim, \succ)$ such that $l \succsim r$ for both rules in the system, but this does not suffice to prove termination. To use for instance Theorem 2.23, we must have a *strong reduction pair*, where the generated strict equality $\succ$ is monotonic.

In the next section, we will see how to make sure that this property is satisfied.

Comment: An often-suggested possibility is to use strongly monotonic functionals for the interpretation domain rather than weakly monotonic ones. Not only would this immediately give a strong reduction pair, it would also be easier to derive strict inequalities (as we would have that $F(n) > F(m)$ if $n > m$).
However, in the setting where λ-abstraction and β-reduction are present, this is not a very convenient interpretation domain. This is due to terms like $\lambda x.0$, where the abstracted variable is not part of the body of the abstraction. To interpret terms like this with a strongly monotonic functional, we would not be able to interpret abstraction in the obvious way. Van de Pol explores this avenue briefly in [104], but it is significantly more complicated, and seems less powerful, than interpretations using weakly monotonic functionals.

4.2 Strongly Monotonic Functionals

To use weakly monotonic algebras as a reduction ordering, or a strong reduction pair, we will need an additional property: if $s \sqsupset t$, then also $C[s] \sqsupset C[t]$. We achieve this by posing the restriction that $\mathcal{J}$ is *strongly monotonic*:

Definition 4.12. An element f of $\mathcal{WM}_{\sigma_1\to\ldots\to\sigma_n\to o}$ is *strongly monotonic in argument i* if for all $N_1 \in \mathcal{WM}_{\sigma_1}, \ldots, N_n \in \mathcal{WM}_{\sigma_n}$ and $M_i \in \mathcal{WM}_{\sigma_i}$ we have:

$$f(N_1,\ldots,N_i,\ldots,N_n) \sqsupset f(N_1,\ldots,M_i,\ldots,N_n) \text{ if } N_i \sqsupset M_i$$

We will require that the $\mathcal{J}(f)$ are strongly monotonic in all arguments required by their arity. This is not special for the higher-order approach; the same strong monotonicity requirement appears in the first-order approach [35].

Example 4.13. Let $\mathcal{A} = (\mathbb{N}, >)$.

- the constant 37 is strongly monotonic in all arguments (it has none);
- the function $\boldsymbol{\lambda} nmk.2^n \cdot (m+1) + k \cdot n$ is strongly monotonic in its first two arguments, but not in its third;
- the function $\boldsymbol{\lambda} f : \mathcal{WM}_{\mathtt{o}\to(\mathtt{o}\to\mathtt{o})\to\mathtt{o}}.\boldsymbol{\lambda} n : \mathcal{WM}_{\mathtt{o}}.f(f(0, \boldsymbol{\lambda} x.0), \boldsymbol{\lambda} x.n + x) + n$ is strongly monotonic in both arguments;
- for every type $\sigma{=}\sigma_1 \to \ldots \to \sigma_n \to \iota$, the function $\boldsymbol{\lambda} x_1 \ldots x_n.x_1(\vec{0}) + \ldots + x_n(\vec{0}) \in \mathcal{WM}_\sigma$ is strongly monotonic in all n arguments (where $x_i(\vec{0})$ is the lowest value function from Example 4.5(2)) – thus, there are strongly monotonic functionals of all types.

For symbols $f : \sigma \in \mathcal{F}$ with $order(\sigma) \leq 3$, strong monotonicity corresponds to the notion *strict* in [104]. For functions with a higher order, the definition of [104] is more permissive.[3] Here we consider strong monotonicity instead of strictness because the strictness requirement significantly complicates the theory of [104], and adoption of a technique often correlates with its simplicity. Moreover, in most common examples of higher-order systems the function symbols have order ≤ 3. As we saw in Example 4.13, strongly monotonic functionals exist for all types.

Definition 4.14 (Extended Monotonic Algebra)**.** A weakly monotonic algebra $(\mathcal{A}, \mathcal{J})$ for an AFSM with function symbols $\mathcal{F}$ is an *extended monotonic algebra* if $\mathcal{J}(f)$ is strongly monotonic in its first n arguments for all $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}$, and each $\mathcal{J}(@^\sigma)$ is strongly monotonic in its first 2 arguments.

Note that the phrase "strongly monotonic in its first n arguments" refers to the elements of the function space $\mathcal{WM}_\sigma$ or $\mathcal{WM}_{\mathtt{cur}(\sigma)}$. For example, $\mathcal{J}(@^{\mathtt{o}\to\mathtt{o}\to\mathtt{o}})$ is an element of the function space $\mathcal{WM}_{\mathtt{o}\to\mathtt{o}\to\mathtt{o}} \to \mathcal{WM}_{\mathtt{o}} \to \mathcal{WM}_{\mathtt{o}} \to \mathcal{WM}_o$, a function which takes *three* arguments. It does not need to be strongly monotonic in its 3rd argument, because we think of application as a function symbol $@^{\sigma\to\tau} : [(\sigma \to \tau) \times \sigma] \longrightarrow \tau$ of arity *2*, where τ may be functional.

Now we arrive at the key result. An extended monotonic algebra for higher-order rewriting is defined pretty much the same as an extended monotonic algebra for first-order rewriting – except the target domain is the class $\mathcal{WM}$ instead of some basic set like the natural numbers. To use weakly monotonic algebras in a strong reduction pair, it only remains to be seen that the relation on terms induced by $\sqsupset$ is monotonic.

[3] For strictness, it is only required that $f(\ldots, m_i, \ldots) \sqsupset f(\ldots, k_i, \ldots)$ if $m_i \sqsupset^{strict} k_i$, where $\sqsupset^{strict}$ is a subrelation of $\sqsupset$.

Lemma 4.15. *Let $(\mathcal{A}, \mathcal{J})$ be an extended monotonic algebra and s, t terms. If $[\![s]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α, then $[\![C[s]]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![C[t]]\!]_{\mathcal{J},\alpha}$ for all valuations α and contexts C.*

Proof. By induction on the form of C. If C is the empty context this is evident, otherwise suppose the assumption already holds for some context D.

If $C[] = \lambda x.D[]$, then $[\![C[s]]\!]_{\mathcal{J},\alpha} = \boldsymbol{\lambda} n.[\![D[s]]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}}$ and we are done because, by the induction hypothesis and definition of $\sqsupseteq$ for functions, this function $\sqsupseteq \boldsymbol{\lambda} n.[\![D[t]]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}} = [\![C[t]]\!]_{\mathcal{J},\alpha}$.

If $C[] = f(s_1, \ldots, D[], \ldots, s_n)$, and $\mathcal{J}(f) = A$ (a strongly monotonic function by assumption), then $[\![C[s]]\!]_{\mathcal{J},\alpha} = A([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![D[s]]\!]_{\mathcal{J},\alpha}, \ldots, [\![s_n]\!]_{\mathcal{J},\alpha})$. By the induction hypothesis, $[\![D[s]]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![D[t]]\!]_{\mathcal{J},\alpha}$, so by strong monotonicity of A in argument i, also $[\![C[s]]\!]_{\mathcal{J},\alpha} \sqsupseteq A([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![D[t]]\!]_{\mathcal{J},\alpha}, \ldots, [\![s_n]\!]_{\mathcal{J},\alpha}) = [\![C[t]]\!]_{\mathcal{J},\alpha}$.

The cases where $C[] = D[] \cdot q$ or $C[] = q \cdot D[]$ are very similar, since $@^\sigma$ is strongly monotonic in its first two arguments. □

> *Comment:* This result is similar to [104, Theorem 4.3.4], but does not require variable valuations to be strongly monotonic. This is possible because of the strong monotonicity requirements on the $@^\sigma$ interpretations.

We thus conclude the background theory of weakly monotonic algebras:

Theorem 4.16. *Let an extended monotonic algebra $(\mathcal{A}, \mathcal{J})$ be given such that always $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} f n.f(n)$. Then the pair $(\succsim, \succ)$ from Theorem 4.10 is a strong reduction pair.*

Theorem 4.16 holds by Theorem 4.10 and Lemma 4.15. It corresponds roughly to [104, Theorem 5.1.4] (taking into account the different formalism).

Example 4.17. The interpretation given in Example 4.11 is strongly monotonic in all arguments for the symbols `nil`, `cons` and `map`. The interpretation of $\mathcal{J}(@^{\sigma\to\tau})$ is strongly monotonic in its first two arguments as required (even though it is not strongly monotonic in its third argument, if it has any).

Thus, the algebra $(\mathcal{A}, \mathcal{J})$ for the signature `map` is an extended monotonic algebra. By Theorem 4.16, the generated pair $(\succsim, \succ)$ is a strong reduction pair.

As we saw in Example 4.11, $l \succ r$ for both rules. Consequently, by Theorem 2.23, the AFSM `map` is terminating.

4.3 Polynomial Interpretations in the Natural Numbers

It remains to be seen how to *find* extended monotonic algebras, preferably automatically. In this section, we will discuss the class of *higher-order polynomials in* $\mathbb{N}$, a specific subclass of the weakly monotonic functionals with $(\mathbb{N}, >, \geq)$ as a well-founded base set. Polynomials are weakly monotonic functionals by nature, and we can pose restrictions to enforce strong monotonicity.

Definition 4.18 (Higher-order Polynomial in $\mathbb{N}$)**.** For a set of variables $X = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, each equipped with a type, the set $Pol(X)$ of *higher-order polynomials* over X is given by the following clauses:

- if $n \in \mathbb{N}$, then $n \in Pol(X)$;
- if $p_1, p_2 \in Pol(X)$, then $p_1 + p_2 \in Pol(X)$;
- if $p_1, p_2 \in Pol(X)$, then $p_1 \cdot p_2 \in Pol(X)$;
- if $x_i : \tau_1 \to \ldots \to \tau_m \to \iota \in X$ with $\iota \in \mathcal{B}$, and $p_1 \in Pol^{\tau_1}(X), \ldots, p_m \in Pol^{\tau_m}(X)$, then $x_i(p_1, \ldots, p_m) \in Pol(X)$
  - here, $Pol^{\iota}(X) = Pol(X)$ for a base type ι, and $Pol^{\sigma\to\tau}(X)$ contains functions $\boldsymbol{\lambda} y \in \mathcal{WM}_\sigma.p$ with $p \in Pol^{\tau}(X \cup \{y\})$.

Note that X, in Definition 4.18, is not fixed. That is, the set $Pol(X)$ is defined for all sets X of variables. A *higher-order polynomial* is an element of any $Pol(X)$.

Noting that $\mathcal{WM}_\sigma = \mathcal{WM}_\tau$ if σ and τ have the same type "form" (so are equal modulo renaming of base types), the following lemma holds for all $\iota \in \mathcal{B}$:

Lemma 4.19. *If $p \in Pol(\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\})$, then $\boldsymbol{\lambda} x_1 \ldots x_n.p$ is a weakly monotonic functional in $\mathcal{WM}_{\sigma_1\to\ldots\to\sigma_n\to\iota}$.*

Proof. First note: *$\boldsymbol{\lambda} nm.n + m$ and $\boldsymbol{\lambda} nm.n \cdot m$ are weakly monotonic functionals in $\mathcal{WM}_{\iota\to\iota\to\iota}$.* This is easy to see.

The lemma holds by induction on the derivation of $p \in Pol(\{\vec{x}\})$.

If $p \in \mathbb{N}$, then $\boldsymbol{\lambda}\vec{x}.p$ is a constant function; its weak monotonicity was demonstrated in Example 4.5(1).

If $p_1, p_2 \in Pol(\{\vec{x}\})$, then by the induction hypothesis $\boldsymbol{\lambda}\vec{x}.p_1$ and $\boldsymbol{\lambda}\vec{x}.p_2$ are both weakly monotonic functionals. Consider the λ-term $L := \lambda y_1 \ldots y_n.A \cdot (F_1 \cdot \vec{y}) \cdot (F_2 \cdot \vec{y})$. By Lemma 4.4(3) $[L]_\alpha$ is a weakly monotonic functional, where $\alpha = \{A \mapsto \boldsymbol{\lambda} nm.n + m,\ F_1 \mapsto \boldsymbol{\lambda}\vec{x}.p_1,\ F_2 \mapsto \boldsymbol{\lambda}\vec{x}.p_2\}$. Thus, $[L]_\alpha = \boldsymbol{\lambda}\vec{x}.p_1 + p_2 \in \mathcal{WM}_{\sigma_1\to\ldots\to\sigma_n\to\iota}$. In the same way (using a valuation with $A \mapsto \boldsymbol{\lambda} nm.n \cdot m$), $\boldsymbol{\lambda}\vec{x}.p_1 \cdot p_2$ is a weakly monotonic functional.

Finally, suppose $p_1 \in Pol^{\tau_1}(\vec{x}), \ldots, p_m \in Pol^{\tau_m}(\vec{x})$, and x_i has type $\tau_1 \to \ldots \to \tau_m \to \iota$. By the induction hypothesis, each $\boldsymbol{\lambda}\vec{x}.p_i \in \mathcal{WM}_{\vec{\sigma}\to\vec{\tau}\to\iota}$. Thus, $\boldsymbol{\lambda}\vec{x}.x_i(\vec{p}) = [\![\lambda\vec{y}.y_i \cdot (z_1 \cdot \vec{y}) \cdots (z_m \cdot \vec{y})]\!]_{\{z_1\mapsto p_1,\ldots,z_m\mapsto p_m\}}$ is a weakly monotonic functional by Lemma 4.4(3). □

Higher-order polynomials are typically represented in the form $a_1 + \ldots + a_n$ (with $n \geq 0$), where each a_i is a *higher-order monomial*, that is, an expression of the form $b \cdot c_1 \cdots c_m$, where $b \in \mathbb{N}$ and each c_i is either a variable x with base type, or a function application $x(\boldsymbol{\lambda}\vec{y}_1.p_1, \ldots, \boldsymbol{\lambda}\vec{y}_k.p_k)$ with all p_j higher-order polynomials again. Examples of higher-order polynomials in the natural numbers are for instance the polynomial 0 and the polynomial $3 + 5 \cdot x^2 \cdot y + F(37 + x)$. To find a *strongly monotonic* functional, it suffices to include, for all variables, a monomial containing only that variable:

Theorem 4.20. *Let $P(x_1, \ldots, x_n)$ be a higher-order polynomial of the form $p_1(\vec{x}) + \ldots + p_m(\vec{x})$, where all $p_i(\vec{x})$ are higher-order monomials. Then $\boldsymbol{\lambda}\vec{x}.P(\vec{x})$ is strongly monotonic in argument i if there is some p_j of the form $a \cdot x_i(\vec{b}(\vec{x}))$, with $a \in \mathbb{N}^+$.*

Proof. Let weakly monotonic functionals $N_1, \ldots, N_n$ be given, and some i, j such that $p_j = a \cdot x_i(\vec{b}(\vec{x}))$ with $a \in \mathbb{N}^+$. It suffices to see that if $N_i \sqsupset M_i$, then also $P(N_1, \ldots, N_i, \ldots, N_n) > P(N_1, \ldots, M_i, \ldots, N_n)$. In the following, $\vec{N}$ is short notation for $N_1, \ldots, N_i, \ldots, N_n$ and $\vec{N'}$ is short notation for $N_1, \ldots, M_i, \ldots, N_n$.

By Lemma 4.19, $\boldsymbol{\lambda}\vec{x}.p_k(\vec{x})$ is a weakly monotonic functional for all k, and this implies that $p_k(\vec{N}) \geq p_k(\vec{N'})$ (since $N_i \sqsupset M_i$ implies $N_i \sqsupseteq M_i$ when $\mathcal{A} = (\mathbb{N}, >, \geq)$). If, moreover, $p_j(\vec{N}) > p_j(\vec{N'})$, then we obtain $P(\vec{N}) > P(\vec{N'})$, as required, by the nature of the addition operator.

Write $p_j(\vec{x}) = a \cdot d(x_i, \vec{x})$, where $d(y, \vec{x}) = y(\boldsymbol{\lambda}\vec{z_1}.b_1(\vec{x}, \vec{z}), \ldots, \boldsymbol{\lambda}\vec{z_m}.b_m(\vec{x}, \vec{z}))$. Since all b_k are polynomials, Lemma 4.19 provides: $\boldsymbol{\lambda}\vec{y_j}.b_j(\vec{N}, \vec{y}) \geq \boldsymbol{\lambda}\vec{y_j}.b_j(\vec{N'}, \vec{y})$. Thus, by weak monotonicity of N_i we know $d(N_i, \vec{N}) \geq d(N_i, \vec{N'})$. By the definition of $N_i \sqsupset M_i$, we also see that $d(N_i, \vec{N'}) > d(M_i, \vec{N'})$. Since for $a > 0$ we generally have $a \cdot k > a \cdot j$ if $k > j$, it follows that $p_j(\vec{N}) = a \cdot d(N_i, \vec{N}) > a \cdot d(M_i, \vec{N'}) = p_j(\vec{N'})$ as required. □

Of course, there is nothing that stops us from defining higher-order polynomials based on other well-founded sets $\mathcal{A}$ as well, provided the set A admits weakly monotonic operators for addition and multiplication: Lemma 4.19 does not use any special features of $\mathbb{N}$. Theorem 4.20 does, however (for example the property that $a \cdot k > a \cdot j$ if $a > 0$). To use polynomials for e.g. matrix interpretations or interpretations in the rational numbers, we would need to derive similar results for the different well-founded set.

Example 4.21. Recall the "functional map" AFSM from Example 2.21. After changing types, this system has the following form:

$$\mathcal{F} = \left\{ \begin{array}{rcl} \mathtt{nil} & : & \mathtt{o} \\ \mathtt{cons} & : & [\mathtt{o} \times \mathtt{o}] \longrightarrow \mathtt{o} \\ \mathtt{fnil} & : & \mathtt{o} \to \mathtt{o} \\ \mathtt{fcons} & : & [(\mathtt{o} \to \mathtt{o}) \times (\mathtt{o} \to \mathtt{o})] \longrightarrow \mathtt{o} \to \mathtt{o} \\ \mathtt{fmap} & : & [(\mathtt{o} \to \mathtt{o}) \times \mathtt{o}] \longrightarrow \mathtt{o} \end{array} \right\}$$

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathtt{fmap}(\mathtt{fnil}, Y) & \Rightarrow & \mathtt{nil} \\ \mathtt{fmap}(\mathtt{fcons}(F, X), Y) & \Rightarrow & \mathtt{cons}(F \cdot Y, \mathtt{fmap}(X, Y)) \end{array} \right\}$$

We consider a polynomial interpretations in the natural numbers:

$$\begin{array}{rcl}
\mathcal{J}(\mathtt{nil}) & = & 0 \\
\mathcal{J}(\mathtt{cons}) & = & \boldsymbol{\lambda} nm.n+m \\
\mathcal{J}(\mathtt{fnil}) & = & \boldsymbol{\lambda} n.0 \\
\mathcal{J}(\mathtt{fcons}) & = & \boldsymbol{\lambda} fgn.1+n+f(n)+g(n) \\
\mathcal{J}(\mathtt{fmap}) & = & \boldsymbol{\lambda} fn.f(n)+n \\
\mathcal{J}(@^{\sigma\to\tau}) & = & \boldsymbol{\lambda} Fn\vec{m}.F(n,\vec{m})+n(\vec{0}) \text{ for all types } \sigma,\tau
\end{array}$$

By Theorem 4.20, the interpretation of each function symbol is strongly monotonic in all arguments required by the symbol's arity. Thus we have a strong reduction pair by Theorem 4.16.

Let $\alpha = \{F \mapsto f,\ X \mapsto g,\ Y \mapsto n\}$. We have:

$$\begin{array}{rcl}
[\![\mathtt{fmap}(\mathtt{fnil},Y)]\!]_{\mathcal{J},\alpha} & = & 0+n \\
 & \geq & 0 \\
 & = & [\![\mathtt{nil}]\!]_{\mathcal{J},\alpha} \\
[\![\mathtt{fmap}(\mathtt{fcons}(F,X),Y)]\!]_{\mathcal{J},\alpha} & = & 1+2\cdot n+f(n)+g(n) \\
 & > & 2\cdot n+f(n)+g(n) \\
 & = & \mathtt{cons}(F\cdot Y,\mathtt{fmap}(X,Y))
\end{array}$$

Using rule removal (Theorem 2.23), we can remove the second rule; the system is terminating if the first rule on its own is terminating. But this is quickly demonstrated by a second polynomial interpretation, with $\mathcal{J}(\mathtt{fmap}) = \boldsymbol{\lambda} fn.f(0)+n+1$ and $\mathcal{J}(\mathtt{nil}) = 0$.

4.4 Overview

In this chapter we have considered the termination method using weakly monotonic algebras, introduced by van de Pol in [104], and extended these results to the setting of AFSMs. Some definitions were generalised (in particular the use of a separate $\geq$ relation in the basis of $\mathcal{WM}$), others restricted or simplified (in particular the choice for *strongly monotonic* functionals rather than *strict* ones).

More than just transposing the technique from [104] (which we could have done with Theorem 3.7), we have seen how weakly monotonic algebras can be used to create either a weak or a strong reduction pair. Thus, we can use algebra interpretations to prove termination either directly (with a reduction ordering), or using rule removal or dependency pairs.

Furthermore, we have discussed the class of higher-order polynomials in $\mathbb{N}$, which provides an easy and systematic way to find both weakly and strongly monotonic functionals. Higher-order polynomials form an elegant method for proving termination by hand and, as we will see in Chapter 8.5, a feasible automatable technique as well. This class could easily be extended to other base sets with addition and multiplication operators – matrix [35], integer [40, 49], rational [96] or arctic [82] interpretations, the sky is the limit!

An Iterative Path Ordering

5

Or, Can we do this in small steps?

In the previous chapter we have seen how the termination method of *monotonic algebras* can be lifted to the higher-order case. Here we shall consider an extension of two methods which are closely related to each other: the *recursive path ordering* (RPO) and the *iterative path ordering* (IPO).

The termination method of recursive path orderings was first defined by Dershowitz [30], and improved for example in [27, 36, 66]. The starting point of this method is a well-founded ordering (also called a precedence) on the function symbols of a TRS, which is lifted to a reduction ordering $\succ$ on terms. In the context of termination of higher-order rewriting, Jouannaud and Rubio present a higher-order extension HORPO of RPO in [63], a definition which is formalised in [81] and extended to HRSs and CRSs with an arity restriction in [106]. Incarnating ideas from the general schema [18], this definition has been strengthened with for instance a *computability closure* and *type orderings* in the extended version of the paper [64]. A later version, the *Computability Path Ordering* (CPO) [19] simplifies the definition and adds several new features.

In the first-order setting, Klop, van Oostrom and de Vrijer [73] present, following an approach originally due to Bergstra and Klop [12], the *iterative lexicographic path ordering* (ILPO) by means of an auxiliary term rewriting system. ILPO can be understood as an iterative definition of the lexicographic path ordering, a variant of RPO [66]. The authors show that ILPO is well-founded, and that it coincides with the lexicographic path ordering if the underlying relation on function symbols is transitive. Although ILPO seems less suitable for automation than RPO, it is an elegant and simple technique, useful for hand-written termination proofs, where a TRS is proved terminating by showing its inclusion in a terminating TRS. ILPO can be further extended to also include a multiset ordering, which was explored (in different ways) in [72] and [77].

In [76] Femke van Raamsdonk and I present an iterative variation of the original HORPO. Going further, however, presents new challenges, since both the extended HORPO from [64] and CPO relate terms of different types. Pursuing this path anyway, and extending the iterative path ordering to the AFSM setting

beyond the most basic steps (yet not necessarily allowing everything that is possible in CPO) leads to something new altogether: a simple yet powerful reduction pair on terms, which is weaker than the existing definitions of HORPO and CPO in some ways but, due to a more fine-grained approach, native transitivity and freedom of choice with regards to application, stronger in others.

Chapter Setup. In Section 5.1 we will discuss the existing definitions of recursive and iterative path orderings, both first-order and higher-order variations.

The *higher-order iterative path ordering* (HOIPO), a higher-order rewriting system defined as an application-free AFSM, is introduced in Section 5.2, and proved terminating in Section 5.3.

Since automation is an important part of this work, and an iterative definition without reduction strategy is not easily automatable, we will then study a recursive definition of the same ordering relation in Section 5.4.

Moving away from application-free systems, Section 5.5 discusses how the results can be used to define a strong reduction pair on AFSMs, without the limitation to application-free meta-terms. In Section 5.6 we will see how to strengthen the method using *argument functions*. Finally, in Section 5.7 we will compare the respective power of HOIPO and the existing *computability path ordering*, CPO.

This chapter extends [76], where an iterative variation of the first definition of HORPO is defined.

5.1 Existing Path Orderings

We will start by studying the existing definitions of (recursive or iterative) path orderings. The definitions of path orderings considered here, both first- and higher-order, share a number of features. Let us introduce these, first.

The Multiset Extension. A multiset is a set which might contain duplicates, and is denoted $\{\{s_1, \ldots, s_n\}\}$. The *multiset extension* of a given relation is defined as follows: $\{\{s_1, \ldots, s_n\}\} \succ_{Mul} \{\{t_1, \ldots, t_m\}\}$ if we can write $\{1, \ldots, n\} = A \cup B$ with B non-empty, and there is a total function π mapping $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$, such that:

- if $\pi(i) \in A$, then $s_{\pi(i)} = t_i$;
- if $\pi(i) \in B$, then $s_{\pi(i)} \succ t_i$;
- for every element j of A there is exactly one i such that $\pi(i) = j$.

The multiset extension of a well-founded ordering is also itself well-founded.

Example 5.1. Let $>$ be the standard greater than operator on natural numbers. Then $\{\{3, 5\}\} >_{Mul} \{\{3\}\} >_{Mul} \{\{2, 2, 2\}\} >_{Mul} \{\{0, 2, 1\}\} >_{Mul} \ldots$

Sometimes this definition does not suffice, in particular when a *pair* of relations is considered rather than a single relation $\succ$. The other relation could for instance be an equivalence relation which is not just equality, or a compatible quasi-ordering. Following [120] we define the *generalised multiset extension* of a pair $(\succsim, \succ)$ of relations as follows: $\{\{s_1, \ldots, s_n\}\} \succ_{gMul} \{\{t_1, \ldots, t_m\}\}$ if we can write $\{1, \ldots, n\} = A \cup B$ with B non-empty, and there is a total function π mapping $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$, such that:

- if $\pi(i) \in A$, then $s_{\pi(i)} \succsim t_i$;
- if $\pi(i) \in B$, then $s_{\pi(i)} \succ t_i$;
- for every element j of A there is exactly one i such that $\pi(i) = j$.

In addition, $\{\{s_1, \ldots, s_n\}\} \succsim_{gMul} \{\{t_1, \ldots, t_m\}\}$ if $n = m$ and there is a permutation π of $\{1, \ldots, n\}$ such that each $s_{\pi(i)} \succsim t_i$.

Thus, the normal multiset extension of $\succ$ is the generalised multiset extension of $(=, \succ)$. Equivalently, the normal multiset extension of $\succ$ can be seen as the generalised multiset extension of $(\succsim, \succ)$, where $\succsim$ is the reflexive closure of $\succ$.

The authors of [120] show that if $\succ$ is well-founded and $\succsim$ is compatible with $\succ$ and transitive, then the generalised multiset extension of $(\succsim, \succ)$ is also well-founded. However, we will not need this result. We will use the generalised multiset extension in the recursive version of StarHorpo, in Section 5.4.

The Lexicographic Extension. The *lexicographic extension* of a given relation is defined as follows: $[s_1, \ldots, s_n] \succ_{Lex} [t_1, \ldots, t_m]$ if there is some i such that $s_1 = t_1, \ldots, s_i = t_i$ and either $i = m < n$ or $s_{i+1} \succ t_{i+1}$. It is well-known that the lexicographic extension of a well-founded relation is again a well-founded relation, provided the domain is restricted to sequences of a bounded length.

Example 5.2. Let $>$ be the standard greater than operator on natural numbers. Then $[3, 5] >_{Lex} [3] >_{Lex} [2] >_{Lex} [0, 37, 42] >_{Lex} \ldots$ The bounded length restriction is needed for well-foundedness because we *do* have, e.g. $[1] >_{Lex} [0, 1] >_{Lex} [0, 0, 1] >_{Lex} \ldots$

As with the multiset extension, we might define the *generalised lexicographic extension* of a pair $(\succsim, \succ)$ as follows: $[s_1, \ldots, s_n] \succ_{gLex} [t_1, \ldots, t_m]$ if there is some i such that $s_1 \succsim t_1, \ldots, s_i \succsim t_i$ and either $i = m < n$ or $s_{i+1} \succ t_{i+1}$. In addition, $[s_1, \ldots, s_n] \succsim_{gLex} [t_1, \ldots, t_m]$ if $n = m$ and each $s_i \succsim t_i$.

We will use this extension in Section 5.4. As with the generalised multiset extension, the extra g will typically be omitted.

Precedence. All versions of the recursive and iterative path ordering are based on some precedence which is lifted to a reduction ordering or reduction pair. Following modern approaches we will assume given a precedence $\blacktriangleright$, which is a *quasi-ordering* on the set of function symbols. Its strict part, $\blacktriangleright \setminus \blacktriangleleft$, commonly

denoted as $\blacktriangleright$, should be a well-founded ordering. Let $\approx$ denote the relation $\underline{\blacktriangleright} \cap \underline{\blacktriangleleft}$; it is not hard to see that this is an equivalence relation. Arities are irrelevant; we may have $f \underline{\blacktriangleright} g$ and even $f \approx g$ regardless of the arities of f and g.

In early versions of the recursive path ordering, $\underline{\blacktriangleright}$ was assumed to be anti-symmetric: if $s \approx t$ then $s = t$. In later definitions, there are no restrictions on $\underline{\blacktriangleright}$ other than well-foundedness of $\blacktriangleright$.

Status. In modern approaches of the recursive path ordering, all symbols are equipped with a *status* in $\{Lex, Mul\}$. This status must respect $\approx$, so if $f \approx g$, then $stat(f) = stat(g)$. Moreover, if $stat(f) \in Lex$, then the arity of equivalent function symbols must be bounded. That is, there is some N such that for all g with $f \approx g$ the arity of g is at most N.

5.1.1 The Recursive Path Ordering (RPO)

There are many definitions of the *recursive path ordering* for first-order TRSs in the literature, as the original definition has been gradually refined and improved. The definition we shall consider here roughly corresponds to [27]. Only one feature (the use of permutations for symbols with a lexicographic status) is omitted here, as we will consider this as an addition afterwards (see Section 5.6).

Definition 5.3 (Recursive Path Ordering)**.** The recursive path ordering from [27] is built from two ingredients: a precedence $\underline{\blacktriangleright}$, and a status for all function symbols (which respects $\approx$, the equivalence relation induced by $\underline{\blacktriangleright}$). First, $\approx$ and the status function together induce an equivalence relation $\sim$ on terms:

- $x \sim y$ if $x = y$ are variables;
- $f(s_1, \ldots, s_n) \sim g(t_1, \ldots, t_n)$ if $f \approx g$ and:
  - if $stat(f) = Lex$, then $s_i \sim t_i$ for all i;
  - if $stat(f) = Mul$, then there is a permutation π of $\{1, \ldots, n\}$ such that always $s_{\pi(i)} \sim t_i$.

The recursive path ordering $\succ$, and the compatible quasi-ordering $\succsim = \succ \cup \sim$, are defined by the following recursive clauses: $s = f(s_1, \ldots, s_n) \succ t$ if

1. some $s_i \succsim t$, *or*
2. $t = g(t_1, \ldots, t_m)$ with $f \blacktriangleright g$ and $s \succ t_i$ for all i, *or*
3. $t = g(t_1, \ldots, t_m)$ with $f \approx g$ and:
   a) $stat(f) = Lex$, $s \succ t_i$ for all i, and $[s_1, \ldots, s_n] \succ_{gLex} [t_1, \ldots, t_m]$, *or*
   b) $stat(f) = Mul$, and $\{\{s_1, \ldots, s_n\}\} \succ_{gMul} \{\{t_1, \ldots, t_m\}\}$

Here, $\succ_{gLex}$ and $\succ_{gMul}$ are the lexicographic and multiset extensions of $(\sim, \succ)$.

Example 5.4. Consider the following TRS, which is contrived but demonstrates all features of RPO in one go:

$$\mathtt{f}(\mathtt{g}(\mathtt{s}(x),y),\mathtt{1},z) \quad \Rightarrow \quad \mathtt{f}(\mathtt{g}(y,\mathtt{s}(x)),\mathtt{0},\mathtt{h}(\mathtt{g}(x,x)))$$

Let $\mathtt{f} \approx \mathtt{h} \blacktriangleright \mathtt{1} \blacktriangleright \mathtt{0} \approx \mathtt{g} \approx \mathtt{s}$. Suppose $stat(\mathtt{f}) = stat(\mathtt{h}) = Lex$, and the status for the other symbols is Mul. Then we have the following reasoning:

1. $\mathtt{f}(\mathtt{g}(\mathtt{s}(x),y),\mathtt{1},z) \succ \mathtt{f}(\mathtt{g}(y,\mathtt{s}(x)),\mathtt{0},\mathtt{h}(\mathtt{g}(x,x)))$ using clause (3a) because:
   - $\mathtt{f} \approx \mathtt{f}$,
   - the left-hand side $\succ$ each part of the right-hand side (2,3,4),
   - $[\mathtt{g}(\mathtt{s}(x),y),\mathtt{1},z] \succ_{lex} [\mathtt{g}(y,\mathtt{s}(x)),\mathtt{0},\mathtt{h}(\mathtt{g}(x,x))]$ by 5 and 7.
2. $\mathtt{f}(\mathtt{g}(\mathtt{s}(x),y),\mathtt{1},z) \succ \mathtt{g}(y,\mathtt{s}(x))$ using clause (1) and 5.
3. $\mathtt{f}(\mathtt{g}(\mathtt{s}(x),y),\mathtt{1},z) \succ \mathtt{0}$ using clause (2).
4. $\mathtt{f}(\mathtt{g}(\mathtt{s}(x),y),\mathtt{1},z) \succ \mathtt{h}(\mathtt{g}(x,x))$ using clause (3a) and 6.
5. $\mathtt{g}(\mathtt{s}(x),y) \sim \mathtt{g}(y,\mathtt{s}(x))$ using the permutation $\pi = \{1 \mapsto 2, 2 \mapsto 1\}$.
6. $\mathtt{g}(\mathtt{s}(x),y) \succ \mathtt{g}(x,x)$ using clause (3b) and 8.
7. $\mathtt{1} \succ \mathtt{0}$ using clause (2).
8. $\mathtt{s}(x) \succ x$ by clause (1).

It can be demonstrated that $\succ$ is well-founded, and that both $\succ$ and $\succsim$ are monotonic and stable. Obviously, $\succsim$ and $\succ$ are compatible. Thus, $(\succsim, \succ)$ is a strong reduction pair (and $\succ$ on its own is a reduction ordering).

Variations. As stated before, there are many variations of the recursive path ordering. In particular, the definitions of ILPO, HORPO and CPO in this section are based on a definition of RPO which does not include an equivalence relation $\sim$. There, $\succsim$ is just the reflexive closure of $\succ$. This is quite relevant: in the derivation given above, for example, it is essential that $\mathtt{g}(\mathtt{s}(x),y) \sim \mathtt{g}(y,\mathtt{s}(x))$.

The use of a quasi-ordering for the precedence has also not always been the standard; traditional approaches assume that $\trianglerighteq$ is just the reflexive closure of $\blacktriangleright$. Nor do all definitions use a status. For example, the *lexicographic path ordering* (LPO) (on which ILPO is based) is the restriction of RPO where $stat(f) = Lex$ for all function symbols f, and where $f \trianglerighteq g$ if and only if either $f \blacktriangleright g$ or $f = g$.

5.1.2 The Iterative Lexicographic Path Ordering (ILPO)

The *iterative lexicographic path ordering* from [73], ILPO, defines essentially the same relation as the lexicographic path ordering, but gives this relation as a term rewriting system. This results in a reduction ordering $\Rightarrow_\star^+$, or a reduction pair $(\Rightarrow_\star^*, \Rightarrow_\star^+)$. Let $\mathcal{F}^\star := \mathcal{F} \cup \{f^\star \mid f \in \mathcal{F}\}$, where $f^\star$ has the same arity as f, and let $\mathcal{R}^\star$ consist of the following rules, for all symbols f, g:

$$\begin{array}{rll} f(x_1,\ldots,x_n) & \Rightarrow_{\texttt{put}} & f^\star(x_1,\ldots,x_n) \\ f^\star(x_1,\ldots,x_n) & \Rightarrow_{\texttt{select}} & x_i \\ f^\star(x_1,\ldots,x_n) & \Rightarrow_{\texttt{copy}} & g(f^\star(\vec{x}),\ldots,f^\star(\vec{x})) \text{ if } f \blacktriangleright g \\ l^\star := f^\star(x_1,\ldots,g(\vec{y}),\ldots,x_n) & \Rightarrow_{\texttt{lex}} & f(x_1,\ldots,g^\star(\vec{y}),l^\star,\ldots,l^\star) \end{array}$$

The clauses above are *rule schemes*: clauses which generate a number of rules in one go. Mostly, we will simply refer to them as rules, since no ambiguity can arise from speaking of, for instance, the `put` rule. We also write $\Rightarrow_{\mathrm{ILPO}}$ for $\Rightarrow_{\mathcal{R}^\star}$.

The TRS $(\mathcal{F}^\star, \mathcal{R}^\star)$ is evidently not terminating. For example, in (some instances of) the `copy` rule, the right-hand side contains the left-hand side! But it *can* be proved that there is no reduction in $\Rightarrow_{\mathrm{ILPO}}$ which contains infinitely many star-free terms.

Thus, for a given set of rules $\mathcal{R}$, we can show termination of the rewrite relation $\Rightarrow_{\mathcal{R}}$ if it is included in the relation $\Rightarrow_{\mathrm{ILPO}}^+$ (which is terminating on terms over $\mathcal{F}$). This we prove by showing that $l \Rightarrow_{\mathrm{ILPO}}^+ r$ for all rules $l \Rightarrow r \in \mathcal{R}$. More generally, $\Rightarrow_{\mathrm{ILPO}}^+$ is a reduction ordering on terms over $\mathcal{F}$.

Example 5.5. The constraint $\texttt{add}(\texttt{s}(x), y) \succ \texttt{add}(x, \texttt{s}(y))$ is oriented by ILPO if we choose $\texttt{add} \blacktriangleright \texttt{s}$:

$$\begin{array}{ll} & \texttt{add}(\texttt{s}(x), y) \\ \Rightarrow_{\texttt{put}} & \texttt{add}^\star(\texttt{s}(x), y) \\ \Rightarrow_{\texttt{lex}} & \texttt{add}(\texttt{s}^\star(x), \texttt{add}^\star(\texttt{s}(x), y)) \\ \Rightarrow_{\texttt{copy}} & \texttt{add}(\texttt{s}^\star(x), \texttt{s}(\texttt{add}^\star(\texttt{s}(x), y))) \\ \Rightarrow_{\texttt{select}} & \texttt{add}(\texttt{s}^\star(x), \texttt{s}(y)) \\ \Rightarrow_{\texttt{select}} & \texttt{add}(x, \texttt{s}(y)) \end{array}$$

Intuitively, the star denotes an intention to make the term under consideration smaller. The restriction of $\Rightarrow_{\mathrm{ILPO}}^+$ to terms over $\mathcal{F}$ defines the same relation as $\succ_{\mathrm{LPO}}$ (the restriction of RPO to the case where $stat(f)$ is always Lex and $\blacktriangleright$ is the reflexive closure of $\blacktriangleright$).

Why ILPO? There are several reasons to consider ILPO over LPO. First, there is a certain elegance to proving termination by showing the inclusion of the rewrite relation in another rewrite relation, which is known to be terminating. Second, the definition of ILPO as a TRS avoids a number of problems in the definition of LPO (which are essentially ignored here, and in most sources). For example, LPO uses the lexicographic extension of a relation which is not yet fully defined.

Another reason is that derivations using ILPO and LPO have a very different form. Where derivations with LPO use proof obligations ($\mathtt{add}(\mathtt{s}(x), y) \succ \mathtt{add}(x, \mathtt{s}(y))$ because (1) $\mathtt{s}(x) \succ x$ and (2) $\mathtt{add}(\mathtt{s}(x), y) \succ \mathtt{s}(y)$), derivations using ILPO take small reduction steps. Often, the decision how exactly to make the term smaller is postponed, and there may be many marked symbols in a term (corresponding to the proof obligations in the LPO case). Depending on your flavour, this style may be preferable for (in particular hand-written) termination proofs, especially using a top-down goal-driven reduction strategy.

In the higher-order case, a further reason appears: $f^\star(\vec{s})$ is the largest term that is "smaller" than $f(\vec{s})$. There is no counterpart of such a term in LPO. Having a term like this is exceedingly useful when, due to β-reduction, a term is copied many times. We will discuss this in more detail at the end of Section 5.4.

Extending ILPO with Multisets. To extend the definition given above with a multiset status, and perhaps get a counterpart of the complete RPO, there are several ways we might go. In [77] Femke van Raamsdonk and I propose an additional two rule schemes:

$$\begin{array}{rcl} f^\star(\vec{x}, g(\vec{y}), \vec{z}) & \Rightarrow_{\mathtt{mul}} & f([x_1|g^\star(\vec{y})], \ldots, g^\star(\vec{y}), \ldots, [z_n|g^\star(\vec{y})]) \text{ if } stat(f) = Mul \\ f^\star(x_1, \ldots, x_n) & \Rightarrow_{\mathtt{ord}} & f^\star(x_{\pi(1)}, \ldots, x_{\pi(n)}) \text{ for some permutation } \pi \end{array}$$

Here $[a|b]$ means either a or b can be chosen, and we may make different choices at different places in the term. The *lex* rule must additionally be updated with a $stat(f) = Lex$ restriction. The relation $\Rightarrow_{\mathrm{IPO}}$ generated by these rules in addition to the ILPO-rules corresponds to a version of RPO where no equivalence relation is used, and where $\approx$ is just equality, but where the status is not fixed.

An alternative way to extend ILPO with a multiset status is (tentatively) explored in [72] (an unpublished variation of [73]). Here, the solution is to alter the definition of terms a bit, to take the status into account:

- every variable is a term;
- if $f \in \mathcal{F}$, and f has arity n, and $s_1, \ldots, s_n$ are terms, then:
  - if $stat(f) = Lex$, then $f(s_1, \ldots, s_n)$ is a term;
  - if $stat(f) = Mul$, then $f(\{\{s_1, \ldots, s_n\}\})$ is a term.

In this definition, $f(\{\{s_1, \ldots, s_n\}\}) = f(\{\{t_1, \ldots, t_n\}\})$ if the *multisets* $\{\{s_1, \ldots, s_n\}\}$ and $\{\{t_1, \ldots, t_n\}\}$ are equal. The TRS $(\mathcal{F}^\star, \mathcal{R}^\star)$ is extended with a single new rule scheme:

$$f^\star(\{\{\vec{x}, g_1(\vec{y_1}), \ldots, g_n(\vec{y_n})\}\}) \Rightarrow_{\mathtt{mst}} f(\{\{\vec{x}, g^\star_{i_1}(\vec{y_{i_1}}), \ldots, g^\star_{i_n}(\vec{y_{i_n}})\}\})$$

Example 5.6. Consider the TRS from Example 5.4, but now with a strict symbol precedence: $\mathtt{f} \blacktriangleright \mathtt{h} \blacktriangleright \mathtt{1} \blacktriangleright \mathtt{0} \blacktriangleright \mathtt{g} \blacktriangleright \mathtt{s}$.

$$\mathtt{f}(\mathtt{g}(\mathtt{s}(x), y), \mathtt{1}, z) \qu\Rightarrow \quad \mathtt{f}(\mathtt{g}(y, \mathtt{s}(x)), \mathtt{0}, \mathtt{h}(\mathtt{g}(x, x)))$$

Suppose $stat(\mathtt{g}) = Mul$, and the status for the other symbols is Lex. Then we have the following reduction:

$$
\begin{array}{ll}
& \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}, z) \\
\Rightarrow_{\mathtt{put}} & \mathtt{f}^\star(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}, z) \\
\Rightarrow_{\mathtt{lex}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}^\star, \mathtt{f}^\star(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}, z)) \\
\Rightarrow_{\mathtt{copy}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}^\star, \mathtt{h}(\mathtt{f}^\star(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}, z))) \\
\Rightarrow_{\mathtt{select}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{1}^\star, \mathtt{h}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}))) \\
\Rightarrow_{\mathtt{copy}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{0}, \mathtt{h}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}))) \\
\Rightarrow_{\mathtt{put}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{0}, \mathtt{h}(\mathtt{g}^\star(\{\{\mathtt{s}(x), y\}\}))) \\
\Rightarrow_{\mathtt{mst}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{0}, \mathtt{h}(\mathtt{g}(\{\{\mathtt{s}^\star(x), \mathtt{s}^\star(x)\}\}))) \\
\Rightarrow^+_{\mathtt{select}} & \mathtt{f}(\mathtt{g}(\{\{\mathtt{s}(x), y\}\}), \mathtt{0}, \mathtt{h}(\mathtt{g}(\{\{x, x\}\}))) \\
= & \mathtt{f}(\mathtt{g}(\{\{y, \mathtt{s}(x)\}\}), \mathtt{0}, \mathtt{h}(\mathtt{g}(\{\{x, x\}\})))
\end{array}
$$

In the higher-order iterative path ordering, we shall not use either approach. Rather, we consider a third technique, which is more suitable for a precedence where $\approx$ is not just equality, and which does not require altering the definition of terms.

5.1.3 The Higher-Order Recursive Path Ordering (HORPO)

The definition of the *higher-order recursive path ordering* [63] considers *Algebraic Functional Systems* (see Chapter 3.4), and therefore has no cases for meta-variables. The reduction ordering is given by a number of inductive clauses, using $\succ$, $\succsim$ and $\succ\!\!\!\succ$. The latter two relations are explained below; for terms[1] s, t we have $s \succ t$ if both sides have the same type and:

(H1)	$s = f(s_1, \ldots, s_m)$ and $s_i \succsim t$ for some $1 \leq i \leq m$
(H2)	$s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, $f \blacktriangleright g$ and $s \succ\!\!\!\succ \{t_1, \ldots, t_n\}$
(H3LEX)	$s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $stat(f) = Lex$, $[s_1, \ldots, s_m] \succ_{Lex} [t_1, \ldots, t_m]$ and $s \succ\!\!\!\succ \{t_1, \ldots, t_m\}$
(H3MUL)	$s = f(s_1, \ldots, s_m)$, $t = f(t_1, \ldots, t_m)$, $stat(f) = Mul$ and $\{\{s_1, \ldots, s_m\}\} \succ_{Mul} \{\{t_1, \ldots, t_m\}\}$
(H4)	$s = f(s_1, \ldots, s_m)$, $t = t_1 \cdot t_2 \cdots t_n$ and $s \succ\!\!\!\succ \{t_1, \ldots, t_n\}$ $(n \geq 2)$
(H5)	$s = s_1 \cdot s_2$, $t = t_1 \cdot t_2$ and $\{\{s_1, s_2\}\} \succ_{Mul} \{\{t_1, t_2\}\}$
(H6)	$s = \lambda x.q$, $t = \lambda x.u$ and $q \succ u$

Here $\succsim$ denotes the reflexive closure of $\succ$. Further, following the notation from [81], the relation $\succ\!\!\!\succ$ between a functional term and a set of terms is defined as follows: $s = f(s_1, \ldots, s_m) \succ\!\!\!\succ \{t_1, \ldots, t_n\}$ if for every $i \in \{1, \ldots, n\}$ we have either $s \succ t_i$, or there exists $j \in \{1, \ldots, m\}$ such that $s_j \succsim t_i$.

The first four clauses of the definition stem directly from the first-order definition of RPO, with the difference that instead of the requirement $s \succ\!\!\!\succ \{t_1, \ldots, t_n\}$ for

[1] Technically HORPO is an ordering on AFS-terms, but the definition of terms in an AFS is exactly the same as in an AFSM. Thus, we can just speak of "terms" here without ambiguity.

HORPO, we have for RPO the simpler $s \succ t_i$ for every i. This is not feasible for the higher-order case because of the type requirements; the relation $\succ$ is only defined on terms of equal type

The type restriction cannot be avoided entirely: without it, we would have, for example, $\mathtt{A}(\mathtt{L}(F), X) \succ F \cdot X$, which leads to an infinite decreasing sequence $\mathtt{A}(\mathtt{L}(\lambda x.\mathtt{A}(x,x)), \mathtt{L}(\lambda x.\mathtt{A}(x,x))) \succ (\lambda x.\mathtt{A}(x,x)) \cdot \mathtt{L}(\lambda x.\mathtt{A}(x,x)) \succ \mathtt{A}(\mathtt{L}(\lambda x.\mathtt{A}(x,x)), \mathtt{L}(\lambda x.\mathtt{A}(x,x)))$. However, it can be weakened; for instance, the definition in [63] equates all base types (which we might do with the type changing functions from Theorem 2.29). Later definitions of HORPO use a type ordering, as we will see in the next section.

Example 5.7. Consider the following definition of the recursor for the natural numbers:

$$\mathcal{F} = \left\{ \begin{array}{rcl} \mathtt{0} & : & \mathtt{nat} \\ \mathtt{s} & : & [\mathtt{nat}] \longrightarrow \mathtt{nat} \\ \mathtt{rec} & : & [\mathtt{nat} \times \mathtt{nat} \times (\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat})] \longrightarrow \mathtt{nat} \end{array} \right\}$$

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathtt{rec}(\mathtt{0}, Y, F) & \Rightarrow & Y \\ \mathtt{rec}(\mathtt{s}(X), Y, F) & \Rightarrow & F \cdot X \cdot \mathtt{rec}(X, Y, F) \end{array} \right\}$$

For the first rewrite rule, we have $\mathtt{rec}(\mathtt{0}, Y, F) \succ Y$ by (H1). For the second:

1. $\mathtt{rec}(\mathtt{s}(X), Y, F) \succ F \cdot X \cdot \mathtt{rec}(X, Y, F)$ by (H4) and 2, 3 and 4.
2. $F \succsim F$ by reflexivity of $\succsim$.
3. $\mathtt{s}(X) \succ X$ by clause (H1).
4. $\mathtt{rec}(\mathtt{s}(X), Y, F) \succ \mathtt{rec}(X, Y, F)$ by choosing $stat(\mathtt{rec}) = Mul$ and using again 2, 3 and also that $Y \succsim Y$ by reflexivity.

HORPO as presented here has some important weaknesses. Although it can be used to show termination of many standard examples, such as `rec` and `map`, the type restrictions often limit the possibilities. For example, assuming that `f` has a base type as output type, we do not have that $\mathtt{f}(\mathtt{1}, F) \succ \mathtt{f}(\mathtt{0}, \lambda x.\mathtt{g}(x))$ even if $\mathtt{1} \blacktriangleright \mathtt{0}$ and $\mathtt{f} \blacktriangleright \mathtt{g}$: this is because $\mathtt{f}(\mathtt{1}, F) \succcurlyeq\!\!\!\succ \{\lambda x.\mathtt{g}(x)\}$ does not hold for type reasons. The authors of [63] combat this weakness by introducing the *computability closure*, a possibly infinite set of terms $CC(f(\vec{s}))$, such that the clause "$s \succcurlyeq\!\!\!\succ \{t_1, \ldots, t_n\}$" everywhere in the definition of HORPO can be replaced by "each $t_i \in CC(s)$". Using a computability closure, the example constraint $\mathtt{f}(\mathtt{1}, F) \succ \mathtt{f}(\mathtt{0}, \lambda x.\mathtt{g}(x))$ can be oriented, since $\lambda x.\mathtt{g}(x) \in CC(f(\mathtt{1}, F))$.

Here, we shall not consider the computability closure. Instead we look at a recent extension of HORPO, which removes the need for a computability closure.

5.1.4 The Computability Path Ordering (CPO)

The *computability path ordering* from [19] is restricted to AFSs where all function symbols have a base type as output type. The method uses a *type ordering* in addition to the precedence and status on the function symbols. This is a quasi-ordering $\geq_T$ with strict part $>_T = \geq_T \setminus \leq_T$, which satisfies the following requirements for all σ, τ, ρ:

well-foundedness $>_T \cup \rhd_\to$ is well-founded
here, $\rhd_\to \tau$ is given by: $\sigma \to \tau \rhd_\to \sigma$;

right arrow subterm $\sigma \to \tau >_T \tau$;

arrow preservation $\sigma \to \tau =_T \rho$ iff $\rho = \sigma' \to \tau'$ with $\sigma' =_T \sigma$ and $\tau' =_T \tau$;

arrow decreasingness if $\sigma \to \tau >_T \rho$, then either $\tau \geq_T \rho$ or else $\rho = \sigma' \to \tau'$ with $\sigma =_T \sigma'$ and $\tau >_T \tau'$.

Using this type ordering, the authors of [19] define an *accessibility* relation. This relation is based on the following definition: let ι denote a base type;

- $Positive(\sigma_1 \to \ldots \to \sigma_n \to \iota) = \{\iota\} \cup Negative(\sigma_1) \cup \ldots \cup Negative(\sigma_n)$;
- $Negative(\sigma_1 \to \ldots \to \sigma_n \to \iota) = Positive(\sigma_1) \cup \ldots \cup Positive(\sigma_n)$.

The set $Acc(f)$ of *accessible argument positions* of a function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \iota$ is the largest set of integers $i \in \{1, \ldots, n\}$ such that:

- no base type κ with $\kappa >_T \iota$ occurs in σ_i;
- no base type κ with $\kappa =_T \iota$ occurs in $Negative(\sigma_i)$.

The accessibility relation, $\rhd_{acc}$, is the smallest transitive relation such that for all function symbols f and all $i \in Acc(f)$: $f(s_1, \ldots, s_n) \rhd_{acc} s_i$. For example, if $\mathtt{ordrec} : [\mathtt{ord} \times \mathtt{a} \times \mathtt{ord} \to \mathtt{a} \to \mathtt{a} \times (\mathtt{nat} \to \mathtt{ord}) \to (\mathtt{nat} \to \mathtt{a}) \to \mathtt{a}] \longrightarrow \mathtt{a}$ and $\mathtt{lim} : [\mathtt{nat} \to \mathtt{ord}] \longrightarrow \mathtt{ord}$, then the meta-variable F is accessible in the meta-term $\mathtt{ordrec}(\mathtt{lim}(F), Y, G, H)$ provided $\mathtt{ord} >_T \mathtt{nat}$. The accessibility relation is typically used to access higher-order subterms which would cause problems[2] if all base types were equal.

The full reduction ordering CPO is an ordering on terms (technically AFS-terms, but terms in an AFS and in an AFSM use exactly the same definition) built over a signature which satisfies the restriction that all symbols have a base output type. It is given by the clauses below, where X is a set of variables, $\succsim^X$ is the reflexive closure of $\succ^X$, and for $s : \sigma$ and $t : \tau$, the notation $s \succsim^X_T t$ denotes that $s \succsim^X t$ and $\sigma \geq_T \tau$. Similarly, $s \succ^X_T t$ denotes that $s \succ^X t$ and $\sigma \geq_T \tau$. The set X will

[2] In particular, problems of the "unwittingly encoding the untyped λ-calculus" kind.

be used to keep track of the bound variables encountered "above" the right-hand side of the current constraint.

Unlike its predecessor, CPO uses a quasi-precedence where $\underline{\blacktriangleright}$ is not necessarily the reflexive closure of $\blacktriangleright$, so $\approx$ may be more than just equality.

1. $s = f(s_1, \ldots, s_n) \succ^X t$ if $f \in \mathcal{F}$ and at least one of:
   a) $t \in X$;
   b) $t = g(t_1, \ldots, t_m)$ with $f \approx g$, $s \succ^X t_1, \ldots, s \succ^X t_m$ and $[s_1, \ldots, s_n]\ (\succ_T^\emptyset \cup \sqsupset_{acc}^{X,s})_{stat(f)}\ [t_1, \ldots, t_m]$
   where $q : \sigma' \sqsupset_{acc}^{X,s} u : \tau'$ if the following requirements hold:
      - $\sigma' \geq_T \tau'$;
      - $u = v \cdot w_1 \cdots w_k$;
      - $q \rhd_{acc} v$;
      - $s \succ^X w_i$ for all w_i;
   c) $t = g(t_1, \ldots, t_m)$ with $f \blacktriangleright g$ and $s \succ^X t_i$ for all i;
   d) $t = t_0 \cdot t_1 \cdots t_m$ and $s \succ^X t_i$ for all i;
   e) $t = \lambda x.q$ and $s \succ^{X \cup \{x\}} q$;
   f) $s_i \succsim_T^\emptyset t$ for some i;
   g) $q \succsim_T^\emptyset t$ for some q such that $s_i \rhd_{acc} q$ for some i.
2. $s = q \cdot u \succ^X t$ if at least one of:
   a) $t \in X$;
   b) $t = v \cdot w$ and $\{\{q, u\}\}(\succ_T^\emptyset)_{mul}\{\{v, w\}\}$;
   c) $t = \lambda x.q$ and $s \succ^X q$
   d) $q \succsim_T^X t$ or $u \succsim_T^X t$;
   e) $q = \lambda x.v$ and $v[x := u] \succsim^X t$.
3. $s = \lambda x : \sigma.q \succ^X t$ if at least one of:
   a) $t \in X$;
   b) $q \succsim_T^X t$;
   c) $q = u \cdot x$ and $u \succsim^X t$;
   d) $t = \lambda y : \tau.u$ and $\sigma =_T \tau$ and $q[x := z] \succ^X u[y := z]$ for some fresh variable z;
   e) $t = \lambda y : \tau.u$ and $\sigma \neq_T \tau$ and $s \succ^X u$.

The authors demonstrate that $\succ_T^\emptyset$ is a reduction ordering. We might also denote this relation as $\succ_{\mathrm{CPO}}$.

Example 5.8. Consider *Brouwer's Ordinals*.

$$\begin{array}{rcl} \mathtt{0} & : & \mathtt{ord} \\ \mathtt{s} & : & [\mathtt{ord}] \longrightarrow \mathtt{ord} \\ \mathtt{lim} & : & [\mathtt{nat} \to \mathtt{ord}] \longrightarrow \mathtt{ord} \\ \mathtt{ordrec} & : & [\mathtt{ord} \times \mathtt{a} \times (\mathtt{ord} \to \mathtt{a} \to \mathtt{a}) \times ((\mathtt{nat} \to \mathtt{ord}) \to (\mathtt{nat} \to \mathtt{a}) \to \mathtt{a})] \longrightarrow \mathtt{a} \end{array}$$

$$\begin{array}{rcl} \mathtt{ordrec}(\mathtt{0}, Y, G, H) & \Rightarrow & Y \\ \mathtt{ordrec}(\mathtt{s}(X), Y, G, H) & \Rightarrow & G \cdot X \cdot \mathtt{ordrec}(X, Y, G, H) \\ \mathtt{ordrec}(\mathtt{lim}(F), Y, G, H) & \Rightarrow & H \cdot F \cdot \lambda x.\mathtt{ordrec}(F \cdot x, Y, G, H) \end{array}$$

The first rule is oriented with clause 1f. The second rule is oriented with clause 1d, using clauses 1b and 1f (multiple times) for the resulting constraints. Using a type ordering with $\mathtt{ord} >_T \mathtt{nat}$, the last rule is also oriented with clause 1d. This gives proof obligations $\mathtt{ordrec}(\mathtt{lim}(F), Y, G, H) \succ^{\emptyset} H$ (which holds by clause 1f), $\mathtt{ordrec}(\mathtt{lim}(F), Y, G, H) \succ^{\emptyset} F$ (which holds by clause 1g), and $\mathtt{ordrec}(\mathtt{lim}(F), Y, G, H) \succ^{\emptyset} \lambda x.\mathtt{ordrec}(F \cdot x, Y, G, H)$.

This last constraint holds by clause 1e and 1b which (omitting the constraints we can easily handle with 1f) yields the proof obligations $\mathtt{ordrec}(\mathtt{lim}(F), Y, G, H) \succ^{\{x\}} F \cdot x$ (which holds by clauses 1d, 1g and 1a), and $\mathtt{lim}(F) \sqsupseteq_{acc}^{\{x\},\mathtt{ordrec}(\ldots)} F \cdot x$ (which holds because $\mathtt{ordrec}(\ldots) \succ^{\{x\}} x$ by clause 1a).

Note that we could not deal with this rule using the original HORPO; we needed accessibility for the last rule.

Not including the type ordering and definition of accessibility, nor a possible equivalence relation that we might wish to add to follow the definition of RPO, or additional rules for meta-terms, this definition spans a full page. It can be simplified a bit, however: if $s \succ_T^{\emptyset} t$, then any $\succ$ clause in its derivation either has the form $q \succ^X u$ with q a functional term, or $q \succ_T^{\emptyset} u$. The proof proceeds by induction: we assume that $s : \sigma \succ^X t : \tau$ and either s is a functional term, or both $X = \emptyset$ and $\sigma \geq_T \tau$, and then prove that the same holds for all clauses used in its derivation.

Thus, we lose some cases, and do not always have to keep track of variables. Still, this definition is a good deal more involved than the first-order definition. This is inherent in the design: the ordering aims to be as powerful as possible.

The iterative path ordering which we will see next, and its recursive variant which we will study in Section 5.4, are not based on CPO. Rather, the starting point is ILPO; the resulting reduction pair has much in common with CPO, but differs in several important respects. As a side bonus, the definition of HOIPO also provides a new iterative path ordering for first-order systems, which corresponds to the full definition of RPO present in Section 5.1.1.

Figure 5.1 gives an overview of the systems discussed so far in their mutual context. For a more complete picture, "RPO without $\sim$" (that is, RPO where no equivalence relation is used, and where $\sim$ is just equality) is included, as this is

the version of RPO which HORPO is based on. A horizontal line in the diagram indicates an equivalence, a vertical or diagonal line indicates inclusion.

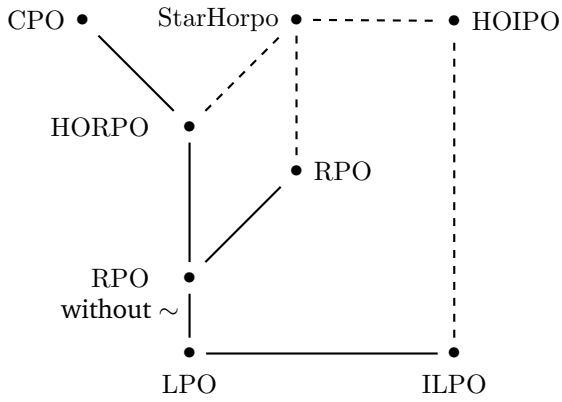


Figure 5.1: Iterative and Recursive Path Orderings – the solid lines represent existing results, the dashed lines represent the two new results presented in this chapter.

The plan for this chapter is given by the dashed lines: we will define an extension of the iterative lexicographic path ordering, and an equivalent recursive counterpart which builds on the recursive path ordering. This new relation will strictly extend the higher-order recursive path ordering but will *not* extend the computability path ordering. Rather, the definitions will be incomparable.

5.2 The Higher-Order Iterative Path Ordering (HOIPO)

To extend ILPO in as natural a way as possible to the higher-order case, we should ideally avoid special cases. One measure to simplify the definitions is to not consider application as a special symbol; this way we do not need a parallel to case 2 of CPO. Functionally, we avoid application by restricting the method to *application-free* AFSMs. That is, AFSMs $(\mathcal{F}, \mathcal{R})$ where no applications occur in either side of any rule, and where the rewrite relation is restricted to terms which do not contain applications.

We saw this as the *IDTS*-formalism from Chapter 3.2. As observed there, limiting interest to application-free AFSMs is not a real restriction, since every AFSM can be transformed into such a form. For example, a rule

$$\mathtt{map}(F, \mathtt{cons}(X, Y)) \Rightarrow \mathtt{cons}(F \cdot X, \mathtt{map}(F, Y))$$

is translated to

$$\mathtt{map}(F, \mathtt{cons}(X, Y)) \Rightarrow \mathtt{cons}(@^{\mathtt{nat}\to\mathtt{nat}}(F, X), \mathtt{map}(F, Y))$$

and additional rules

$$@^{\sigma\to\tau}(\lambda x.F(x), X) \Rightarrow F(X)$$

are added for all types σ, τ to simulate β-reduction. We will say some more about this in Section 5.5. For now, we will just limit interest to application-free meta-terms. Other than this, there are no restrictions. Unlike CPO, function symbols may have an arbitrary output type.

With an eye on the extensions of Chapter 6, and possibly other transformations, we will include one new "special case" feature: *minimal elements*.

In the application-free setup used in the coming sections, we will define a *strong reduction pair*, which, in the absence of β-reduction, does not need to include beta. That is, a pair $(\succsim, \succ)$ of a quasi-ordering and a compatible well-founded ordering, such that both relations are monotonic and meta-stable. In Section 5.5 we will use this reduction pair for application-free meta-terms to create a strong reduction pair for unrestricted AFSMs.

Towards defining an ILPO-like strong reduction pair, let $\mathcal{F} = \mathcal{F}_1 \uplus \{\bot_\sigma \mid \sigma \text{ a type}\}$ be a set of function symbols, and let $\mathcal{F}^\star$ be $\mathcal{F}$ extended with for every function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \alpha \in \mathcal{F}_1$, infinitely many new "marked" symbols $f^\star_{\tau_1,\ldots,\tau_m,\rho} : [\sigma_1 \times \ldots \times \sigma_n \times \tau_1 \times \ldots \tau_m] \longrightarrow \rho$. Such a symbol $f^\star_{\tau_1,\ldots,\tau_m,\rho}$ is added for all m, and all types $\tau_1, \ldots, \tau_m, \rho$.

The symbols $f^\star_{\vec{\tau},\rho}$ take possibly more arguments than f itself, and may have any output type. With this choice, we avoid certain limitations in what a term of the form $f^\star(\vec{s})$ can be reduced to. Note that no marked symbols are added for the special symbols $\bot_\sigma$. Types will commonly be omitted when they are clear from context. For a term $s = \lambda\vec{x}.f(\vec{s})$ with $f \in \mathcal{F}_1$, let $s^\star$ denote $\lambda\vec{x}.f^\star_\tau(\vec{s})$, where τ is the output type of f; $s^\star$ is undefined if s does not have this form.

Now we fix a precedence $\trianglerighteq$ on $\mathcal{F}_1$ (which does not need to satisfy any additional constraints; we may have a precedence where $\approx$ is not the equality), and choose a status function $stat$ which maps the symbols of $\mathcal{F}_1$ to $\{Lex, Mul\}$. The set of rules $\mathcal{R}^\star$ is generated by the rule schemes in Figure 5.2, each of which represents a finite or infinite number of rules. We will informally discuss these rule schemes afterwards.

Explicit type denotations for the $f^\star$ have been omitted; the rules preserve types, so for instance in the `put` rule, if $f : [\vec{\sigma}] \longrightarrow \tau$, the $f^\star$ should be read as $f^\star_\tau$. In the `select` rule, if the $f^\star$ on the left-hand side has a type denotation $\vec{\sigma}, \tau$ and $Z_i : [\rho_1 \times \ldots \times \rho_m] \longrightarrow \rho$, then the right-hand side should be read as $Z_i(f^\star_{\vec{\sigma},\rho_1}(\vec{Z}), \ldots, f^\star_{\vec{\sigma},\rho_m}(\vec{Z}))$, and so on. The legality of the last rule, labelled `bot`, is questionable, since we normally do not permit rules whose left-hand side is a meta-variable. However, this system is not expected to be terminating, and we can easily adapt the definition of IDTSs to allow such rules.

The *higher-order iterative path ordering* (HOIPO) is the relation $\Rightarrow^+_{\mathcal{R}^\star}$. This relation is also denoted $\Rightarrow_\star$.

Comparing these rule schemes (usually just called *rules*) to the definition of ILPO from Section 5.1.2 there are some observations to make:

$f(Z_1, \ldots, Z_n) \Rightarrow_{\texttt{put}} f^\star(Z_1, \ldots, Z_n)$
$f^\star(\vec{Z}) = f^\star(Z_1, \ldots, \lambda\vec{x}.Z_i(\vec{x}), \ldots, Z_n) \Rightarrow_{\texttt{select}} Z_i(f^\star(\vec{Z}), \ldots, f^\star(\vec{Z}))$
$f^\star(Z_1, \ldots, Z_n) \Rightarrow_{\texttt{copy}} g(f^\star(\vec{Z}), \ldots, f^\star(\vec{Z}))$ if $f \blacktriangleright g$
$f^\star(\vec{Z}) = f^\star(Z_1, \ldots, s_i, \ldots, Z_n) \Rightarrow_{\texttt{lex}} g(Z_1, \ldots, s_i^\star, f^\star(\vec{Z}), \ldots, f^\star(\vec{Z}))$
if $f \approx g,\ stat(f) = Lex,\ i \leq ar(f),\ i - 1 \leq ar(g)$
where either $i > ar(g)$ and $s_i = Z_i$,
or $ar(g) \geq i$ and $s_i = \lambda\vec{x}.h(Z_1'(\vec{x}), \ldots, Z_k'(\vec{x}))$ and $s_i^\star = \lambda\vec{x}.h^\star(\vec{Z}'(\vec{x}))$
note: if $i > ar(g)$ the right-hand side should be read: $g(Z_1, \ldots, Z_{i-1})$
$f^\star(s_1, \ldots, s_n, Z_{n+1}, \ldots, Z_m) \Rightarrow_{\texttt{mul}} g(t_1, \ldots, t_k)$
if $f \approx g,\ stat(f) = Mul,\ ar(f) = n$, and each s_i is either a unique meta-variable Z_i or has the form $\lambda\vec{x}.h(\vec{Z_i'}(\vec{x}))$; each t_i is some s_j if s_j is a meta-variable, or $s_j^\star$ otherwise; each s_j which is a meta-variable occurs at most once in $\{\{t_1, \ldots, t_k\}\}$. Moreover, $\{\{\vec{s}\}\} \neq \{\{\vec{t}\}\}$.
$f^\star_{\vec{\sigma},\tau\to\rho}(Z_1, \ldots, Z_n) \Rightarrow_{\texttt{abs}} \lambda x.f^\star_{\vec{\sigma},\tau,\rho}(Z_1, \ldots, Z_n, x)$

$f(Z_1, \ldots, Z_n) \Rightarrow_{\texttt{equiv}} g(Z_1, \ldots, Z_n)$
if $f \approx g,\ stat(f) = Lex$ and $ar(f) = ar(g) = n$
$f(Z_1, \ldots, Z_n) \Rightarrow_{\texttt{equiv}} g(Z_{\pi(1)}, \ldots, Z_{\pi(n)})$
if $f \approx g,\ stat(f) = Mul,\ \pi$ a permutation and $ar(f) = ar(g) = n$
$Z \Rightarrow_{\texttt{bot}} \bot_\sigma$ if $Z : \sigma$

Figure 5.2: Rules of HOIPO

- The first four rule schemes essentially correspond to the rule schemes of ILPO; the next deals with multiset status, and the `abs` clause is new, for dealing with abstractions. The two `equiv` rule schemes are inherently non-terminating, and are used to generate the equivalence relation which corresponds to $\sim$. The `bot` rule reduces any term to a "minimal symbol" $\bot_\sigma$.
- The `put` rule is similar to its first-order counterpart: add a mark to the function symbol, without changing types. As with ILPO, the star denotes an intention to make the term smaller.
- The `select` rule adds new functionality: it can be used in the same way as the first-order `select` rule (taking for $\vec{x}$ the empty vector), but also deals with functional subterms. For example, if $\mathtt{f} : [\sigma \times (\tau \to \sigma)] \longrightarrow \sigma$, then $\mathtt{f}^\star_{\vec{\rho},\sigma}(\mathtt{a}, \lambda x.\mathtt{g}(x))$ reduces with the `select` rule to either `a` or to $\mathtt{g}(\mathtt{f}^\star_{\vec{\rho},\tau}(\mathtt{a}, \lambda x.\mathtt{g}(x)))$. Note that $f^\star$ might have a different type in its occurrences in the right-hand side than in the left-hand side (in the example, this happens if $\sigma \neq \tau$).
- The `copy` rule corresponds closely to its first-order counterpart, but again, the $f^\star(\vec{Z})$ might need to be retyped to fit at an argument position.

- The `lex` rule is where it gets interesting. Mostly, this rule stays true to its first-order counterpart (permitting for the necessary type-changing of marked terms), but the subterm which is marked might be an abstraction. Moreover, since $f^\star(\vec{s})$ may have more arguments than $ar(f)$, the subterm which is marked must be in the "original" set of arguments.
- This definition of the `mul` rule diverges both from the choice made in [72] and the one in [77]. Essentially, a term $f(s_1, \ldots, s_n, q_{n+1}, \ldots, q_k) \Rightarrow_{\texttt{mul}} g(t_1, \ldots, t_m)$ if $f \approx g$ and $stat(f) = Mul$, and moreover $\{\{\vec{s}\}\}(\Rightarrow_{\texttt{put}})_{Mul}\{\{\vec{t}\}\}$.
- The `abs` rule scheme simply introduces an abstraction; this rule scheme is the only one which alters the arity of some $f^\star$.

How can we use this new rewrite relation? Certainly, $\Rightarrow_\star$ is non-terminating. In the first-order case, termination is proved if $l \Rightarrow_\star^+ r$ for all rules $l \Rightarrow r$ in the original system, but doing something similar here is a bit troublesome: both because even on star-free terms $\Rightarrow_\star$ is not terminating, and because $\Rightarrow_\star^+$ is a relation on terms, and the rules use meta-terms. However, both objections are easily remedied.

First, let us consider an extension of the definition of a reduction.

Definition 5.9 (Meta-rewriting)**.** A *meta-substitution* γ follows exactly the definition of a substitution from Chapter 2.2, but maps to meta-terms rather than just terms. Applying a meta-substitution is defined the same as applying a substitution. A *meta-context* is a meta-term with exactly one occurrence of a special symbol $\Box_\sigma$.

As with terms, we say $s \Rightarrow_{\mathcal{R}} t$ for some set of rules $\mathcal{R}$ and meta-terms s, t if either $s = C[l\gamma]$ and $t = C[r\gamma]$ for some meta-context C, rule $l \Rightarrow r$ and meta-substitution γ, or $s = C[(\lambda x.q) \cdot u]$ and $t = C[q[x := u]]$ for some meta-context C and meta-terms q, u.

Denoting $\Rightarrow_{\lambda\texttt{put}}$ for the relation $\lambda\vec{x}.s \Rightarrow_{\lambda\texttt{put}} \lambda\vec{x}.s^\star$ (where $\vec{x}$ may have length 0), we have:

Theorem 5.10. *The pair* $(\Rightarrow_\star^*, \Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^*)$ *is a strong reduction pair on application-free meta-terms over* $\mathcal{F}$*.*

Note that $s \Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^* t$ if and only if $s^\star \Rightarrow_\star^* t$.

Proof. The last part of the proof, well-foundedness of $\Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^*$, is postponed to Section 5.3, but this theorem will not be used in that section.

It is evident that both $\Rightarrow_\star^*$ and $\Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^*$ are transitive (since $\Rightarrow_{\lambda\texttt{put}}$ is included in $\Rightarrow_\star$), that the former is reflexive, and that the pair is compatible.

Moreover, $\Rightarrow_\star$ is monotonic by the definition of (meta-)rewriting. As a consequence, $\Rightarrow_\star^*$ is monotonic as well, and so is $\Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^*$: we observe that $(\lambda x.s)^\star = \lambda x.s^\star$, and if $s_i^\star \Rightarrow_\star^* t_i$, then either

$$f^\star(s_1, \ldots, s_i, \ldots, s_n) \Rightarrow_{\texttt{mul}} f(s_1, \ldots, s_i^\star, \ldots, s_n) \text{ (if } stat(f) = Mul)$$

or, if $stat(f) = Lex$:

$$\begin{array}{rll} f^\star(s_1,\ldots,s_i,\ldots,s_n) & \Rightarrow_{\mathtt{lex}} & f(s_1,\ldots,s_i^\star, f^\star(\vec{s}),\ldots,f^\star(\vec{s})) \\ & \Rightarrow^*_{\mathtt{select}} & f(s_1,\ldots,s_i^\star,\ldots,s_n) \end{array}$$

Thus, if $s = C[t]$ and $t \Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow^*_\star q$, and C is an application-free context (not meta-context!), then $s \Rightarrow_{\lambda\mathtt{put}} s^\star \Rightarrow^*_\star C[t^\star] \Rightarrow^*_\star C[q]$ as required.

As for (meta-)stability, for any substitution whose domain contains $FMV(s)$ we have $s\gamma \Rightarrow^*_\star t\gamma$, and if the reduction $s \Rightarrow_\star t$ is at the top or directly below an abstraction at the top, then even $s\gamma \Rightarrow_\star t\gamma$. This immediately gives stability of the pair. The truth of these two claims follows in two steps:

(I) Let s be a meta-term, γ a meta-substitution on domain $FMV(s)$ and δ a substitution whose domain contains $FMV(s\gamma)$, and assume that the free variables in s do not occur in $dom(\delta)$. Then $(s\gamma)\delta = s(\gamma\delta)$. This holds with a simple induction on the form of s. The variable case is obvious by the domain restriction of δ, the case of a functional term and abstraction just use the induction hypothesis, and if $s = Z(s_1,\ldots,s_n)$, then consider γ. We can write $\gamma(Z) = \lambda x_1 \ldots x_n.t$, and have $(s\gamma)\delta = t[x_1 := s_1\gamma,\ldots,x_n := s_n\gamma]\delta = t\delta[x_1 := s_1\gamma\delta,\ldots,x_n := s_n\gamma\delta]$ since the x_i do not occur in $dom(\delta)$ (which we can assume by α-conversion). On the other hand, $s(\gamma\delta) = t\delta[x_1 := s_1(\gamma\delta),\ldots,x_n := s_n(\gamma\delta)]$, which by the induction hypothesis is the same thing.

(II) The two claims now hold by a straightforward induction on the form of s. In the base case, $s = l\delta$ and $t = r\delta$ for some meta-substitution γ on domain $FMV(l)$ and rule $l \Rightarrow r$, we use that $s\gamma = l(\delta\gamma) \Rightarrow_\star r(\delta\gamma) = t\gamma$ (the rules of $\mathcal{R}^\star$ do not contain free variables). All induction cases are obvious. The least trivial of them is when $s = Z(s_1,\ldots,s_i,\ldots,s_n)$ and $t = Z(s_1,\ldots,s_i',\ldots,s_n)$, in which case, writing $\gamma(Z) = \lambda x_1 \ldots x_n.q$, we have $s\gamma = q[x_1 := s_1\gamma,\ldots,x_i := s_i\gamma,\ldots,x_n := s_n\gamma] \Rightarrow^*_{\mathcal{R}} q[x_1 := s_1\gamma,\ldots,x_i := s_i'\gamma,\ldots,x_n := s_n\gamma] = t\gamma$ by induction hypothesis.

Finally, $\Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow^*_\star$ is well-founded by the combination of Lemma 5.13 and 5.18, which we will see in the next section. □

Example 5.11. Recall the recursor for the natural numbers:

$$\begin{array}{rcl} \mathtt{rec}(\mathtt{0},Y,F) & \Rightarrow & Y \\ \mathtt{rec}(\mathtt{s}(X),Y,F) & \Rightarrow & F \cdot X \cdot \mathtt{rec}(X,Y,F) \end{array}$$

To prove its termination, it suffices to convert it to an IDTS by the transformation described in Chapter 3.2 (a transformation which preserves and reflects termination), and show that:

$$\begin{array}{rcl} \mathtt{rec}^\star(\mathtt{0},Y,F) & \Rightarrow^*_\star & Y \\ \mathtt{rec}^\star(\mathtt{s}(X),Y,F) & \Rightarrow^*_\star & @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F,X),\mathtt{rec}(X,Y,F)) \end{array}$$

As a precedence, let $\mathtt{rec} \blacktriangleright @^{\sigma,\tau}$ for all σ; despite the infinite number of function symbols, this precedence is well-founded. The first rule is easily disposed of: we

immediately have $\mathtt{rec}^\star(\mathtt{0}, Y, F) \Rightarrow_{\mathtt{select}} Y$. For the second rule:

$$\begin{array}{ll}
& \mathtt{rec}^\star(\mathtt{s}(X), Y, F) \\
\Rightarrow_{\mathtt{copy}} & @^{\mathtt{nat},\mathtt{nat}}(\mathtt{rec}^\star_{\mathtt{nat}\to\mathtt{nat}}(\mathtt{s}(X), Y, F), \mathtt{rec}^\star(\mathtt{s}(X), Y, F)) \\
\Rightarrow_{\mathtt{mul}} & @^{\mathtt{nat},\mathtt{nat}}(\mathtt{rec}^\star_{\mathtt{nat}\to\mathtt{nat}}(\mathtt{s}(X), Y, F), \mathtt{rec}(\mathtt{s}^\star(X), Y, F)) \\
\Rightarrow_{\mathtt{select}} & @^{\mathtt{nat},\mathtt{nat}}(\mathtt{rec}^\star_{\mathtt{nat}\to\mathtt{nat}}(\mathtt{s}(X), Y, F), \mathtt{rec}(X, Y, F)) \\
\Rightarrow_{\mathtt{copy}} & @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(\mathtt{rec}^\star_{\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}}(\mathtt{s}(X), Y, F), \\
& \qquad \mathtt{rec}^\star(\mathtt{s}(X), Y, F)), \mathtt{rec}(X, Y, F)) \\
\Rightarrow_{\mathtt{select}} & @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F, \mathtt{rec}^\star(\mathtt{s}(X), Y, F)), \mathtt{rec}(X, Y, F)) \\
\Rightarrow_{\mathtt{select}} & @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F, \mathtt{s}(X)), \mathtt{rec}(X, Y, F)) \\
\Rightarrow_{\mathtt{put}} & @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F, \mathtt{s}^\star(X)), \mathtt{rec}(X, Y, F)) \\
\Rightarrow_{\mathtt{select}} & @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F, X), \mathtt{rec}(X, Y, F))
\end{array}$$

Here, type denotations are included for the $\mathtt{rec}^\star$ only where the typing diverges from the type declaration of $\mathtt{rec}$.

Example 5.12. For an example where meta-variables are used in a more interesting way, consider $\mathtt{map}$ from Example 2.3; assume that all base types have been collapsed to the same base type $\mathtt{o}$. Choosing a precedence $\mathtt{map} \blacktriangleright \mathtt{cons}, \mathtt{nil}$ and taking *Mul* as the status of all function symbols, we immediately obtain that $\mathtt{map}^\star(\lambda x.F(x), \mathtt{nil}) \Rightarrow_{\mathtt{copy}} \mathtt{nil}$. The second rule requires a bit more work:

$$\begin{array}{ll}
& \mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)) \\
\Rightarrow_{\mathtt{copy}} & \mathtt{cons}(\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)), \mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y))) \\
\Rightarrow_{\mathtt{mul}} & \mathtt{cons}(\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)), \mathtt{map}(\lambda x.F(x), \mathtt{cons}^\star(X, Y))) \\
\Rightarrow_{\mathtt{select}} & \mathtt{cons}(\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)), \mathtt{map}(\lambda x.F(x), Y)) \\
\Rightarrow_{\mathtt{select}} & \mathtt{cons}(F(\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y))), \mathtt{map}(\lambda x.F(x), Y)) \\
\Rightarrow_{\mathtt{select}} & \mathtt{cons}(F(\mathtt{cons}(X, Y)), \mathtt{map}(\lambda x.F(x), Y)) \\
\Rightarrow_{\mathtt{put}} & \mathtt{cons}(F(\mathtt{cons}^\star(X, Y)), \mathtt{map}(\lambda x.F(x), Y)) \\
\Rightarrow_{\mathtt{select}} & \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y))
\end{array}$$

5.3 Termination

To complete the theory of Section 5.2, we must see that $\Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow_\star^*$ is well-founded when considered as a relation on terms over $\mathcal{F}$. Following the proof approach used for the first-order case in [73], we show this by means of an auxiliary relation, which contains the restriction of $\Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow_\star^*$ to star-free terms and is well-founded. Unlike [73], this auxiliary relation is *not* itself a (meta-) rewrite relation in any of the common formalisms, both in order to simplify the termination proof, and because it is combined with an equivalence relation.

The relation $\hookrightarrow$ that we will define in this section is a relation on *AFSM-terms*. There is no restriction to purely functional terms. However, $\hookrightarrow$ itself will not be a rewrite relation. It is merely used to prove termination of $\Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow_\star^*$ and serves no other function.

Fixing a set of function symbols $\mathcal{F} = \mathcal{F}_1 \uplus \{\bot_\sigma \mid \sigma \text{ a type}\}$ as before, and a precedence $\blacktriangleright$ and status function $stat$ on $\mathcal{F}_1$, let the set $\mathcal{F}^\omega$ be given by:

$$\begin{array}{l}\{f^\omega : \sigma \mid f : \sigma \in \mathcal{F}_1\} \\ \cup \{\bot_\sigma \mid \sigma \text{ a type}\} \\ \cup \{f^n_{\tau_1,\ldots,\tau_k,\rho} : [\vec{\sigma} \times \tau_1 \times \ldots \times \tau_k] \longrightarrow \rho \mid \\ \qquad f : [\vec{\sigma}] \longrightarrow \alpha \in \mathcal{F}_1,\ n \in \mathbb{N},\ \tau_1, \ldots, \tau_k,\ \rho \text{ types}\}\end{array}$$

That is, $\mathcal{F}^\omega$ consists of all "minimal" symbols in $\mathcal{F}$, and in addition contains for all normal symbols $f : [\vec{\sigma}] \longrightarrow \alpha$ in $\mathcal{F}_1$ a symbol $f^\omega : \sigma$, as well as infinitely many symbols $f^n_{\vec{\tau},\rho} : [\vec{\sigma} \times \vec{\tau}] \longrightarrow \rho$. Such symbols exist for all n and combinations of new input types and output type.

Note that this is very similar to the definition of $\mathcal{F}^\star$, but the original symbols are marked with an ω (the first limit ordinal), and rather than one marked symbol $f^\star_{\vec{\tau},\rho}$ for all function symbols f and types $\vec{\tau}, \rho$, we introduce countably many symbols $f^n_{\vec{\tau},\rho}$ marked with a natural number.

We consider the equivalence relation $\sim$, which corresponds closely to the first-order definition, on terms over $\mathcal{F}^\omega$. Here, $s \sim t$ can *only* hold if s and t have the same type!

- $x \sim y$ if $x = y$ are variables;
- $\bot_\sigma \sim \bot_\sigma$;
- $\lambda x.s \sim \lambda x.t$ if $s \sim t$;
- $f(s_1, \ldots, s_n) \sim g(t_1, \ldots, t_n)$ if $f \approx' g$ (**) and:
  - if $stat(f) = Lex$, then $s_i \sim t_i$ for all i;
  - if $stat(f) = Mul$, then there is a permutation π of $\{1, \ldots, n\}$ such that always $s_{\pi(i)} \sim t_i$; if $f = h^k$ for some k, then $\pi(i) = i$ for $i > ar(h)$.

  (**) Here, $f \approx' g$ means that either $f = h^\omega,\ g = i^\omega$ and $h \approx i$, or $f = h^k_{\vec{\sigma},\tau},\ g = i^k_{\vec{\sigma},\tau}$ and $h \approx i$. In both cases, $ar(h) = ar(i)$ must hold. The status function is assumed to be extended to marked symbols, so $stat(h^k_{\vec{\sigma},\tau}) = stat(h^\omega) = stat(h)$.

We will consider terms over $\mathcal{F}^\omega$, which may contain application. For any term of the form $s = \lambda\vec{x}.f^\omega(\vec{t})$ with $f \in \mathcal{F}_1$, let s^n denote the term $\lambda\vec{x}.f^n_\tau(\vec{t})$ if $f : [\vec{\sigma}] \longrightarrow \tau$; s^n is not defined if s has a different form. Terms of the form $\lambda\vec{x}.f^\omega(\vec{t})$ are called *markable*.

Now consider the relation $\hookrightarrow$ on terms over $\mathcal{F}^\omega$, given by the following clauses:

1. $f^\omega(\vec{s}) \hookrightarrow f^n_\tau(\vec{s})$ for any $n \in \mathbb{N}$ and $f : [\vec{\sigma}] \longrightarrow \tau \in \mathcal{F}_1$
2. $f^{n+1}_{\vec{\sigma},\tau}(\vec{s}) \hookrightarrow s_i \cdot f^n_{\vec{\sigma},\rho_1}(\vec{s}) \cdots f^n_{\vec{\sigma},\rho_m}(\vec{s})$ if $s_i : \rho_1 \to \ldots \to \rho_m \to \tau$
3. $f^{n+1}_{\vec{\sigma},\tau}(\vec{s}) \hookrightarrow g^\omega(f^n_{\vec{\sigma},\rho_1}(\vec{s}), \ldots, f^n_{\vec{\sigma},\rho_m}(\vec{s}))$ if $f \blacktriangleright g : [\rho_1 \times \ldots \times \rho_m] \longrightarrow \tau$
4. $f^{n+1}_{\vec{\sigma},\tau}(s_1, \ldots, s_m, \vec{t}) \hookrightarrow g^\omega(s_1, \ldots, s_i^n, f^n_{\vec{\sigma},\alpha_{i+1}}(\vec{s}, \vec{t}), \ldots, f^n_{\vec{\sigma},\alpha_k}(\vec{s}, \vec{t}))$
   if $f : [\rho_1 \times \ldots \times \rho_m] \longrightarrow \rho'$ and $f \approx g$ and $stat(f) = Lex$ and $g : [\rho_1 \times \ldots \times \rho_i \times \alpha_{i+1} \times \ldots \times \alpha_k] \longrightarrow \tau$,
   where $k = ar(g) \geq i - 1$ and $m \geq i$;
   if $ar(g) = i - 1$ then s_i does not need to be markable
5. $f^{n+1}_{\vec{\sigma},\tau}(s_1, \ldots, s_m, \vec{t}) \hookrightarrow g^\omega(q_1, \ldots, q_k)$
   if $ar(f) = m$, $f \approx g$ and $stat(f) = Mul$, and $\{\{\vec{s}\}\} \sqsupset_{Mul} \{\{\vec{q}\}\}$ where u is the smallest relation such that $u \sqsupset u^n$ for all u, n, and the corresponding $\sqsupseteq$ is its reflexive closure; $g^\omega(\vec{q})$ must be well-typed, and must have type τ
6. $f^{n+1}_{\vec{\sigma},\tau\to\rho}(\vec{s}) \hookrightarrow \lambda x.f^n_{\vec{\sigma},\tau,\rho}(\vec{s}, x)$, for some fresh $x : \tau \in \mathcal{V}$
7. Some monotonicity clauses:

$$\begin{array}{rcll} f(s_1, \ldots, s_i, \ldots, s_n) & \hookrightarrow & f(s_1, \ldots, s_i', \ldots, s_n) & \text{if } s_i \hookrightarrow s_i', f \in \mathcal{F}^\omega \\ s \cdot t & \hookrightarrow & s' \cdot t & \text{if } s \hookrightarrow s' \\ \lambda s. & \hookrightarrow & \lambda s'. & \text{if } s \hookrightarrow s' \end{array}$$

   The fact that the right-hand side of an application does not reduce is deliberate; $\hookrightarrow$ does not need to be completely monotonic.
8. $(\lambda x.s) \cdot t \hookrightarrow s[x := t]$
9. $s \hookrightarrow \bot_\sigma$ if $s : \sigma$ and $s \neq \bot_\sigma$

Checking each of the clauses, $\hookrightarrow$ preserves types, and reduces well-typed terms to other well-typed terms over $\mathcal{F}^\omega$. Moreover, the relation is almost stable (if $s \hookrightarrow t$ then $s\gamma \hookrightarrow^= t\gamma$), and preserves $\sim$ (provided we add a clause $s_1 \cdot s_2 \sim t_1 \cdot t_2$ if each $s_i \sim t_i$): if $s \sim t$ and $t \hookrightarrow q$, then there is some u such that $s \hookrightarrow u$ and $q \sim u$. Thus, $\sim \cdot \hookrightarrow \subseteq \hookrightarrow \cdot \sim$. We will see shortly that $\hookrightarrow^+$ defines a well-founded ordering on terms. The relevance of this is given by the following result:

Lemma 5.13. *If $\hookrightarrow$ is well-founded on terms over $\mathcal{F}^\omega$, then $\Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow^*_\star$ is well-founded on meta-terms over $\mathcal{F}$.*

Proof. For s an application-free term over $\mathcal{F}$, and t a term over $\mathcal{F}^\omega$, let $s\ R^n\ t$ if this can be derived with the following inductive clauses:

$$\begin{array}{rcll} x & R^n & x & \text{if } x \text{ a variable, for all } n \in \mathbb{N} \\ \lambda x.s & R^n & \lambda x.t & \text{if } s\ R^n\ t \\ f(\vec{s}) & R^n & f^\omega(\vec{s'}) & \text{if } f \in \mathcal{F}, \text{ each } s_i\ R^n\ s_i' \\ f^\star_{\vec{\sigma},\tau}(\vec{s}) & R^n & f^k_{\vec{\sigma},\tau}(\vec{s'}) & \text{if } \mathbb{N} \ni k \geq n, \text{ each } s_i\ R^n\ s_i' \end{array}$$

Note that a term containing stars is associated to infinitely many terms over $\mathcal{F}^\omega$, but to a star-free term we can assign a unique term $trans(s)$ such that $s\ R^n\ trans(s)$ for all $n \in \mathbb{N}$.

By stability of $\Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^*$, it suffices to see that for any two application-free *terms* s, t: if $s \Rightarrow_{\texttt{put},top} \cdot \Rightarrow_\star^* t$ then $trans(s) \hookrightarrow^+ trans(t)$. We don't need to prove anything special for meta-terms, since an infinite reduction on meta-terms $s_1, s_2, \ldots$ can be transformed into an infinite reduction on terms $s_1\gamma, s_2\gamma, \ldots$

Claim: *if s, t are application-free terms over $\mathcal{F}^\star$ such that $s \Rightarrow_\star t$, and $s\ R^{N+1} q$ for some term q, then there is a term u such that $t\ R^N\ u$ and either $q \sim u$ or $q \hookrightarrow^+ u$; if the step $s \Rightarrow_\star t$ uses a* `put` *rule even $q \hookrightarrow u$.*

Suppose the claim holds, and recall that: (*) $\sim \cdot \hookrightarrow \subseteq \hookrightarrow \cdot \sim$, and $\sim$ is an equivalence relation. Towards a contradiction, suppose that $\hookrightarrow$ is well-founded while $\Rightarrow_{\lambda\texttt{put}} \cdot \Rightarrow_\star^*$ is not well-founded on terms over $\mathcal{F}$. Thus, there are star-free terms $s_1, s_2, \ldots$ such that each $s_i \Rightarrow_{\texttt{put}} \cdot \Rightarrow_\star^* s_{i+1}$. Suppose the reduction from s_i to s_{i+1} has length n. Since s_i is star-free, we know that $s_i\ R^{n+1}\ trans(s_i)$, and by the claim and (*), we have that $trans(s_i) \hookrightarrow^+ u \sim v$ for some terms u, v such that $s_{i+1}\ R^0\ v$. But, considering the definition of R^0, this term v can only be $trans(s_{i+1})$. Thus, $trans(s_1) \hookrightarrow^+ \cdot \sim trans(s_2) \hookrightarrow^+ \cdot \sim \ldots$ gives an infinite descending sequence, which, using (*) again, generates an infinite descending sequence over $\hookrightarrow$, contradiction.

It remains to be seen that the claim holds, which we will do by induction on the size of s. To this end, we make the following observation: *(**) If $v\ R^{N+1}\ w$, then also $v\ R^N\ w$.* This is obvious due to the $\geq n$ in the definition.

Now, let $s \Rightarrow_\star t$ and suppose $s\ R^{N+1}\ q$. We must find some u such that $t\ R^N\ u$ and either $q \sim u$ or $q \hookrightarrow^+ u$. Consider the form of s and of the reduction. First suppose $s \Rightarrow_{\texttt{bot}} t$ by a topmost reduction, so $t = \bot_\sigma$. If also $s = \bot_\sigma$ then $s \sim t$ and we are done, otherwise $s \hookrightarrow t$ by clause 9. Otherwise, either the reduction uses another rule, or is done in a subterm. It is clear that s cannot be a variable, as variables do not reduce (other than with $\Rightarrow_{\texttt{bot}}$).

Suppose the reduction is done in a subterm. Since s is application-free, there are two possibilities. Either $s = f(s_1, \ldots, s_i, \ldots, s_n) \Rightarrow_\star f(s_1, \ldots, s_i', \ldots, s_n) = t$ for some $f \in \mathcal{F}^\star$ or $s = \lambda x.s' \Rightarrow_\star \lambda x.t' = t$. In either case, we just use the relevant monotonicity clause, and (**). For example, if $f = g^\star$, then q has the form $g^k(q_1, \ldots, q_n)$ where each $s_j\ R^{N+1}\ q_j$ and $k \geq N+1$. By the induction hypothesis we find q_i' such that either $q_i \hookrightarrow^+ q_i'$ or $q_i \sim q_i'$, and $s_i'\ R^n\ q_i'$. Thus, using the first monotonicity clause or the definition of $\sim$ respectively, either $q \hookrightarrow^+ u := g^k(q_1, \ldots, q_i', \ldots, q_n)$ or $q \sim u$, and moreover $t\ R^N\ u$ because also $k \geq N$, each $s_j\ R^N\ q_j$ by (**), and $s_i'\ R^N\ q_i'$ as we have seen. The case where s is an abstraction is even easier. Note that the number of steps in the reduction $q \hookrightarrow^+ u$ or $q \sim u$ corresponds with the number of steps in the reduction in the subterm.

What remains is the base case: $s \Rightarrow_\star t$ by a topmost step. We consider each of the rules that might have been used.

put $s = f(\vec{s})$, $t = f^\star(\vec{s})$ and $q = f^\omega(\vec{q})$ with each $s_i\ R^{N+1}\ q_i$. Then $q \hookrightarrow u := f^N(\vec{q})$, and indeed $t\ R^N\ u$ by (**). Note that a single `put` step is replaced by a single $\hookrightarrow$ step, a property which is preserved in the induction step.

select $s = f^\star_{\vec{\sigma},\tau}(\vec{s})$ with $s_i = \lambda\vec{x}.v$ and $t = v[x_1 := f^\star_{\vec{\sigma},\rho_1}(\vec{s}), \ldots, x_n := f^\star_{\vec{\sigma},\rho_m}(\vec{s})]$, using the right retypings; $q = f^{k+1}_{\vec{\sigma},\tau}(\vec{q})$ for some $k \geq N$. Then we also have $f^\star_{\vec{\sigma},\rho_j}(\vec{s})\ R^N\ f^k_{\vec{\sigma},\rho_j}(\vec{q})$ for all j (using (**)). We can write $q_i = \lambda\vec{x}.w$ with $v\ R^{n+1}\ w$.

Now note: if $s'\ R^p\ t'$ and γ, δ are substitutions on a domain of variables, such that each $\gamma(x)\ R^p\ \delta(x)$, then also $s'\gamma\ R^p\ t'\delta$, as is demonstrated by a simple induction on the form of s'.

Therefore, $t\ R^N\ u := w[x_1 := f^k_{\vec{\sigma},\rho_1}(\vec{q}), \ldots, x_n := f^k_{\vec{\sigma},\rho_n}(\vec{q})]$. By clauses 2 and 8, also $q \hookrightarrow q_i \cdot f^k_{\vec{\sigma},\rho_1}(\vec{q}) \cdots f^k_{\vec{\sigma},\rho_n}(\vec{q}) \hookrightarrow^* u$.

copy $s = f^\star_{\vec{\sigma},\tau}(\vec{s})$ and $t = g(f^\star_{\vec{\sigma},\rho_1}(\vec{s}), \ldots, f^\star_{\vec{\sigma},\rho_m}(\vec{s}))$ with $f \blacktriangleright g$, using the right retypings. Write $q = f^{k+1}_{\vec{\sigma},\tau}(\vec{q})$ with $k \geq N$; let $u := g^\omega(f^k_{\vec{\sigma},\rho_1}(\vec{q}), \ldots, f^k_{\vec{\sigma},\rho_m}(\vec{q}))$. Then $q \hookrightarrow u$ by clause 3, and $t\ R^N\ u$ by (**).

lex $s = f^\star(s_1, \ldots, s_n)$ and $t = g(s_1, \ldots, s_{i-1}, s_i^\star, f^\star(\vec{s}), \ldots, f^\star(\vec{s}))$ with $f \approx g$ and $stat(f) = Lex$ and $i \leq ar(f)$, $i - 1 \leq ar(g)$. We can write $q = f^{k+1}(q_1, \ldots, q_n)$ with $n \geq ar(f) \geq i$ and $k \geq N$. If $i - 1 = ar(g)$, so $t = g(s_1, \ldots, s_{i-1})$, then let $u := g^k(q_1, \ldots, q_{i-1})$. Certainly $t\ R^N\ u$, and also $q \hookrightarrow u$ by clause 4. Otherwise, s_i is a markable term; let $u := g^\omega(q_1, \ldots, q_{i-1}, q_i^k, f^k(\vec{q}), \ldots, f^k(\vec{q}))$, with suitable retypings. Then clearly $t\ R^N\ u$, and also $q \hookrightarrow u$ by clause 4.

mul $s = f^\star(s_1, \ldots, s_m, \ldots, s_n)$ and $t = g(\vec{t})$ with $f \approx g$ and $stat(f) = Mul$ and $m = ar(f)$. By the rule definition, choosing for C the set of indexes where the rule uses a meta-variable, and for D the set where it does not, we can find some function π mapping $\{1, \ldots, ar(g)\}$ to $\{1, \ldots, m\}$ such that if $\pi(i) \in C$ then $s_{\pi(i)} = t_i$, and if $\pi(i) \in D$ then $s^\star_{\pi(i)} = t_i$. Moreover, π is injective on C, and $\{\{\vec{s}\}\} \neq \{\{\vec{t}\}\}$, which means that either D is non-empty, or $\{\{\vec{t}\}\}$ is a strict sub-multiset of $\{\{\vec{s}\}\}$.

We can write $q = f^{k+1}(q_1, \ldots, q_m, \ldots, q_n)$ with each $s_j\ R^{N+1}\ q_j$. Let $u := g^\omega(u_1, \ldots, u_{ar(g)})$, where $u_j = q_{\pi(j)}$ if $\pi(j) \in C$, and $u_j = q^n_{\pi(j)}$ if $\pi(j) \in D$. Then clearly $t\ R^N\ u$. We also have $q \hookrightarrow u$ if we can prove: $\{\{\vec{q}\}\} \sqsupset_{Mul} \{\{\vec{u}\}\}$.

Now, let A be the set of indexes j such that $j \in C$ and there is some i such that $\pi(i) = j$; let $B := \{1, \ldots, m\} \backslash A$. Then B is non-empty, because $D \subseteq B$ and, if $\{\{\vec{t}\}\}$ has fewer elements than $\{\{\vec{s}\}\}$, then at least one element of C does not occur in A. Since π is injective on C, for every element j of A there is exactly one i such that $\pi(i) = j$. For all $i \in \{1, \ldots, ar(g)\}$, if $\pi(i) \in A$, then $\pi(i) \in C$ so $q_{\pi(i)} = u_i$, so certainly $q_{\pi(i)} \sqsupseteq u_j$, and if $\pi(i) \in B$, then $\pi(i) \in D$, so $q^n_{\pi(i)} = u_i$, and therefore $q_{\pi(i)} \sqsupset u_i$

abs $s = f^{\star}_{\vec{\sigma},\tau\to\rho}(\vec{s})$ and $t = \lambda x.f^{\star}_{\vec{\sigma},\tau,\rho}(\vec{s},x)$. We can write $q = f^{k+1}_{\vec{\sigma},\tau\to\rho}(\vec{q})$ with $k \geq N$. Choose $u := \lambda x.f^{k}_{\vec{\sigma},\tau,\rho}(\vec{q},x)$. Then both $t\ R^N\ u$, and $q \hookrightarrow u$ by clause 6.

equiv I $s = f(\vec{s})$, $t = g(\vec{s})$ with $f \approx g$ and $stat(f) = Lex$; let $q = f^{\omega}(\vec{q})$ and let $u := g^{\omega}(\vec{q})$. Then $q \sim u$ and $t\ R^N\ u$ by (**).

equiv II $s = f(s_1,\ldots,s_n)$ and $t = g(s_{\pi(1)},\ldots,s_{\pi(n)})$ for some permutation π and symbols $f \approx g$ with status Mul. Writing $q = f^{\omega}(q_1,\ldots,q_n)$ choose $u := g^{\omega}(q_{\pi(1)},\ldots,q_{\pi(n)})$. Then clearly $t\ R^{N+1}\ u$ so also $t\ R^N\ u$ by (**), and $q \sim u$.

The rules **bot** we already handled. □

Having proved this much, it only remains to be seen that $\hookrightarrow$ is indeed terminating. To see this, we will use Tait's and Girard's *computability* technique. The definition and proof here follows closely that in [63] (but can be simplified because the application $\cdot$ is only monotonic in its first argument).

Definition 5.14 (Computability with Respect to $\hookrightarrow$)**.** A base-type term is computable if it is terminating under $\hookrightarrow$. A term $s : \sigma \to \tau$ is computable if for all computable terms t of type σ also $s \cdot t$ is computable.

We quickly obtain the following results:

Lemma 5.15. *For all types σ:*

1. *If $s : \sigma$ is not an abstraction, it is computable if all its direct $\hookrightarrow$ reducts are computable.*
2. *The minimal symbol $\bot_\sigma$ is a computable term.*
3. *All variables of type σ are computable.*
4. *All computable terms of type σ are terminating.*
5. *If $s : \sigma$ is computable, then for all t with $s \hookrightarrow t$ also t is computable.*

Proof. By joined induction on σ. If σ is a base type, then all five claims are obvious because computability coincides with termination. We merely need to note that the $\bot_\sigma$ symbols do not reduce to anything, and a variable $x : \sigma$ reduces only to $\bot_\sigma$.

If we know that the claims hold for σ and τ, then they also hold for $\sigma \to \tau$:

1. Suppose s is not an abstraction, and all its $\hookrightarrow$-reducts are computable; s is computable if $s \cdot t$ is computable for all computable $t : \sigma$. By induction hypothesis 1, $s \cdot t : \tau$ is computable if all its reducts are, so if $\bot_\tau$ is computable (which is the case by induction hypothesis 2), and $s' \cdot t$ is computable whenever $s \hookrightarrow s'$ (as this is the only other way an application reduces when its

head is not an abstraction). But this is obviously the case, since such s' is computable by assumption.

2. By 1 (which is already proved), $\bot_{\sigma\to\tau}$ is computable if its direct reducts are; since it has none, we are done.

3. By 1 (which is already proved), a variable $x : \sigma \to \tau$ is computable if its direct reducts are. But this is only $\bot_{\sigma\to\tau}$, which, by 2 (which is already proved), is computable.

4. To see that a computable term $s : \sigma \to \tau$ is terminating, suppose this is not the case: $s \hookrightarrow s_1 \hookrightarrow s_2 \hookrightarrow \ldots$. By induction hypothesis 3 variables of type σ are computable, so $s \cdot x$ is computable for some variable, and therefore terminating by induction hypothesis 4. However, by clause 7 of the definition of $\hookrightarrow$ also $s \cdot x \hookrightarrow s_1 \cdot x \hookrightarrow s_2 \cdot x \hookrightarrow \ldots$, contradiction.

5. Suppose $s : \sigma \to \tau$ is computable and $s \hookrightarrow s'$; we must see that $s' \cdot t$ is computable for all computable $t : \sigma$. But this is obvious: $s \cdot t$ is computable by definition, and by induction hypothesis 5 its reduct $s' \cdot t$ is computable, too.

□

Lemma 5.16. *If $s[x := t]$ is computable for all computable $t : \sigma$, then $\lambda x.s : \sigma \to \tau$ is computable.*

Proof. Assume $s[x := t]$ is computable for all computable $t : \sigma$. Then also s itself is computable, since $s = s[x := x]$ and x is computable by Lemma 5.15(3). By Lemma 5.15(4), s is terminating. Thus, let us prove the lemma with induction on s, ordered with $\hookrightarrow$. By the induction hypothesis, $\lambda x.s'$ is computable for all reducts $s \hookrightarrow s'$, because if $s \hookrightarrow s'$, then $s[x := t] \hookrightarrow s'[x := t]$, is computable for all computable t by Lemma 5.15(5).

By definition, $\lambda x.s : \sigma \to \tau$ is computable if $(\lambda x.s) \cdot t$ is computable for all computable $t : \sigma$. By Lemma 5.15(1) this is the case if all direct reducts of this term are computable. It has two reducts: $s[x := t]$, which is computable by assumption, and $(\lambda x.s') \cdot t$ for some s' such that $s' \hookrightarrow s$, in which case the induction hypothesis gives computability of the result. □

Lemma 5.17. *If $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}^\omega$ and $s_1 : \sigma_1, \ldots, s_n : \sigma_n$ are computable terms, then $f(s_1, \ldots, s_n)$ is computable as well.*

Proof. In the definition of *status*, we have assumed that if $stat(f) = Lex$, then there is some N such that all symbols $\approx$-equal to f have arity at most N. Let maxarity be a function mapping f with $stat(f) = Lex$ to the smallest such N.

We will prove, by induction on $\langle f, (s_1, \ldots, s_{ar(f)}), k, (s_{ar(f)+1}, \ldots, s_n)\rangle$, that the term $f^k_{\vec{\sigma},\tau}(s_1, \ldots, s_n)$ is computable if all s_i are computable, where $k \in \mathbb{N} \cup \{\omega\}$, for $f \in \mathcal{F}_1$. Note that if $f = \bot_\sigma$ we already know that the term is

computable by Lemma 5.15(2). The components of the induction are ordered lexicographically. The first component, f, is ordered by $\blacktriangleright$ (with two $\approx$-equal symbols considered equal). The second component is ordered by $\hookrightarrow_{lex}$ restricted to sequences of length at most $\mathsf{maxarity}(f)$ if the status of the first component is Lex, otherwise by $\hookrightarrow_{mul}$. This relation is well-founded when restricted to terms which terminate under $\hookrightarrow$, and this holds for the s_i by Lemma 5.15(4). Note that if $f \approx g$, then $stat(f) = stat(g)$, and if this status is Lex then $\mathsf{maxarity}(f) = \mathsf{maxarity}(g)$, so this dependence of the second component of the induction hypothesis on the first is not problematic. The third component, k, is ordered by the normal greater-than relation for natural numbers, with additionally $\omega > k$ for all $k \in \mathbb{N}$, and the fourth by the multiset extension of $\hookrightarrow$, which is well-founded because computable terms terminate.

$f^k(\vec{s})$ is not an abstraction, so by Lemma 5.15(1) it suffices to see that all its reducts are computable. We will consider each clause that might be used to obtain $s = f^k(\ldots) \hookrightarrow t$, and show that t is computable. In several of the clauses, the left-hand side has the form $f^{k+1}_{\vec{\sigma},\tau}(\vec{s})$ while the right-hand side has subterms $f^{k}_{\vec{\sigma},\rho}(\vec{s})$. In each of these subterms, note that we have not made any assumptions about the σ_i and τ. Thus, this subterm is computable by the third component of the induction hypothesis. Let us refer to this reasoning as (***).

1. The right-hand side is computable by the third component of the induction hypothesis (the others are unaltered).

2. The right-hand side is an application of computable terms by (***) and the assumption that all s_i are computable.

3. This reduction lowers the first component of the induction hypothesis. We merely need to see that indeed all subterms of the right-hand side are computable, which holds by (***).

4. $s = f^{k+1}_{\vec{\sigma},\tau}(s_1, \ldots, s_m, \ldots, s_n)$ with $m = ar(f) \leq \mathsf{maxarity}(f)$, and $t = g^\omega(s_1, \ldots, s_i^k, \vec{q})$ where $stat(f) = Lex$, $f \approx g$, all q_j are computable by (***), $i \leq m$ and $ar(g)$, which is the length of the sequence $[s_1, \ldots, s_i^k, \vec{q}]$, is at most $\mathsf{maxarity}(g) = \mathsf{maxarity}(f)$.

   As the first component is unchanged (modulo $\approx$), and all arguments of g^ω are computable (both the q_j, the s_j, and s_i^k by Lemma 5.15(5) because $s_i \Rightarrow_{\mathtt{put}} s_i^k$), we can apply the induction hypothesis if the second component (originally $(s_1, \ldots, s_m)$ is decreased, which, following the definition of the lexicographic extension, is the case both if $ar(g) = i-1$ and if $ar(g) \geq i$, because $s_i \Rightarrow_{\mathtt{put}} s_i^n$).

5. As in the previous case, this reduction keeps the first component of the induction hypothesis unchanged, while lowering the second, as $q \Rightarrow_{\mathtt{put}} q^n$.

6. $s = f^{k+1}_{\vec{\sigma},\tau\to\rho}(\vec{s})$ and $t = \lambda x.f^{k}_{\vec{\sigma},\tau,\rho}(\vec{s},x)$. By the third component of the induction hypothesis, $f^{k}_{\vec{\sigma},\tau,\rho}(\vec{s},q)$ is computable for all computable q (note that all arguments are still computable as required, and that the second component is $(s_1,\ldots,s_{ar(f)})$, and is not altered by adding an additional argument q; the fourth component increases, but the third component is more significant). By Lemma 5.16, this means that t is computable.

7. The monotonicity clause reduces either the second or fourth component, while keeping the others the same.

8. $t = \bot_\sigma$ is computable by Lemma 5.15(2).

□

The results on the computability of terms with various forms can now be combined in one termination result:

Lemma 5.18. *For all terms s and substitutions γ where all $\gamma(x)$ are computable terms, $s\gamma$ is computable.*

Proof. By induction on the form of s.

If s is a variable, then $s\gamma$ is either computable by Lemma 5.15(3) (if $s \notin dom(\gamma)$), or it equals $\gamma(x)$ and is computable by assumption.

If $s = \lambda x.t$, then using α-conversion x can be assumed to be fresh, so $\delta := \gamma \cup [x := q]$ is a computable substitution for any computable term δ. Then $t\delta$ is computable by the induction hypothesis, and $t\delta = t\gamma[x := q]$. Thus, $\lambda x.(t\gamma)$ is computable by Lemma 5.16.

If $s = f(s_1,\ldots,s_n)$, then by the induction hypothesis all $s_i\gamma$ are computable. By Lemma 5.17 this means that $f(s_1\gamma,\ldots,s_n\gamma) = s\gamma$ is computable. □

Thus, all terms are computable (choosing for γ the empty substitution) and therefore terminating (by Lemma 5.15(4)). This completes the proof of Theorem 5.10.

5.4 StarHorpo

The iterative method is elegant, but has a significant downside when trying to prove termination automatically: it is not at all evident whether or not $s \Rightarrow^*_\star t$, and without a suitable *reduction strategy*, we have little way to test other than reducing terms for an arbitrary amount of time, and seeing whether we find t eventually.

Rather than defining a reduction strategy, let us study a recursive definition of the same ordering. This will serve to provide some decidability for the relation and give a direction for a future reduction strategy. Moreover, this makes the relation more comparable to common definitions of the higher-order recursive path ordering. We shall call the resulting reduction pair StarHorpo.

To start, consider the following definition, which shall be needed to deal with the `select` rule recursively. For meta-terms $s : \sigma_1 \to \ldots \to \sigma_n \to \tau$ and $t_1 : \sigma_1, \ldots, t_n : \sigma_n$, let $s\langle t_1, \ldots, t_n\rangle$ be defined by the following clauses:

- if $s = \lambda x_1 \ldots x_n.q$, then $s\langle t_1, \ldots, t_n\rangle = q[x_1 := t_1, \ldots, x_n := t_n]$ (this includes the case where $n = 0$, so $s\langle\rangle = s$ for any s);
- if $s = \lambda x_1 \ldots x_k.f(s_1, \ldots, s_m)$ with $k < n$, then let $\gamma := [x_1 := t_1, \ldots, x_k := t_k]$ and define $s\langle t_1, \ldots, t_n\rangle = f^\star_{\sigma_{k+1},\ldots,\sigma_n,\tau}(s_1\gamma, \ldots, s_m\gamma, t_{k+1}, \ldots, t_n)$;
- if $s = \lambda x_1 \ldots x_k.f^\star_{\rho_1,\ldots,\rho_p,\sigma_{k+1}\to\ldots\to\sigma_n\to\tau}(s_1, \ldots, s_m)$ with $k < n$, then let $\gamma := [x_1 := t_1, \ldots, x_k := t_k]$, and define $s\langle t_1, \ldots, t_n\rangle = f^\star_{\rho_1,\ldots,\rho_p,\sigma_{k+1},\ldots,\sigma_n,\tau}(s_1\gamma, \ldots, s_m\gamma, t_{k+1}, \ldots, t_n)$.

If s does not have one of these forms (for example, if $n > 0$ and s is a variable or meta-variable), then $s\langle t_1, \ldots, t_n\rangle$ is simply not defined.

StarHorpo is a strong reduction pair $(\succeq_\star, \succ_\star)$. Unlike the recursive definitions we have seen so far, $\succeq_\star$ is not the union of $\succ_\star$ with some equivalence relation. Rather, the latter is expressed essentially as $\Rightarrow_{\lambda\mathtt{put}} \cdot \succeq_\star$. This is both to preserve a strong relation with the iterative setting, and because, in the setting with meta-variables, the quasi-ordering $\succeq_\star$ should be *more* than the union of a strict relation and an equivalence relation: it would be useful to at least have $F(s) \succeq_\star F(t)$ when $s \succ_\star t$, but if F is a meta-variable then neither an equivalence $F(s) \sim F(t)$ nor a strict relation $F(s) \succ_\star F(t)$ would be preserved under substitution.

Definition 5.19. Let $s \succ_\star t$ if $s^\star \succeq_\star t$, where $\succeq_\star$ is a relation on equal-typed meta-terms, which is recursively defined as follows:

(Var)	x	$\succeq_\star$	x	if	$x \in \mathcal{V}$
(Abs)	$\lambda x.s$	$\succeq_\star$	$\lambda x.t$	if	$s \succeq_\star t$
(Meta)	$Z(\vec{s})$	$\succeq_\star$	$Z(\vec{t})$	if	each $s_i \succeq_\star t_i$
(Fun)	$f(\vec{s})$	$\succeq_\star$	$g(\vec{t})$	if	$f \approx g$ and $\lvert\vec{s}\rvert = \lvert\vec{t}\rvert$ and $\vec{s} \succeq_{\star gstat(f)} \vec{t}$ [1]
(Put)	$f(\vec{s})$	$\succeq_\star$	t	if	$f^\star(\vec{s}) \succeq_\star t$
(Select)	$f^\star(\vec{s})$	$\succeq_\star$	t	if	$s_i\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle \succeq_\star t$ [2]
(F-Abs)	$f^\star(\vec{s})$	$\succeq_\star$	$\lambda x.t$	if	$f^\star(\vec{s}, x) \succeq_\star t$
(Copy)	$f^\star(\vec{s})$	$\succeq_\star$	$g(\vec{t})$	if	$f \blacktriangleright g$ and $f^\star(\vec{s}) \succeq_\star t_i$ for all i [3]
(Stat)	$f^\star(\vec{s})$	$\succeq_\star$	$g(\vec{t})$	if	$f \approx g$ and $(s_1, \ldots, s_{ar(f)}) \succ_{\star gstat(f)} \vec{t}$ [1] and $f^\star(\vec{s}) \succeq_\star t_i$ for all i [3,4]
(Bot)	s	$\succeq_\star$	$\bot_\sigma$	if	$s : \sigma$

[1] Here, we use the generalised multiset or lexicographic extensions of $(\succeq_\star, \succ_\star)$, as defined at the start of Section 5.1.
[2] Here, the $f^\star(\vec{s})$ instances are retyped to fit the type of s_i. s_i itself may not be retyped, and the output type of s_i must be the same as the type of t.
[3] Here, the $f^\star(\vec{s})$ instances are retyped to fit the type of t_i.
[4] Note that if $(s_1, \ldots, s_n) \succ_{\star gMul} (t_1, \ldots, t_m)$, then automatically $f^\star(\vec{s}, \vec{q}) \succeq_\star t_i$ for all i: this is easily seen with clause (Select), converting a clause $s_i \succ_\star t_j$ into $s_i \succeq_\star t_j$ either by (Abs) and (Put), or by (Bot).

This recursive definition corresponds in a close way to the definition of $\Rightarrow_\star$, and has been designed to generate the same relation. Let us boldly venture the following claim:

Claim 5.20. *For application-free meta-terms over $\mathcal{F}$:*

- $s \Rightarrow_\star^* t$ *if and only if* $s \succeq_\star t$
- $s \Rightarrow_{\mathtt{put},top} \cdot \Rightarrow_\star^* t$ *if and only if* $s \succ_\star t$

Here, $\Rightarrow_\star^*$ is the extension of reduction to meta-terms from Definition 5.9.

Suppose Claim 5.20 holds. Then by Theorem 5.10, $(\succeq_\star, \succ_\star)$ is a reduction pair.

Example 5.21. Consider once more the main rule from Example 5.7. Choosing again $stat(\mathtt{rec}) = Mul$, we can prove its termination in a very similar way, but using StarHorpo instead of CPO. Let $\mathtt{rec} \blacktriangleright @^\sigma$ for all σ.

1. $\mathtt{rec}^\star(\mathtt{s}(X), Y, F) \succeq_\star @^{\mathtt{nat},\mathtt{nat}}(@^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F, X), \mathtt{rec}(X, Y, F))$ by (Copy), 2 and 3.
2. $\mathtt{rec}^\star_{\mathtt{nat}\to\mathtt{nat}}(\mathtt{s}(X), Y, F) \succeq_\star @^{\mathtt{nat},\mathtt{nat}\to\mathtt{nat}}(F, X)$ by (Copy), 4 and 5.
3. $\mathtt{rec}(\mathtt{s}(X), Y, F) \succ_\star \mathtt{rec}(X, Y, F)$ by (Stat), remark [4] below the definition of $\succeq_\star$, 6 and two applications of (Meta) for the last two arguments.
4. $\mathtt{rec}^\star_{\mathtt{nat}\to\mathtt{nat}\to\mathtt{nat}}(\mathtt{s}(X), Y, F) \succeq_\star F$ by (Select), where $F \gtrsim F$ by (Meta).
5. $\mathtt{rec}^\star(\mathtt{s}(X), Y, F) \succeq_\star X$ by (Select), (Put) and 6.
6. $\mathtt{s}^\star(X) \succeq_\star X$ by clauses (Select) and (Meta).

Example 5.22. Consider the `map` function which we proved to be terminating with HOIPO in Example 5.12. The following reasoning proves its termination in the recursive way, taking $\mathtt{map} \blacktriangleright \mathtt{nil}, \mathtt{cons}$.

1. $\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)) \succeq_\star \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y))$ by clause (Copy) and 2 and 3.
2. $\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)) \succeq_\star F(X)$ by clause (Select) and 4.
3. $\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y)) \succeq_\star \mathtt{map}(\lambda x.F(x), Y)$ by clause (Stat), 5, 6, and remark [4].
4. $(\lambda x.F(x))\langle \mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y))\rangle = F(\mathtt{map}^\star(\lambda x.F(x), \mathtt{cons}(X, Y))) \succeq_\star F(X)$ by (Meta) and 7
5. $\lambda x.F(x) \succeq_\star \lambda x.F(x)$ by (Abs), (Meta) and (Var).
6. $\mathtt{cons}(X, Y)) \succ_\star Y$ because $\mathtt{cons}^\star(X, Y)) \succeq_\star Y$ by (Select) and (Meta).

7. $\texttt{map}^\star(\lambda x.F(x), \texttt{cons}(X,Y)) \succeq_\star X$ by (Select) and 8.
8. $\texttt{cons}(X,Y)) \succeq_\star X$ by (Put), (Select) and (Meta).

Whether HOIPO or StarHorpo is a better choice depends on your preferences, and the setting (for example, the recursive version is likely to be a better choice for an automatic tool, and is used for instance in WANDA, as we will see in Chapter 8, while the iterative approach is perhaps more charming for a classroom).

The rest of this section is dedicated to proving that Claim 5.20 indeed holds. First, we shall consider an extension $\succeq^e_\star$ of $\succeq_\star$ to terms over $\mathcal{F}^\star$, whose restriction to terms over $\mathcal{F}$ is exactly $\succeq_\star$. We will see that $\Rightarrow_\star$ is included in $\succeq^e_\star$, that that $\succeq_\star$ is included in $\Rightarrow^*_\star$, and consequently, that $\Rightarrow^*_\star$ coincides with $\succeq^*_\star$ on star-free terms. Next, we will see that $\succeq^e_\star$ is transitive, and therefore $\succeq_\star$ itself coincides with $\Rightarrow^*_\star$. The proof of Claim 5.20 then follows quickly.

5.4.1 Extending $\succeq_\star$

Claim 5.20 states a strong relation between $\Rightarrow^*_\star$ and $\succeq_\star$. A first step towards proving this claim is the following result, which shows that $\succeq_\star$ is at most as powerful as $\Rightarrow^*_\star$.

Lemma 5.23. *Let s, t be application-free meta-terms over $\mathcal{F}^\star$. If $s \succeq_\star t$ then $s \Rightarrow^*_\star t$.*

Proof. By induction on the derivation of $s \succeq_\star t$; consider the clause used to derive this inequality.

(Var) $x \Rightarrow^*_\star x$ because $\Rightarrow^*_\star$ is reflexive.

(Abs) $\lambda x.s \Rightarrow^*_\star \lambda x.t$ because by induction $s \Rightarrow^*_\star t$.

(Meta) $Z(\vec{s}) \Rightarrow^*_\star Z(\vec{t})$ because by induction each $s_i \Rightarrow^*_\star t_i$.

(Fun) If $stat(f) = Lex$, then $f(\vec{s}) \Rightarrow_{\texttt{equiv}} g(\vec{s}) \Rightarrow^*_\star g(\vec{t})$ because each $s_i \Rightarrow^*_\star t_i$ by induction hypothesis.

If $stat(f) = Mul$, then there is some permutation π such that $s_{\pi(i)} \succeq_\star t_i$ for all i. By the induction hypothesis $f(\vec{s}) \Rightarrow_{\texttt{equiv}} g(s_{\pi(1)}, \ldots, s_{\pi(n)}) \Rightarrow^*_\star g(t_1, \ldots, t_n)$.

(Put) $f(\vec{s}) \Rightarrow_{\texttt{put}} f^\star(\vec{s}) \Rightarrow^*_\star t$ by the induction hypothesis.

(Select) Let $s = f^\star(\vec{s}) : \tau$ and $s_i : \sigma_1 \to \ldots \to \sigma_n \to \tau$. Suppose that $s_i = \lambda x_1 \ldots x_k.q$ and either $k = n$, or $k < n$ and q is not an abstraction. We have $s \succeq_\star t$ because $s_i\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle \succeq_\star t$, with exactly n repetitions of $f^\star(\vec{s})$ (which may have various types). By the induction hypothesis we are done if $s \Rightarrow^*_\star s_i\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle$.

If $k = n$, then indeed $s \Rightarrow_{\texttt{select}} q[x_1 := f^\star(\vec{s}), \ldots, x_n := f^\star(\vec{s})]$, which is exactly $s_i\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle$.

If $k < n$, then $s_i\langle\ldots\rangle$ is only well-defined if q has the form $g(\vec{t})$ or $g^\star(\vec{t})$, and $s_i\langle f^\star(\vec{s}),\ldots,f^\star(\vec{s})\rangle = g^\star(\vec{t\gamma}, f^\star_{\vec{\alpha},\rho_{k+1}}(\vec{s}),\ldots,f^\star_{\vec{\alpha},\rho_n}(\vec{s}))$, where γ is the substitution $[x_1 := f^\star_{\vec{\alpha},\rho_1}(\vec{s}),\ldots,x_k := f^\star_{\vec{\alpha},\rho_k}(\vec{s})]$. Writing $\delta := [x_1 := f^\star_{\vec{\alpha},\rho_1}(\vec{s}),\ldots,x_n := f^\star_{\vec{\alpha},\rho_n}(\vec{s})]$ for fresh variables $x_{k+1},\ldots,x_n$, we can rewrite this term as $g^\star(\vec{t},x_{k+1},\ldots,x_n)\delta$.

Since $s_i = \lambda x_1,\ldots,x_k.q \Rightarrow^*_{\mathtt{abs}} \lambda x_1,\ldots,x_n.g^\star(\vec{t},x_{k+1},\ldots,x_n)$, we indeed have that $s \Rightarrow^*_{\mathtt{abs}} \cdot \Rightarrow_{\mathtt{select}} s_i\langle f^\star(\vec{s}),\ldots,f^\star(\vec{s})\rangle$.

(F-Abs) $f^\star_{\vec{\sigma},\tau\to\rho}(\vec{s}) \Rightarrow_{abs} \lambda x.f^\star_{\vec{\sigma},\tau,\rho}(\vec{s},x)$, which by induction hypothesis $\Rightarrow^*_\star \lambda x.t$.

(Copy) $f^\star(\vec{s}) \Rightarrow_{\mathtt{copy}} g(f^\star(\vec{s}),\ldots,f^\star(\vec{s}))$, and each $f^\star(\vec{s}) \Rightarrow^*_\star t_i$ by the induction hypothesis.

(Stat) Here, $f^\star(\vec{s},\vec{q}) \succeq_\star g(\vec{t})$ because $f \approx g$, $|\vec{s}| = ar(f)$, $\vec{s} \succ_{\star gstat(f)} \vec{t}$ and $f^\star(\vec{s}) \succeq_\star t_i$ for all i. Consider $stat(f)$.

If $stat(f) = Lex$, then there is some i such that $s_1 \succeq_\star t_1,\ldots,s_{i-1} \succeq_\star t_{i-1}$, and either $|\vec{t}| = i-1 < |\vec{s}|$, or $s_i \succ_\star t_i$. In the first case, $f^\star(\vec{s},\vec{t}) \Rightarrow_{\mathtt{lex}} g(s_1,\ldots,s_{i-1}) \Rightarrow^*_\star g(t_1,\ldots,t_{i-1})$ by the induction hypothesis on the inequalities $s_j \succeq_\star t_j$. In the second case, $f^\star(\vec{s},\vec{t}) \Rightarrow_{\mathtt{lex}} g(s_1,\ldots,s_{i-1},s^\star_i, f^\star(\vec{s},\vec{t}),\ldots,f^\star(\vec{s},\vec{t}))$. By the induction hypothesis, $s_j \Rightarrow^*_\star t_j$ for $j < i$. $s^\star_i \Rightarrow^*_\star t_i$ (since $s_i \succ_\star t_i$ means that $s^\star_i \succeq_\star t_i$), and $f^\star(\vec{s},\vec{t}) \Rightarrow^*_\star t_j$ for $j > i$. Thus, $f^\star(\vec{s},\vec{t}) \Rightarrow_{\mathtt{lex}} \cdot \Rightarrow^*_\star g(\vec{t})$ as required.

If $stat(f) = Mul$, then let $|\vec{s}| = n$, $|\vec{t}| = m$. There is a function $\pi : \{1,\ldots,m\} \to \{1,\ldots,n\}$ and disjoint sets A, B such that:

- $\{1,\ldots,n\} = A \cup B$ with B non-empty;
- for every element j of A there is exactly one i such that $\pi(i) = j$;
- if $\pi(i) \in A$ then $s_{\pi(i)} \succeq_\star t_i$
- if $\pi(i) \in B$ then $s^\star_{\pi(i)} \succeq_\star t_i$.

Let $u_1,\ldots,u_m$ be defined as follows: if $\pi(i) \in A$ then $u_i := s_{\pi(i)}$, otherwise $u_i := s^\star_{\pi(i)}$. By the induction hypothesis, $g(\vec{u}) \Rightarrow^*_\star g(\vec{t})$. If, moreover, $f^\star(\vec{s},\vec{q}) \Rightarrow_{\mathtt{mul}} g(\vec{u})$ we are done. To see that this is the case, let us construct the exact instance of the `mul` rule which can be used for this reduction.

Let $l := f^\star(l_1,\ldots,l_n,Z_{n+1},\ldots,Z_k)$, where $l_i := Z_i$ is a meta-variable if either $i \in A$, or there is no j with $\pi(j) = i$. Otherwise, s_i is markable and thus has the form $\lambda\vec{x}.h(\vec{v})$, so let $l_i := \lambda\vec{x}.h(\vec{Z}_i(\vec{x}))$, where $\vec{Z}_i(\vec{x})$ should be read as $(Z_{i,1}(\vec{x}),\ldots,Z_{i,p}(\vec{x}))$. Let $r := g(r_1,\ldots,r_m)$, where $r_i = Z_{\pi(i)}$ if $i \in A$, and otherwise $r_i = l^\star_{\pi(i)}$. This is well-defined since in those cases $l_{\pi(i)}$ is markable.

Certainly $l \Rightarrow r$ is one of the `mul` rules:

- not $\{\{\vec{l}\}\} = \{\{\vec{r}\}\}$, because if all $\pi(i) \in A$ then $m < n$ because B is non-empty and π is injective on A
- each l_i which is a meta-variable occurs at most once in $\{\{\vec{r}\}\}$, for these i are either in A, where π is injective, or l_i is unused altogether

Moreover, there is a substitution γ such that $l\gamma = f^\star(s_1, \ldots, s_n, q_1, \ldots, q_k)$ and $r\gamma = g(q_1, \ldots, q_m)$. This is obvious by the respective definitions.

(Bot) $s \Rightarrow_{\mathtt{bot}} \bot_\sigma$.

□

It is harder to see that also $\Rightarrow_\star^*$ is included in $\succeq_\star$, since the single-step relations of $\Rightarrow_\star^*$ are not included in $\succeq_\star$: $\succeq_\star$ is a relation on terms over $\mathcal{F}$, and without having a reduction strategy yet, it is difficulty to analyse, exactly, what $\Rightarrow_\star^*$ is.

In order to analyse $\succeq_\star$ in the same context as $\Rightarrow_\star$, we shall therefore consider an extension, which coincides with $\succeq_\star$ on terms over $\mathcal{F}$. Let $s \succ_\star^e t$ if $s^\star \succeq_\star^e t$, and let $\succeq_\star^e$ be recursively defined by the clauses from Definition 5.19, and in addition:

(In)	$f^\star(\vec{s})$	$\succeq_\star^e$	$g^\star(\vec{t})$	if	$f \approx g$ and $ar(f) = ar(g) = k$ and $(s_1, \ldots, s_k) \succeq^e_{\star gstat(f)} (t_1, \ldots, t_k)$ and $\lvert\vec{s}\rvert = \lvert\vec{t}\rvert$ and $s_i \succeq_\star^e t_i$ if $i > k$
(Marked1)	$f^\star(\vec{s})$	$\succeq_\star^e$	$g^\star(\vec{t})$	if	$f \blacktriangleright g$ and $f^\star(\vec{s}) \succeq_\star^e t_i$ for all i
(Marked2)	$f^\star(\vec{s})$	$\succeq_\star^e$	$g^\star(\vec{t})$	if	$f \approx g$ and $f^\star(\vec{s}) \succeq_\star^e t_i$ for all i and $ar(f) = k,\ ar(g) = m$ and $(s_1, \ldots, s_k) \succ^e_{\star gstat(f)} (t_1, \ldots, t_m)$
(Drop)	$f^\star(\vec{s}, q)$	$\succeq_\star^e$	t	if	$f^\star(\vec{s}) \succeq_\star^e t$ and $\lvert\vec{s}\rvert \geq ar(f)$
(Swap)	$f^\star(s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n) \succeq_\star^e t$ if $ar(f) < i$ and $f^\star(s_1, \ldots, s_j, \ldots, s_i, \ldots, s_n) \succeq_\star^e t$				
(Args)	$f^\star(\vec{s})$	$\succeq_\star^e$	t	if	$f^\star(\vec{s}, q_1, \ldots, q_n) \succeq_\star^e t$ where each q_i has the form $f^\star(s_1, \ldots, s_k)$ with $ar(f) \leq k \leq \lvert\vec{s}\rvert$
(Select)	$f^\star(\vec{s})$	$\succeq_\star$	t	if	$s_i\langle q_1, \ldots, q_m\rangle \succeq_\star t$, where each q_j has the form $f^\star(s_1, \ldots, s_k)$, with $ar(f) \leq k \leq \lvert\vec{s}\rvert$

Note that the new (Select) clause includes the old one, so we do not have to consider the original (Select) clause when proving properties about $\succeq_\star^e$.

Let us start by making some straightforward observations about $\succeq_\star^e$.

Lemma 5.24. *For all meta-terms s, t over $\mathcal{F}^\star$:*

1. *if $s \succ_\star^e t$, then $s \succeq_\star^e t$;*
2. *$\succeq_\star^e$ is reflexive;*
3. *if $f^\star_{\vec{\sigma},\tau}(\vec{s}) \succeq_\star^e g^\star_{\vec{\rho},\tau}(\vec{s})$ by clause (Marked1), (Marked2) or (In), then this holds for any τ.*

Proof. If $t = \bot_\sigma$ then (1) holds by (Bot), otherwise $s = \lambda\vec{x}.q$ with q an unmarked functional term, and $t = \lambda\vec{x}.u$ with $q^\star \succeq^e_\star u$; in this case we conclude with (Abs) and (Put).

(2) holds by induction, using clauses (Var), (Abs), (Meta), (Fun) and (In).

(3) is obvious from the definition of the relevant clauses. □

The important thing to know is the following: $\succeq^e_\star$ and $\succ^e_\star$ define exactly the same relation as $\succeq_\star$ and $\succ^e_\star$ on meta-terms over $\mathcal{F}$. This is demonstrated by the following lemma, since obviously $\succeq_\star$ is included in $\succeq^e_\star$.

Lemma 5.25. *If s, t are meta-terms over $\mathcal{F}$ and $s \succeq^e_\star t$ or $s \succ^e_\star t$, then also $s \succeq_\star t$ or $s \succ_\star t$, respectively.*

Proof. Define the relation $\sharp$ on terms over $\mathcal{F}^\star$ as follows:

- $x\sharp x$ if x is a variable
- $\lambda x.s\sharp\lambda x.t$ if $s\sharp t$
- $Z(s_1,\dots,s_n)\sharp Z(t_1,\dots,t_n)$ if each $s_i\sharp t_i$
- $f(s_1,\dots,s_n)\sharp f(t_1,\dots,t_n)$ if each $s_i\sharp t_i$ and $f \in \mathcal{F}$
- $f^\star_{\vec{\sigma},\rho}(s_1,\dots,s_n,t_1,\dots,t_m)\sharp f^\star_{\vec{\tau},\rho}(q_1,\dots,q_n,u_1,\dots,u_k)$ if $ar(f) = n$ and each $s_i\sharp q_i$ and for each u_i: either some $t_j\sharp u_i$, or $f^\star(\vec{s},\vec{t})\sharp u_i$

We easily derive:

(I) if $q\sharp u$ and both sides are markable, then $q^\star\sharp u^\star$;

(II) if $f^\star_{\vec{\sigma},\rho}(\vec{s})\sharp f^\star_{\vec{\tau},\rho}(\vec{t})$, then this holds for any ρ;

(III) if $q\sharp u$ and $v\sharp w$, then also $q[x := v]\sharp u[x := w]$ (where x is a variable).

The last statement holds by induction on the definition of $\sharp$.

We will see: if s and q are terms over $\mathcal{F}^\star$ and t is a term over $\mathcal{F}$, and if $q\sharp s$ and either $s \succeq^e_\star t$ or $s \succ^e_\star t$, then also $q \succeq_\star t$ or $q \succ_\star t$ respectively. To this end, we will employ induction on the derivation of $s \succ^e_\star t$ or $s \succeq^e_\star t$. Considering all the clauses, if $s'\ R\ t'$ is used in the derivation of $s\ R\ t$, then t' is a subterm of t, and consequently will also be unmarked. Hence, we do not have to check in every case whether the induction hypothesis is applicable.

First, if $s \succ^e_\star t$, this holds because $s^\star \succeq^e_\star t$. By (I) also $q^\star\sharp s^\star$, and by the induction hypothesis therefore $q \succ_\star t$.

Alternatively, $s \succeq^e_\star t$; consider the clause used to derive this.

(Var) If $s = t$ is a variable, then q must be the same variable, so $q \succeq_\star t$ by the same clause.

(Abs), (Meta), (Fun) In these cases we can trivially use the induction hypothesis. The hardest is (Fun), where $s = f(s_1, \ldots, s_n),\ t = g(t_1, \ldots, t_n)$ and $f \approx g$ and $\vec{s} \succeq^e_{\star gstat(f)} \vec{t}$. By definition of $\sharp$ we have $q = f(q_1, \ldots, q_n)$ where each $q_i \sharp s_i$. If $stat(f) = Lex$, then we have for all i that $q_i \sharp s_i \succeq^e_\star t_i$, so by the induction hypothesis $q_i \succeq_\star t_i$. If $stat(f) = Mul$, then there is a permutation π such that for all i: $q_{\pi(i)} \sharp s_{\pi(i)} \succeq^e_\star t_i$, so by the induction hypothesis $q_{\pi(i)} \succeq_\star t_i$. Either way, $\vec{q} \succeq_{\star gstat(f)} \vec{t}$, so $q \succeq_\star t$ by (Fun).

(Put) As the $\succ^e_\star$ case described above.

(Select) $s = f^\star_{\sigma_1,\ldots,\sigma_n,\tau}(\vec{s})$ and $s_i\langle w_1, \ldots, w_m\rangle \succeq^e_\star t$, where $s_i : \rho_1 \to \ldots \to \rho_m \to \tau$ and each $w_j = f^\star_{\sigma_1,\ldots,\sigma_{k_j},\rho_j}(s_1, \ldots, s_{ar(f)+k_j})$ for some k_j. We have $q = f^\star_{\vec{\alpha},\tau}(\vec{q})$ and by the definition of $\sharp$, we must have $f^\star_{\vec{\alpha},\rho_j}(\vec{q}) \sharp w_j$ for all j.

There are two possibilities for s_i: either $i > ar(f)$ and $f^\star_{\vec{\sigma},\rho_1\to\ldots\to\rho_m\to\tau}(\vec{q}) \sharp s_i$, or $q_j \sharp s_i$ for some j.

In the first case, we can write $s_i = f^\star(u_1, \ldots, u_{ar(f)}, \vec{v})$ where each $q_j \sharp u_j$ for $j \leq ar(f)$, and since $f^\star_{\vec{\alpha},\rho_k}(\vec{q}) \sharp w_k$ for all k we see that also $q \sharp f^\star(\vec{u}, \vec{v}, \vec{w}) = s_i\langle w_1, \ldots, w_m\rangle$. Now we can apply the induction hypothesis, and obtain that $q \succeq_\star t$.

In the second case, suppose that $q' \sharp s'$ and $v_1 \sharp w_1, \ldots, v_m \sharp w_m$ together imply that $q'\langle v_1, \ldots, v_m\rangle \sharp s'\langle w_1, \ldots, w_m\rangle$. Then we can conclude by the induction hypothesis and the original (Select) rule, since $q_j \sharp s_i$ and $f^\star_{\vec{\alpha},\rho_j}(\vec{q}) \sharp w_j$ for all j. We prove this by induction on m. If $m = 0$ the result is obvious, and also if q' and s' are functional terms (with or without star) the definition of $\sharp$ quickly gives the required result. Otherwise, $s' = \lambda x.s''$ and $q' = \lambda x.q''$ and $q'\langle\vec{v}\rangle = q''[x := v_1]\langle v_2, \ldots, v_m\rangle$ and $s'\langle\vec{w}\rangle = s''[x := w_1]\langle w_2, \ldots, w_m\rangle$. We have the required result by the induction hypothesis and (III).

(F-Abs) $s = f^\star_{\vec{\sigma},\tau\to\rho}(\vec{s})$ and $t = \lambda x.t'$ with $f^\star_{\vec{\sigma},\tau,\rho}(\vec{s}, x) \succeq^e_\star t'$. We have $q = f^\star_{\vec{\alpha},\tau\to\rho}(\vec{q})$. Since also $f^\star_{\vec{\alpha},\tau,\rho}(\vec{q}, x) \sharp f^\star_{\vec{\sigma},\tau,\rho}(\vec{s}, x)$, the induction hypothesis and clause (F-Abs) give that $q \succeq_\star t$.

(Copy) $s = f^\star_{\vec{\sigma},\tau}(\vec{s})$ and $t = g(\vec{t})$ where $f \blacktriangleright g$ and $f^\star_{\vec{\sigma},\rho_i}(\vec{s}) \succeq^e_\star t_i$ for all i. Writing $q = f^\star_{\vec{\alpha},\tau}(\vec{q})$ the induction hypothesis and (II) provide that also $f^\star_{\vec{\alpha},\rho_i}(\vec{q}) \succeq_\star t_i$. Therefore we can use (Copy) to obtain $q \succeq_\star t$.

(Stat) $s = f^\star_{\vec{\sigma},\tau}(\vec{s})$ and $t = g(\vec{t})$ where $f \approx g$ and $f^\star_{\vec{\sigma},\rho_i}(\vec{s}) \succeq^e_\star t_i$ for all i, and $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star gstat(f)} \vec{t}$. As in the (Copy) case we can write $q = f^\star_{\vec{\alpha},\tau}(\vec{q})$ and have $f^\star_{\vec{\rho_i}}(\vec{q}) \succeq_\star t_i$ for all i. Moreover, $q_i \sharp s_i$ for all $i \leq ar(f)$. Consequently, if $s_i \succeq^e_\star t_j$ or $s_i \succ^e_\star t_j$ is used in the derivation of the subclause $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star gstat(f)} \vec{t}$, then we also have $q_i \succeq_\star t_j$ or $q_i \succ_\star t_j$ respectively by the induction hypothesis. Thus, $(q_1, \ldots, q_{ar(f)}) \succ_{\star gstat(f)} \vec{t}$, and we can complete with (Stat).

(Bot) $t = \bot_\sigma$, so also $q \succeq^e_\star t$ by (Bot).

(In), (Marked1), (Marked2) None of these cases could be applied, since t is star-free.

(Drop), (Swap), (Args) In each of these cases, $s \succeq^e_\star t$ because $s' \succeq^e_\star t$ for some s' such that $s \sharp s'$. Every time it is evident that also $q \sharp s'$, so by the induction hypothesis $s' \succeq^e_\star t$ implies $q \succeq_\star s'$.

□

Thus, the original and extended version of $\succeq_\star$ coincide on terms over $\mathcal{F}$. However, on terms over $\mathcal{F}^\star$ they differ, and it is this which makes the extended version such a useful aid, both in proving transitivity of $\succeq_\star$ (which we will do in Section 5.4.2), and to see that $\Rightarrow^*_\star$ is included in $\succeq_\star$. A first step towards this is given by the following lemma:

Lemma 5.26. *$\Rightarrow_\star$ is included in $\succeq^e_\star$.*

Proof. The relation $\succeq_\star$ is monotonic over meta-terms by clauses (Abs), (Meta), (Fun) and (In). Thus, we only need to see that $s \succeq_\star t$ if $s \Rightarrow_\star t$ by a topmost step. Consider the rule that was used to derive $s \Rightarrow_\star t$.

put $s = f(\vec{s})$ and $t = f^\star(\vec{s})$. Then $s \succeq^e_\star t$ by clause (Put) and reflexivity of $\succeq^e_\star$.

select $s = f^\star(\vec{s})$ where $s_i = \lambda\vec{x}.q$, and $t = q[x_1 := f^\star(\vec{s}), \ldots, x_n := f^\star(\vec{s})] = s_i\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle$. Hence, $s \succeq^e_\star t$ by (Select) and reflexivity.

copy $s = f^\star(\vec{s})$ and $t = g(f^\star(\vec{s}), \ldots, f^\star(\vec{s}))$. Since each $f^\star_{\vec{\sigma},\rho_i}(\vec{s}) \succeq^e_\star f^\star_{\vec{\sigma},\rho_i}(\vec{s})$ by reflexivity, we have $s \succeq^e_\star t$ by (Copy).

lex $s = f^\star(\vec{s})$ and $t = g(s_1, \ldots, s_{i-1}, s_i^\star, f^\star(\vec{s}), \ldots, f^\star(\vec{s}))$ with $f \approx g$ and $i \leq ar(f)$. Whether $i \geq ar(g)$ or not, $[s_1, \ldots, s_n] \succeq^e_{\star\, gLex} [s_1, \ldots, s_{i-1}, s_i^\star, f^\star(\vec{s}), \ldots, f^\star(\vec{s})] =: [\vec{t}]$, either because the latter is a strict initial subsequence of the former, or because $s_i \succ^e_\star s_i^\star = t_i$.

Moreover, for every j we have $f^\star(\vec{s}) \succeq^e_\star t_j$: if $j > i$ this holds by reflexivity of $\succeq^e_\star$, if $j < i$ by the (Select) clause, and if $j = i$ by the combination of (Select) and the observation that $s_i \succeq^e_\star s_i^\star$ and Lemma 5.24(1). Thus, $s \succeq^e_\star t$ by clause (Stat).

mul $s = f^\star(\vec{s})$ and $t = g(\vec{t})$ where, much like we have seen in the proof of Lemma 5.13, $\{\{s_1, \ldots, s_{ar(f)}\}\} \sqsupset_{Mul} \{\{\vec{t}\}\}$ if $\sqsupset$ is the relation $q \sqsupset q^\star$ (and the corresponding $\sqsupseteq$ its reflexive closure). Since this $\sqsupset$ is a subrelation of $\succ^e_\star$, we also have that $s \succeq^e_\star t$ by clause (Stat) since, as argued in the case of `lex`, also $f^\star(\vec{s}) \succeq^e_\star t_i$ for all i by the (Select) rule, possibly combined with (Put) and (Abs).

abs $s = f^\star(\vec{s}) \succeq^e_\star \lambda x.f^\star(\vec{s}, x)$ by (F-Abs) and reflexivity.

equiv $s = f(\vec{s})$, $t = g(\vec{s})$ and $f \approx g$, $\vec{s} =_{gstat(f)} \vec{t}$. By reflexivity of $\succeq_\star$ this means $\vec{s} \succeq_{\star gstat(f)} \vec{t}$, so $s \succeq_\star t$ by (Fun).

bot $s : \sigma$ and $t = \bot_\sigma$. By (Bot) also $s \succeq^e_\star t$.

□

Lemmas 5.23 and 5.26, together with the transitivity result of the next section, show that $\succeq_\star, \succeq^e_\star$ and $\Rightarrow^*_\star$ define the same relation on meta-terms over $\mathcal{F}$. However, $\succeq^e_\star$ should not be considered of interest in and of itself; this extension is merely meant as a way to prove properties about $\succeq_\star$.

5.4.2 Transitivity

Finally, we will see that $\succeq^e_\star$ is transitive (and therefore $\succeq_\star$ is transitive as well!). This is both of an important theoretical interest (as $\succeq_\star$ must be transitive to be an ordering), and essential for $\succeq_\star$ and $\Rightarrow^*_\star$ to define the same relation. This result does not follow immediately; we will need several lemmas to use in the proof.

Lemma 5.27. *Let γ, δ be substitutions on the same domain such that always $\gamma(x) \succeq^e_\star \delta(x)$, and suppose $s \succeq^e_\star t$. Then also $s\gamma \succeq^e_\star t\delta$.*

Proof. By induction first on whether or not $A := dom(\gamma)$ contains meta-variables, second on the derivation of $s \succeq^e_\star t$; consider the clause used to derive this.

(Var) $s = t = x$. If $x \in A$ then $s\gamma = \gamma(x) \succeq^e_\star \delta(x) = t\delta$ by assumption. If $x \notin A$, then $s\gamma = s \succeq^e_\star t = t\gamma$ by (Var).

(Meta) $s = Z(s_1, \ldots, s_n)$ and $t = Z(t_1, \ldots, t_n)$. If $Z \notin A$ then $s\gamma = Z(s_1\gamma, \ldots, s_n\gamma)$ and $t\gamma = Z(t_1\delta, \ldots, t_n\delta)$. By the second induction hypothesis each $s_i\gamma \succeq^e_\star t_i\delta$, so $s\gamma \succeq^e_\star t\delta$ by (Meta). If, however, $Z \in A$, then let $\gamma(Z) = \lambda x_1 \ldots x_n.q$ and $\delta(Z) = \lambda x_1 \ldots x_n.u$ with $q \succeq^e_\star u$. Then $s\gamma = q[\vec{x} := \vec{s\gamma}]$ and $t\delta = u[\vec{x} := \vec{t\delta}]$. By the second induction hypothesis each $s_i\gamma \succeq^e_\star t_i\delta$, and by the first induction hypothesis therefore $s\gamma \succeq^e_\star t\delta$.

All other cases are immediately obvious with the second induction hypothesis. We consider for instance **(Select)**, the most complicated of the lot: $s = f^*(\vec{s})$ and some $s_i\langle q_1, \ldots, q_n\rangle \succeq^e_\star t$, where each q_j has the form $f^*(s_1, \ldots, s_{k_j})$. By the induction hypothesis $s_i\langle q_1, \ldots, q_n\rangle\gamma \succeq^e_\star t\gamma$. Whether s_i is an abstraction or a functional term, $s_i\langle q_1, \ldots, q_m\rangle\gamma = (s_i\gamma)\langle q_1\gamma, \ldots, q_n\gamma\rangle$ (as can easily be seen with induction on n), and if q_j is obtained from $f^*(\vec{s})$ by dropping a number of arguments, then $q_j\gamma$ is obtained from $f^*(\vec{s})\gamma$ in the same way. Therefore also $f^*(\vec{s})\gamma = f^*(\vec{s\gamma}) \succeq^e_\star t\gamma$ by (Select). □

Lemma 5.28. *If $s \succeq^e_\star t$ and t is markable, then either s is markable or $s = q^\star$. In the first case, $s^\star \succeq^e_\star t^\star$, in the second, $s \succeq^e_\star t^\star$.*

Note that this lemma is a special case of transitivity, where the second step merely adds a star.

Proof. The lemma holds by induction on the derivation of $s \succeq^e_\star t$, assuming that t is indeed markable. Consider the clause used to derive $s \succeq^e_\star t$. It could not be one of (Var), (Meta), (Bot), (Marked1), (Marked2) or (In), for in these cases t is not markable.

(Abs) $s = \lambda x.s'$ and $t = \lambda x.t'$. Since $t^\star = \lambda x.t'^\star$, and if s is markable then $s^\star = \lambda x.s'^\star$, we can complete with the induction hypothesis.

(Fun) $s = f(\vec{s}) \succeq^e_\star t = g(\vec{s})$ because $\vec{s} \succeq^e_{\star gstat(f)} \vec{t}$, so by (In) also $s^\star \succeq^e_\star t^\star$.

(Put) $s \succeq^e_\star t$ because $s^\star \succeq^e_\star t$. By the induction hypothesis also $s^\star \succeq^e_\star t^\star$.

(Select), (Drop), (Swap), (Args) $s \succeq^e_\star t$ because $q \succeq^e_\star t$ for some meta-term q. By the induction hypothesis, either $q^\star \succeq^e_\star t$ (if $q^\star$ is markable), or $q \succeq^e_\star t$ (if not). By Lemma 5.24(1) we have $q \succeq^e_\star t$ in both cases. Thus, $s \succeq^e_\star t$ by the same clause (which suffices, because s is already marked).

(F-Abs) $s \succeq^e_\star t$ because $s' \succeq^e_\star t$ for some marked meta-term s'. By the induction hypothesis also $s' \succeq^e_\star t^\star$, so $s \succeq^e_\star t^\star$ by (F-Abs) (which suffices because s is already marked).

(Copy), (Stat) s is already marked, and $s \succeq^e_\star t^\star$ by (Marked1) or (Marked2).

□

Lemma 5.29. *If $s \succeq^e_\star t$ and $q_1 \succeq^e_\star u_1, \ldots, q_n \succeq^e_\star u_n$, then $s\langle\vec{q}\rangle \succeq^e_\star t\langle\vec{u}\rangle$ if the latter is defined.*

Proof. If $n = 0$, then $s\langle\vec{q}\rangle = s \succeq^e_\star t = t\langle\vec{u}\rangle$ by assumption, so this case we need not consider. Suppose the lemma holds for $n = 1$. Then it holds for all n, with induction on n, for by the induction hypothesis, $s\langle q_1\rangle\langle q_2, \ldots, q_n\rangle \succeq^e_\star t\langle u_1\rangle\langle u_2, \ldots, u_n\rangle$. Since both sides are defined, and investigating the definition of $\langle\cdots\rangle$, we can conclude that $s\langle\vec{q}\rangle \succeq^e_\star t\langle\vec{u}\rangle$.

Thus, we only need to show that if $s \succeq^e_\star t$ and $q \succeq^e_\star u$ then also $s(q) \succeq^e_\star t(u)$ if the latter is defined. We use induction on the derivation $s \succeq^e_\star t$, and use a case analysis on the clause used to derive this.

(Var), (Meta), (Bot) Not applicable, since $t\langle\vec{u}\rangle$ is not defined in these cases.

(Abs) $s = \lambda x.s'$ and $t = \lambda x.t'$ with $s' \succeq^e_\star t'$. By Lemma 5.27 $s(q) = s'[x := q] \succeq^e_\star t'[x := u] = t(q)$.

(Fun), (In) $s = f^{(\star)}(\vec{s})$ and $t = g^{(\star)}(\vec{t})$ with $s_i \succeq^e_\star t_i$ for $i > ar(f)$, and $(s_1, \ldots, s_{gstat(f)}) \succeq^e_{\star_{\mathrm{stat}_f}} (t_1, \ldots, t_{gstat(f)})$. Using (In) and the assumption that $q \succeq^e_\star u$, we have $s\langle q\rangle = f^\star(\vec{s}, q) \succeq^e_\star g^\star(\vec{t}, u)$.

(Put) $s = f(\vec{s}) \succeq^e_\star t$ because $f^\star(\vec{s}) \succeq^e_\star t$. By the induction hypothesis $s\langle q\rangle = f^\star(\vec{s}, q) = f^\star(\vec{s})\langle q\rangle \succeq^e_\star t\langle u\rangle$ (without using (Put)).

(Select) $s = f^\star(\vec{s})$ and $s_i\langle v_1, \ldots, v_n\rangle \succeq^e_\star t$ for some $v_1, \ldots, v_n$ obtained from $f^\star(\vec{s})$ by dropping arguments. Because $q \succeq^e_\star u$, we have $f^\star(\vec{s}, q) \succeq^e_\star u$ by (Select), so the induction hypothesis provides that $s_i\langle q_1, \ldots, q_n, f^\star(\vec{s}, q)\rangle = s_i\langle q_1, \ldots, q_n\rangle\langle f^\star(\vec{s}, q)\rangle \succeq^e_\star t\langle u\rangle$. Since each q_j is obtained from $f^\star(\vec{s})$ by dropping arguments, and $f^\star(\vec{s})$ is obtained from $f^\star(\vec{s}, q)$ in the same way, we have $s\langle q\rangle = f^\star(\vec{s}, q) \succeq^e_\star t\langle u\rangle$ by (Select).

(F-Abs) $s = f^\star(\vec{s}) \succeq^e_\star \lambda x.t' = t$ because $f^\star(\vec{s}, x) \succeq^e_\star t'$. By Lemma 5.27 $s\langle q\rangle = f^\star(\vec{s}, q) = f^\star(\vec{s}, x)[x := q] \succeq^e_\star t'[x := q] = t\langle q\rangle$.

(Copy), (Marked1) $s = f^\star(\vec{s})$, $t = g^{(\star)}(\vec{t})$, $f \blacktriangleright g$ and $f^\star(\vec{s}) \succeq^e_\star t_i$ for all i. Also always $f^\star(\vec{s}, q) \succeq^e_\star t_i$, by (Drop), and $f^\star(\vec{s}, q) \succeq^e_\star u$ as well by (Select) (selecting q). Thus, $s\langle q\rangle = f^\star(\vec{s}, q) \succeq^e_\star g^\star(\vec{t}, u) = t\langle u\rangle$ by clause (Marked1).

(Stat), (Marked2) $s = f^\star(\vec{s})$ and $t = g^{(\star)}(\vec{t})$ with $f \approx g$ and $f^\star(\vec{s}) \succeq^e_\star t_i$ for each i, and $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star_{gstat(f)}} (t_1, \ldots, t_{ar(g)})$. By (Drop) also $f^\star(\vec{s}, q) \succeq^e_\star t_i$, and $f^\star(\vec{s}, q) \succeq^e_\star u$ by (Select). Thus $s\langle q\rangle = f^\star(\vec{s}, q) \succeq^e_\star g^\star(\vec{t}, u) = t\langle u\rangle$ by (Marked2).

(Drop) $s = f^\star(\vec{s}, v) \succeq^e_\star t$ because $f^\star(\vec{s}) \succeq^e_\star t$. By the induction hypothesis $f^\star(\vec{s}, q) \succeq^e_\star t(u)$, so by (Swap) $s\langle q\rangle = f^\star(\vec{s}, v, q) \succeq^e_\star t\langle u\rangle$ because $f^\star(\vec{s}, q, v) \succeq^e_\star t\langle u\rangle$ by (Drop).

(Swap) $s = f^\star(s_1, \ldots, s_i, \ldots, s_j, \ldots, s_n) \succeq^e_\star t$ because $f^\star(s_1, \ldots, s_j, \ldots, s_i, \ldots, s_n) \succeq^e_\star t_m$. By the induction hypothesis $f^\star(s_1, \ldots, s_j, \ldots, s_i, \ldots, s_n, q) \succeq^e_\star t\langle u\rangle$, so by (Swap) again $s\langle q\rangle \succeq^e_\star t\langle u\rangle$.

(Args) $s = f^\star(\vec{s}) \succeq^e_\star t$ because $f^\star(\vec{s}, v_1, \ldots, v_n) \succeq^e_\star t$, where each v_i is obtained from $f^\star(\vec{s})$ by removing some arguments. By the induction hypothesis $f^\star(\vec{s}, v_1, \ldots, v_n, q) \succeq^e_\star t(u)$. By (Swap) also $f^\star(\vec{s}, q, \vec{v}) \succeq^e_\star t(u)$. Since all v_j are obtained from $f^\star(\vec{s}, q)$ by removing some arguments (including the new argument q), we thus have $f^\star(\vec{s}, q) \succeq^e_\star t(u)$ by (Args).

□

Lemma 5.30. *If $s \succeq^e_\star f^\star(\vec{s}, t)$ then also $s \succeq^e_\star f^\star(\vec{s})$ (if well-defined).*

As with Lemma 5.29, this is a special case of the transitivity proof, where the second step is a topmost (Drop) application.

Proof. We use induction on the derivation of $s \succeq^e_\star f^\star(\vec{s}, t)$ and case analysis on the rule used to derive it. If this is any of (Star), (Select), (Args), (Drop) or (Swap), then $s \succeq^e_\star f^\star(\vec{s}, t)$ because some $q \succeq^e_\star f^\star(\vec{s}, t)$, and by the induction hypothesis also $q \succeq^e_\star f^\star(\vec{s})$ and thus $s \succeq^e_\star f^\star(\vec{s})$ by the same clause. The clause cannot be any of (Var), (Abs), (Meta), (Fun), (F-Abs), (Copy), (Stat) or (Bot), since the right-hand side does not match. Only (In), (Marked1) and (Marked2) remain.

If $s \succeq^e_\star f^\star(\vec{s}, t)$ by (In), write $s = g^\star(\vec{q}, u)$ where $ar(g) = ar(f) \leq |\vec{s}|$ and $q_i \succeq^e_\star s_i$ for $i > ar(g)$ and $u \succeq^e_\star t$. Then also $g^\star(\vec{q}) \succeq^e_\star f^*(\vec{s})$ by the same rule, and therefore $s \succeq^e_\star f^\star(\vec{s})$ by (Drop).

If $s \succeq^e_\star f^\star(\vec{s}, t)$ by (Marked1), then by the same clause $s \succeq^e_\star f^\star(\vec{s})$.

If $s \succeq^e_\star f^\star(\vec{s}, t)$ by (Marked2), then note that $|\vec{s}| \geq ar(f)$ because $f^\star(\vec{s})$ is well-defined. Thus, t is an optional argument, and also $s \succeq^e_\star f^\star(\vec{s})$ by (Marked2). □

Lemmas 5.24–5.30 provide all the context we need to plunge, at last, into the proof of transitivity.

Theorem 5.31. *If $s \succeq^e_\star t \succeq^e_\star q$ then $s \succeq^e_\star q$.*

Proof. Given $s \succeq^e_\star t \succeq^e_\star q$, we use induction on the derivation of $t \succeq^e_\star q$ first (IH1), and the derivation of $s \succeq^e_\star t$ second. If either clause is (Bot), then $q = \bot_\sigma$ and definitely $s \succeq^e_\star q$, so assume this clause is not used. Consider the form of s.

- if s is a variable, t and q can only be the same variable; $s \succeq^e_\star q$ by (Var);
- if s is an abstraction $\lambda x.s'$, then $t = \lambda x.t'$ and $q = \lambda x.q'$ with $s' \succeq^e_\star t' \succeq^e_\star q'$; by the first induction hypothesis also $s' \succeq^e_\star q'$, so by (Abs) $s \succeq^e_\star q$;
- if s is a meta-variable application $Z(s_1, \ldots, s_n)$, then $t = Z(\vec{t})$ and $q = Z(\vec{q})$ have the same form, and each $s_i \succeq^e_\star t_i \succeq^e_\star q_i$; again we just use the induction hypothesis, and clause (Meta).

Thus, we can safely assume that s is a (marked or unmarked) functional term.

If $s \succeq^e_\star t$ by one of the clauses (Put), (Select), (Drop), (Swap) or (Args), then $s \succeq^e_\star t$ because some $u \succeq^e_\star t$. By the second induction hypothesis also $u \succeq^e_\star q$, so $s \succeq^e_\star q$ by the same clause. If $s \succeq^e_\star t$ by clause (F-Abs), then $t = \lambda x.t'$ with $u \succeq^e_\star t'$ for some fixed u. In this case q can only have the form $\lambda x.q'$ with $t' \succeq^e_\star q'$, and the induction hypothesis gives $u \succeq^e_\star q'$, and therefore $s \succeq^e_\star \lambda x.q' = q$ by the first induction hypothesis.

What remains for the derivation of $s \succeq^e_\star t$ are the clauses (Fun), (Copy), (Stat), (In), (Marked1) and (Marked2). For these, we must also consider the rule that was used to derive $t \succeq^e_\star q$. The table below shows all relevant combinations of clauses; the indexes are the ones used in the proof below.

	Fun	Copy	Stat	In	Marked1	Marked2
Fun	3	4	5	1	1	1
Put	2	2	2	1	1	1
Select	1	1	1	7	8	8
F-Abs	1	1	1	6	6	6
Copy	1	1	1	10	10	10
Stat	1	1	1	14	10	12
In	1	1	1	13	11	12
Marked1	1	1	1	10	10	10
Marked2	1	1	1	14	10	12
Args	1	1	1	9	9	9
Drop	1	1	1	9	9	9
Swap	1	1	1	9	9	9

Note that, since the right-hand side of each of the clauses (Fun), (Copy), (Stat), (In), (Marked1) and (Marked2) is functional, the second clause in these cases cannot be any of (Var), (Abs) or (Meta), which explains their omission.

1. A clause where the right-hand side is unmarked cannot be followed by one where the left-hand side is marked, and vice versa.

2. If $s \succeq^e_\star t \succeq^e_\star q$ for some unmarked functional meta-term s, and the latter inequality uses clause (Put), then $t \succeq^e_\star q$ holds because $t^\star \succeq^e_\star q$. By IH1, it suffices to see that $s \succeq^e_\star t^\star$, which holds by clause (Put) and Lemma 5.28.

3. If $s \succeq^e_\star t \succeq^e_\star q$ both by (Fun), then write $s = f(\vec{s})$, $t = g(\vec{t})$, $q = h(\vec{q})$ and $f \approx g \approx h$, so also $f \approx h$ (since $\approx$ is an equivalence relation on the symbols). If $stat(f) = Lex$ (so also $stat(g) = Lex$), then we simply have $s_i \succeq^e_\star t_i \succeq^e_\star q_i$ for all i, and therefore $s_i \succeq^e_\star q_i$ by IH1, so $f(\vec{s}) \succeq^e_\star h(\vec{q})$ by (Fun). If $stat(f) = Mul$, then we can find permutations π and ρ of $(1, \ldots, ar(f))$ such that always $s_{\pi(i)} \succeq^e_\star t_i$ and $t_{\rho(i)} \succeq^e_\star q_i$. Combining two permutations gives a permutation again, and also $s_{\pi(\rho(i))} \succeq^e_\star q_i$ by IH1, so $\vec{s} \succeq^e_{\star gMul} \vec{q}$ and therefore $s \succeq^e_\star t$ by (Fun).

4. If $s \succeq^e_\star t$ by (Copy) and $t \succeq^e_\star q$ by (Fun), write $s = f^\star(\vec{s})$, $t = g(\vec{t})$ and $q = h(\vec{q})$ where $f \blacktriangleright g \approx h$, so also $f \blacktriangleright h$. Whatever the status of g, we can find a permutation π (possibly the identity), such that the derivation $t \succeq^e_\star q$ uses that $t_{\pi(i)} \succeq^e_\star q_i$ for all i. Since $f^\star(\vec{s}) \succeq^e_\star t_j$ for all j, IH1 gives that $f^\star(\vec{s}) \succeq^e_\star q_i$ for all i, so $s \succeq^e_\star q$ by (Copy).

5. If $s \succeq^e_\star t$ by (Stat) and $t \succeq^e_\star q$ by (Fun), write $s = f^\star(\vec{s})$, $t = g(\vec{t})$ and $q = h(\vec{q})$ where $f \approx g \approx h$ so also $f \approx h$. As in case 4, $f^\star(\vec{s}) \succeq^e_\star q_i$ for all i. Consider the status of f (which is also the status of g and h).

   a) If $stat(f) = Lex$, then there is some k such that $s_1 \succeq^e_\star t_1, \ldots, s_{k-1} \succeq^e_\star t_{k-1}$ and either $k - 1 = ar(g) < ar(f)$ or $s^\star_k \succeq^e_\star t_k$; furthermore,

all $t_i \succeq^e_\star q_i$. By the first induction hypothesis, $s_1 \succeq^e_\star q_1, \ldots, s_{k-1} \succeq^e_\star q_{k-1}$, $s^\star_k \succeq^e_\star q_k$, so also $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star gLex} \vec{q}$.

b) If $stat(f) = Mul$ then let $\{1, \ldots, ar(f)\} = A \cup B$ with B non-empty. There are a permutation π such that each $t_{\pi(i)} \succeq^e_\star q_i$, and a function ρ such that for every element j of A there is exactly one i such that $\pi(i) = j$, and $s_{\rho(i)} \succeq^e_\star t_i$ if $\rho(i) \in A$, otherwise $s^\star_{\rho(i)} \succeq^e_\star t_i$. By the nature of a permutation, the function $\rho \circ \pi$ also satisfies the requirement that for every $j \in A$ there is a unique i such that $\rho(\pi(i)) = j$. Thus, by IH1, also $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star gMul} (q_1, \ldots, q_{ar(h)})$: if $\rho(\pi(i)) \in A$, then $s_{\rho(\pi(i))} \succeq^e_\star t_{\pi(i)} \succeq^e_\star q_i$ so $s_{\rho(\pi(i))} \succeq^e_\star q_i$, and if $\rho(\pi(i)) \in B$ then $s^\star_{\rho(\pi(i))} \succeq^e_\star t_{\pi(i)} \succeq^e_\star q_i$, so $s_{\rho(\pi(i))} \succ^e_\star q_i$.

6. If $s = f^\star_{\vec{\sigma},\tau\to\rho}(\vec{s}) \succeq^e_\star$ by (In), (Marked1) or (Marked2), and $t \succeq^e_\star q$ by (F-Abs), then $t = g^\star_{\vec{\alpha},\tau\to\rho}(\vec{t})$ and $q = \lambda x.q'$ with $g^\star_{\vec{\alpha},\tau,\rho}(\vec{t}, x) \succeq^e_\star q'$. By (IH1) and the same clause (F-Abs) it suffices to see that $f^\star_{\vec{\sigma},\tau,\rho}(\vec{s}, x) \succeq^e_\star g^\star_{\vec{\alpha},\tau,\rho}(\vec{t}, x)$. But this holds in each of those three cases: if (In) was used because the Lex/Mul relationship on the first $ar(f)$ and $ar(g)$ arguments is not affected by adding an argument, if (Marked2) was used for the same reason and because $f^\star(\vec{s}, x) \succeq^e_\star t_i$ by (Drop) and $f^\star(\vec{s}, x) \succeq^e_\star x$ by (Select), and if (Marked1) was used because $f \blacktriangleright g$ and $f^\star(\vec{s}, x) \succeq^e_\star t_1, \ldots, t_n, x$ for the same reasons.

7. If $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (In) and $t \succeq^e_\star q$ by (Select), then $t_i\langle u_1, \ldots, u_n\rangle \succeq^e_\star q$ for some $u_1, \ldots, u_n$ obtained from the appropriate $g^\star(\vec{t})$ by dropping arguments. There is some permutation π such that $s_{\pi(j)} \succeq^e_\star t_j$ for all j. By IH1 and the (Select) clause it suffices if $s_{\pi(i)}\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle \succeq^e_\star t_i\langle u_1, \ldots, u_n\rangle$. We use Lemma 5.29: $s_{\pi(i)} \succeq^e_\star t_i$ by assumption, and $f^\star(\vec{s}) \succeq^e_\star u_i$ by (Drop) and (In).

8. If $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (Marked1) or (Marked2) and $t \succeq^e_\star q$ by (Select), then $t_i\langle u_1, \ldots, u_n\rangle \succeq^e_\star q$ for some $u_1, \ldots, u_n$ obtained from the appropriate $g^\star(\vec{t})$ by dropping arguments. Using (Args) and IH1, it suffices if $f^\star(\vec{s}, f^\star(\vec{s})) \succeq^e_\star t_i\langle u_1, \ldots, u_n\rangle$, where the new argument $f^\star(\vec{s})$ is equipped with the same type as t_i. By (Select), this holds if $f^\star(\vec{s})\langle f^\star(\vec{s}), \ldots, f^\star(\vec{s})\rangle \succeq^e_\star t_i\langle\vec{u}\rangle$, which by Lemma 5.29 is the case because $f^\star(\vec{s}) \succeq^e_\star t_i$ by the definition of (Marked1) and (Marked2), and each $f^\star(\vec{s}) \succeq^e_\star u_i$ by Lemmas 5.24(3) and 5.30 (as $f^\star(\vec{s}) \succeq^e_\star g^\star(\vec{t})$).

9. If $t \succeq^e_\star q$ by (Args), (Drop) or (Swap) this is because some $u \succeq^e_\star q$. By IH1 it suffices to prove that $s \succeq^e_\star u$.

   If $s = f^\star(\vec{s}) \succeq^e_\star t = f^\star(\vec{t})$ by (In), then in the cases (Drop) and (Swap) it is evident that also $s \succeq^e_\star u$ by the same (Drop) or (Swap) clause, combined with (In). If $t \succeq^e_\star q$ by (Args), we also need to observe that $f^\star(\vec{s}) \succeq^e_\star$ "$g^\star(\vec{t})$ with some arguments dropped" by Lemmas 5.24(3) and 5.30.

If $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (Marked1) or (Marked2), then the order and number of arguments beyond the arity of g are entirely irrelevant; they only share the property that $f^\star(\vec{s}) \succeq^e_\star t_i$ for all i. Therefore also $s \succeq^e_\star u$ by (Marked1) or (Marked2): (Drop) only removes an extra argument, (Swap) changes the order of arguments, and (Args) adds arguments obtained from $g^\star(\vec{t})$ by dropping arguments. $f^\star(\vec{s}) \succeq^e_\star$ these extra arguments by Lemmas 5.24(3) and 5.30.

10. If $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (In), (Marked1) or (Marked2), and $t \succeq^e_\star q = h^{(\star)}(\vec{q})$ by (Copy) or (Marked1), then $f \blacktriangleright h$ either by compatibility of $\approx$ and $\blacktriangleright$, or by transitivity of $\blacktriangleright$. Moreover, $t \succeq^e_\star q$ holds because $g^\star(\vec{t}) \succeq^e_\star$ each q_i, so by IH1 and Lemma 5.24(3) also $f^\star(\vec{s}) \succeq^e_\star q_i$. Thus, $s \succeq^e_\star q$ by (Copy) or (Marked1).

    The same reasoning holds if $s \succeq^e_\star t$ by (Marked1) and $t \succeq^e_\star q$ by (Stat) or (Marked2): here, too, $g^\star(\vec{t}) \succeq^e_\star q_i$ is used for all i, and $f \blacktriangleright h$.

11. If $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (Marked1) and $t \succeq^e_\star g^\star(\vec{q}) = q$ by (In), then the latter holds because for some permutation π, $t_{\pi(i)} \succeq^e_\star q_i$ for all i. The former holds because $f^\star(\vec{s}) \succeq^e_\star t_j$ for all j. As $f^\star(\vec{s}) \succeq^e_\star t_{\pi(i)} \succeq^e_\star q_i$ for all i, which implies $f^\star(\vec{s}) \succeq^e_\star q_i$ by IH1, and $f \blacktriangleright g \approx h$ implies $f \blacktriangleright h$, we have $s \succeq^e_\star q$ by (Marked1).

12. Suppose $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (Marked2), and $t \succeq^e_\star q = h^{(\star)}(\vec{q})$ by either (Stat), (In) or (Marked2). Since $f \approx g \approx h$ also $f \approx h$. We will prove $s \succeq^e_\star q$ by (Stat) if q is unmarked, or by (Marked2) if q is marked.

    To this end, we first note that by IH1 $f^\star(\vec{s}) \succeq^e_\star q_i$ for all i: this is either because $f^\star(\vec{s}) \succeq^e_\star g^\star(\vec{t}) \succeq^e_\star q_i$ by Lemma 5.24(3), or because $f^\star(\vec{s}) \succeq^e_\star t_j \succeq^e_\star q_i$ for some j (where $f^\star(\vec{t}) \succeq^e_\star q_i$ is used in the derivation of $t \succeq^e_\star q$ if the clause (Stat) or (Marked2) was used, and otherwise $t_j \succeq^e_\star q_i$ is used). What remains to be seen is that $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star gstat(f)} (q_1, \ldots, q_{ar(h)})$. Consider $stat(f)$.

    - If $stat(f) = Lex$, then there is some $i \leq ar(f)$ such that $s_1 \succeq^e_\star t_1, \ldots, s_{i-1} \succeq^e_\star t_{i-1}$ and either $ar(g) = i - 1$, or $s^\star_i \succeq^e_\star t_i$.

      If $t \succeq^e_\star q$ by (In), then $ar(g) = ar(h)$ and each $t_j \succeq^e_\star q_j$, so also $s_1 \succeq^e_\star q_1, \ldots, s_{i-1} \succeq^e_\star q_{i-1}$ and either $ar(h) < i$ or $s^\star_i \succeq^e_\star q_i$, which gives the required lexicographic inequality.

      If $t \succeq^e_\star q$ by (Stat) or (Marked2), then there is some $k \leq ar(g)$ such that $t_1 \succeq^e_\star q_1, \ldots, t_{k-1} \succeq^e_\star q_{k-1}$ and either $ar(h) = k - 1$ or $t^\star_k \succeq^e_\star q_k$. Let $N := \min(i, k)$; then $N \leq ar(f)$ and $N \leq ar(g)$. For $j < N$ we have $s_j \succeq^e_\star t_j \succeq^e_\star q_j$, so by IH1 also $s_j \succeq^e_\star q_j$. If $N > ar(h)$ we are therefore done. Otherwise, $s^\star_N \succeq^e_\star q_N$: if $i < k$ because $s^\star_i \succeq^e_\star t_i \succeq_\star q_i$, and if $i \geq k$ because $s^\star_k \succeq_\star t^\star_k \succeq_\star s_k$ by Lemma 5.28.

- If $stat(f) = Mul$, then $\{1, \ldots, ar(f)\} = A \cup B$, and there is some function π mapping $\{1, \ldots, ar(g)\}$ to $A \cup B$ which touches every element of A exactly once, and if $\pi(i) \in A$ then $s_{\pi(i)} \succeq^e_\star t_i$ and if $\pi(i) \in B$ then $s^\star_{\pi(i)} \succeq^e_\star t_i$. B is non-empty.
  Moreover, we can write $\{1, \ldots, ar(g)\} = C \cup D$, and there is a function ρ mapping $\{1, \ldots, ar(h)\}$ to $C \cup D$, which touches every element of C exactly once, such that if $\rho(j) \in C$ then $t_{\rho(j)} \succeq^e_\star t_j$ and if $\rho(j) \in D$ then $t^\star_{\rho(j)} \succeq^e_\star t_j$. Since the second clause may be (In), it could be that D is empty.
  Let $A' = \{\pi(i) \mid i \in C\} \cap A$ and $B' = \{1, \ldots, ar(f)\} \setminus A'$. Then $B' \supseteq B$, so is non-empty. Then the function $\pi \circ \rho$ touches every element of A' exactly once: if $i \in A'$ then $i \in A$, so there is exactly one j such that $\pi(j) = i$, and by definition of A' this $j \in C$, so there is exactly one k such that $\rho(k) = j$. Hence, there is exactly one k where $\pi(\rho(k)) = i$. If $\pi(\rho(i)) \in A'$, then $s_{\pi(\rho(i))} \succeq^e_\star t_{\rho(i)} \succeq^e_\star q_i$ (because $\pi(\rho(i)) \in A$ and $\rho(i) \in C$), so by IH1 $s_{\pi(\rho(i))} \succeq^e_\star q_i$. If $\pi(\rho(i)) \in B'$, then either $s^\star_{\pi(\rho(i))} \succeq^e_\star t_{\rho(i)} \succeq^e_\star q_i$ (if $\rho(i) \in C$ but $\pi(\rho(i)) \in B$), or $s^{(\star)}_{\pi(\rho(i))} \succeq^e_\star t_{\rho(i)}$ and $t^\star_{\rho(i)} \succeq^e_\star q_i$ (if $\rho(i) \notin C$). In the first case we immediately have $s^\star_{\pi(\rho(i))} \succeq^e_\star q_i$ by IH1, and in the second case Lemma 5.28 and IH1 provide $s^\star_{\pi(\rho(i))} \succeq^e_\star q_i$. Either way, the multiset requirement is satisfied.

13. If $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t}) \succeq^e_\star q = h^\star(\vec{q})$ both by (In), then $f \approx g \approx h$ so $f \approx h$, and all arities and argument numbers are equal. If $stat(f) = Lex$ we simply have that $s_i \succeq^e_\star t_i \succeq^e_\star q_i$ for all i, and therefore $s_i \succeq^e_\star q_i$ by IH1 and thus $s \succeq^e_\star q$ by (In). If $stat(f) = Mul$, there are permutations π, ρ of $\{1, \ldots, n\}$ (where $n := ar(f)$) such that each $s_{\pi(i)} \succeq^e_\star t_i$ and $t_{\rho(i)} \succeq^e_\star q_i$, and $s_i \succeq^e_\star t_i \succeq^e_\star q_i$ whenever $i > n$. By the induction hypothesis $s_{\pi(\rho(i))} \succeq^e_\star q_i$ for $i \leq n$ and $s_i \succeq^e_\star q_i$ for $i > n$. Thus also $s \succeq^e_\star q$ by (In).

14. Suppose $s = f^\star(\vec{s}) \succeq^e_\star t = g^\star(\vec{t})$ by (In) and $t \succeq^e_\star q = h^{(\star)}(\vec{q})$ by (Stat) or (Marked2). Certainly $f \approx h$, so we will also derive $s \succeq^e_\star t$ by (Stat) or (Marked2), depending on whether q is marked or not. We know that $g^\star(\vec{t}) \succeq^e_\star q_i$ for all i, so by IH1 and Lemma 5.24(3), we have $f^\star(\vec{s}) \succeq^e_\star q_i$ as well. What remains to be seen is that is that $(s_1, \ldots, s_{ar(f)}) \succ^e_{\star gstat(f)} (q_1, \ldots, q_{ar(h)})$. Consider $stat(f)$.

    - If $stat(f) = Lex$, then $s_i \succeq^e_\star t_i$ for all i and there is some k such that $t_1 \succeq^e_\star q_1, \ldots, t_{k-1} \succeq^e_\star q_{k-1}$ and either $ar(h) < k \leq ar(g) = ar(f)$, or $t^\star_k \succeq^e_\star q_k$. By the induction hypothesis also $s_i \succeq^e_\star q_i$ for $i < k$, which suffices if $ar(h) < k$. Otherwise, by Lemma 5.28, $s^\star_k \succeq^e_\star t^\star_k \succeq^e_\star q_k$, which by IH1 implies that $s^\star_k \succeq^e_\star q_k$. This gives the required lexicographic inequality.
    - If $stat(f) = Mul$, then there is a permutation π such that $s_{\pi(i)} \succeq^e_\star t_i$ for all $i \in \{1, \ldots, ar(g)\} = A \cup B$, and a function ρ which maps

$\{1,\ldots,ar(h)\}$ to $A \cup B$ and touches all elements of A exactly once such that $t_{\rho(j)} \succeq^e_\star t_j$ if $\rho(j) \in A$ and otherwise $t^\star_{\rho(j)} \succeq^e_\star t_j$. B is non-empty. Let $A' = \pi(A)$ and $B' = \pi(B)$. Then the function $\pi \circ \rho$ touches the elements of A' exactly once (since π is a permutation), B' is non-empty, if $\pi(\rho(i)) \in A'$ then $s_{\pi(\rho(i))} \succeq^e_\star t_{\rho(i)} \succeq^e_\star q_i$ implies $s_{\pi(\rho(i))} \succeq^e_\star q_i$ by IH1, and if $\pi(\rho(i)) \in B'$ then $s^\star_{\pi(\rho(i))} \succeq^e_\star t^\star_{\rho(i)} \succeq^e_\star q_i$ by Lemma 5.28, which implies $s^\star_{\pi(\rho(i))} \succeq^e_\star q_i$ by IH1.

□

Having at last proved transitivity of $\succeq^e_\star$, we are almost ready to dispense of this extended relation. We merely need to convert the results back to $\succeq_\star$.

Theorem 5.32. *Claim 5.20 is true: $\succeq_\star$ defines exactly the same relation as $\Rightarrow^*_\star$ on meta-terms over $\mathcal{F}$, and $\succ_\star$ can be expressed as $\Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow^*_\star$.*

Proof. If $s \succeq_\star t$, then by Lemma 5.23 $s \Rightarrow^*_\star t$. If $s \succ_\star t$, then $s \Rightarrow_{\mathtt{put},top} \cdot \Rightarrow^*_\star t$ by the same lemma.

If $s \Rightarrow^*_{\mathcal{R}} t$, then by Lemma 5.26 $s\ (\succeq^e_\star)^*\ t$. By Theorem 5.31 (transitivity), this implies that $s \succeq^e_\star t$, and by Lemma 5.25 ($\succeq^e_\star$ and $\succeq_\star$ coincide on star-free meta-terms) we conclude that $s \succeq_\star t$. If $s \Rightarrow_{\lambda\mathtt{put}} \cdot \Rightarrow^*_\star t$, then $s^\star \Rightarrow^*_\star t$, so $s^\star \succeq^e_\star t$ by Lemma 5.26 and transitivity. As $s \succ^e_\star t$, and $\succ^e_\star$ coincides with $\succ_\star$ on star-free meta-terms, we conclude that $s \succ_\star t$. □

Corollary 5.33. *$(\succeq_\star, \succ_\star)$ is an application-free strong reduction pair.*

It is worth observing that transitivity of the ordering is not so common as may be expected. Both HORPO and CPO are non-transitive; the corresponding reduction pair is $(\succ^*_{\mathrm{HORPO}}, \succ^+_{\mathrm{HORPO}})$.

Example 5.34. Suppose we need a strong reduction pair to orient a number of requirements, one of which is: $\mathtt{f}(\lambda x.\mathtt{g}(x), X) \succ \mathtt{g}(X)$. Here, $\mathtt{f} : [\mathtt{o} \to (\mathtt{o} \times \mathtt{o})] \longrightarrow \mathtt{o}$ and $\mathtt{g} : [\mathtt{o}] \longrightarrow \mathtt{o}$. Suppose also that for the other constraints we must take $\mathtt{f} \blacktriangleright \mathtt{g}$.

We do not succeed with $\succsim_{\mathrm{CPO}}$, even though we *can* orient the constraint with its transitive closure: $\mathtt{f}(\lambda x.\mathtt{g}(x), X) \succ_{\mathrm{CPO}} (\lambda x.\mathtt{g}(x)) \cdot X \succ_{\mathrm{CPO}} \mathtt{g}(X)$.

With $\succ_\star$, we can orient the rule in one go: $\mathtt{f}^\star(\lambda x.\mathtt{g}(x), X) \succ_\star \mathtt{g}(X)$ by (Select), because $(\lambda x.\mathtt{g}(x))\langle \mathtt{f}^\star(\lambda x.\mathtt{g}(x), X)\rangle = \mathtt{g}(\mathtt{f}^\star(\lambda x.\mathtt{g}(x), X)) \succeq_\star \mathtt{g}(X)$. This holds by (Fun): $\mathtt{f}^\star(\lambda x.\mathtt{g}(x), X)) \succeq_\star X$ by (Select).

Example 5.35. Suppose we must orient $\mathtt{f}(\lambda x.\mathtt{g}(x,x), X, Y) \succ \mathtt{g}(X,Y)$, and other clauses have caused a constraint that $\mathtt{f} \blacktriangleright \mathtt{g}$. We easily succeed with $\succ_\star$ as before, but to orient this with $\succ^+_{\mathrm{CPO}}$, we must construct some term t such that $\mathtt{f}(\lambda x.\mathtt{g}(x,x), X, Y) \succ_{\mathrm{CPO}} t$ and both $t \succ_{\mathrm{CPO}} X$ and $t \succ_{\mathrm{CPO}} Y$. Whether this is possible depends on types, and whether or not there are function symbols $\blacktriangleright$-smaller than $\mathtt{f}$.

CPO and StarHorpo will be compared in a bit more detail in Section 5.7.

5.5 A Reduction Pair for AFSMs

So far, we have derived a strong reduction pair *for application-free terms*. But although we have seen, in Chapter 3.2, that we can always transform an AFSM into an IDTS (where everything is application-free), it would still be nice to have a result immediately on our formalism of choice. Fortunately, this is very easy.

Definition 5.36 (StarHorpo reduction pair)**.** Given an AFSM $(\mathcal{F}, \mathcal{R})$, let $\mathcal{F}_@ := \mathcal{F} \cup \{@^{\sigma\to\tau} : [\sigma\to\tau\times\sigma]\longrightarrow\tau \mid \sigma,\tau \text{ types}\}$. Let μ be the function which replaces all appearances of an application $s\cdot t$ with $s:\sigma\to\tau$ in a meta-term by $@^{\sigma,\tau}(s,t)$.

Fixing a well-founded precedence $\blacktriangleright$ and a status function on $\mathcal{F}_@$, let $(\succsim,\succ)$ be given by: $s\succsim t$ if $\mu(s)\succeq_\star\mu(t)$ and $s\succ t$ if $\mu(s)\succ_\star\mu(t)$.

Theorem 5.37. *The pair $(\succsim,\succ)$ from Definition 5.36 is a strong reduction pair.*

Proof Sketch. It is easy enough to see that always $\mu(s\gamma)=\mu(s)\gamma^\mu$, where $\gamma^\mu(x)=\mu(\gamma(x))$. Having this, most important properties are inherited from $(\succeq_\star,\succ_\star)$. The last one, $\succsim$ includes β, holds because $\mu((\lambda x.s)\cdot t)^\star = @^\star_{\sigma,\tau}(\lambda x.\mu(s),\mu(t)) \Rightarrow_{\texttt{select}} \mu(s)[x:=@^\star_{\sigma,\tau}(\lambda x.\mu(s),\mu(t))] \Rightarrow^*_{\texttt{select}} \mu(s)[x:=\mu(t)] = \mu(s[x:=t])$. □

Since $\succeq_\star$ and $\Rightarrow^*_\star$ can be used interchangeably, Theorem 5.37 is both a result about the iterative and the recursive definition of HOIPO. The proof is merely a sketch because the result will actually follow as a consequence of Theorem 5.39.

5.6 Function Symbol Transformations

A notable omission from both the version of RPO and StarHorpo defined here, is that the lexicographical extension is only used in one orientation. We can easily orient a rule $\texttt{Plus}(\texttt{s}(X),Y)\Rightarrow\texttt{Plus}(X,\texttt{s}(Y))$, but we cannot deal with the very similar rule $\texttt{Plus}(X,\texttt{s}(Y))\Rightarrow\texttt{Plus}(\texttt{s}(X),Y)$. The definition of RPO in [27] does not have this problem: here, symbols with status *Lex* are equipped with a permutation for the arguments. The second `Plus` example can be dealt with by assigning `Plus` the permutation $\pi=[2,1]$.

In this chapter I decided against immediately including such a permutation. The proofs are complex enough as they are, and it is unnecessary to immediately include this feature in the theory, just as it was unnecessary to consider application in the definition of HOIPO: we can simply add the permutation on arguments of lexicographic symbols afterwards. That is, in the function μ from Definition 5.36, we could also replace functional terms $f(s_1,\ldots,s_n)$ by $f(s_{\rho(1)},\ldots,s_{\rho(n)})$ for some fixed permutation ρ for the symbol f.

But we can do more! As formalised for first-order rewriting in IsaFoR [121], and also implemented in for instance AProVE [45], we might "filter" away a function symbol f with only one argument, replacing a meta-term $f(s)$ by just s. More generally, we can manipulate function symbols in any way we like, provided their arguments are preserved if we want to obtain a strong reduction pair:

Definition 5.38 (Argument (Preserving) Function)**.** Let Σ be a set of function symbols, containing a symbol $\bot_\sigma$ for all types σ.

An *argument function* from $\mathcal{F}_@$ to Σ is a function π which maps every function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}_@$ to a term $\lambda x_1 \ldots x_n.s : \sigma_1 \to \ldots \to \sigma_n \to \tau$ over Σ such that $FV(s) \subseteq \{x_1, \ldots, x_n\}$. An *argument preserving function* additionally has the property that $FV(s) = \{x_1, \ldots, x_n\}$. An argument (preserving) function is extended to application-free meta-terms as follows:

$$\begin{array}{rcll} \overline{\pi}(x) & = & x & \text{for } x \in \mathcal{V} \\ \overline{\pi}(\lambda x.s) & = & \lambda x.\overline{\pi}(s) & \\ \overline{\pi}(Z(s_1, \ldots, s_n)) & = & Z(\overline{\pi}(s_1), \ldots, \overline{\pi}(s_n)) & \text{for } Z \in \mathcal{M} \\ \overline{\pi}(f(s_1, \ldots, s_n)) & = & t[x_1 := \overline{\pi}(s_1), \ldots, x_n := \overline{\pi}(s_n)] & \text{if } \pi(f) = \lambda x_1 \ldots x_n.t \end{array}$$

Note that in the last case, the substitution used is actually a meta-substitution, since some of the $\overline{\pi}(s_i)$ may not be terms. Typically, argument preserving functions are used in the two ways suggested above, and for minimal symbols:

- changing the order of arguments in a term with status Lex, so that the more important arguments are put in front (that is, $f(s_1, \ldots, s_n)$ becomes $f'(s_{\rho(1)}, \ldots, s_{\rho(n)})$ for some new symbol f' and permutation ρ);
- filtering away a symbol f with only one argument (just like, when using polynomial interpretations, f could be interpreted by the function $\boldsymbol{\lambda} n.n$);
- mapping a 0-ary symbol to $\bot_\sigma$, so it becomes minimal for $\succeq_\star$.

Argument functions are more general, and also permit for instance replacing $f(s_1, \ldots, s_n)$ by $f'(s_{i_1}, \ldots, s_{i_k})$, so filtering some arguments away.

The definition does not pose restrictions on what form an argument function should have, as we have nothing to lose, and perhaps something to gain, by posing the result in a general way.

Theorem 5.39. *Consider a set of function symbols $\mathcal{F}$, and an argument function π on $\mathcal{F}_@$, which maps to application-free meta-terms over some set of symbols Σ, and which maps each $@^{\sigma,\tau}$ either to itself or to $\lambda xy.@^{\sigma,\tau}{}_{swap}(y, x)$. Let $\blacktriangleright$ be a well-founded precedence on the symbols of Σ, and let $stat$ be a status function on this same set.*

Let the pair $(\succsim, \succ)$ be given by:

- *$s \succsim t$ if $\overline{\pi}(\mu(s)) \succeq_\star \overline{\pi}(\mu(t))$;*
- *$s \succ t$ if $\overline{\pi}(\mu(s)) \succ_\star \overline{\pi}(\mu(t))$.*

Then $(\succsim, \succ)$ is a weak reduction pair, and if π is an argument preserving function it is even a strong reduction pair.

Proof. We must see that $\succ$ is a well-founded ordering relation (so transitive and well-founded), that $\succsim$ is a quasi-ordering (so transitive and reflexive) compatible with $\succ$, that both relations are (meta-)stable, that $\succsim$ is monotonic (and so is $\succ$ if π is an argument preserving function), and that $\succsim$ contains beta. The last requirement was already demonstrated in the proof sketch of Theorem 5.37, and $\overline{\pi}$ does not cause problems by the restriction that π can at most permute the arguments of $@^{\sigma,\tau}$. So let us consider the other constraints.

If $s \succ t \succ q$, then $\overline{\pi}(\mu(s)) \succ_\star \overline{\pi}(\mu(t)) \succ_\star \overline{\pi}(\mu(q))$, so by transitivity of $\succ_\star$ we have $\overline{\pi}(\mu(s)) \succ_\star \overline{\pi}(\mu(q))$ as required. In the same way, transitivity of $\succsim$, well-foundedness of $\succ$ and compatibility of the pair are inherited from the corresponding properties of $(\succeq_\star, \succ_\star)$.

In order to prove stability, let $\gamma^{\pi,\mu}$ denote the substitution which maps x to $\overline{\pi}(\mu(\gamma(x)))$ for all (meta-)variables $x \in dom(\gamma)$ for some given substitution γ. We can see that $\overline{\pi}(\mu(s\gamma)) = \overline{\pi}(\mu(s))\gamma^{\pi,\mu}$ for all s and γ whose domain contains all meta-variables in s, by induction first on the number of meta-variables in $dom(\gamma)$, second on the form of s. Write $\overline{\pi}\mu(s)$ as short-hand notation for $\overline{\pi}(\mu(s))$.

- if s is a variable not in $dom(\gamma)$, both sides are s;
- if s is a variable in $dom(\gamma)$, both sides are $\overline{\pi}\mu(\gamma(s))$;
- if s is a meta-variable application $Z(s_1, \ldots, s_n)$ and $\gamma(Z) = \lambda x_1 \ldots x_n.t$, then we have $\overline{\pi}\mu(s\gamma) = \overline{\pi}\mu(t[x_1 := s_1\gamma, \ldots, x_n := s_n\gamma])$ and $\overline{\pi}\mu(s)\gamma^{\pi,\mu} = \overline{\pi}\mu(t)[x_1 := \overline{\pi}\mu(s_1)\gamma^{\pi,\mu}, \ldots, x_n := \overline{\pi}\mu(s_n)\gamma^{\pi,\mu}]$, which by the second induction hypothesis equals $\overline{\pi}\mu(t)[x_1 := \overline{\pi}\mu(s_1\gamma), \ldots, x_n := \overline{\pi}\mu(s_n\gamma)]$. By the first induction hypothesis, both sides are equal;
- if $s = \lambda x.t$ then $\overline{\pi}\mu(s\gamma) = \lambda x.\overline{\pi}\mu(t\gamma)$, which by the induction hypothesis equals $\lambda x.(\overline{\pi}\mu(t)\gamma^{\pi,\mu}) = \overline{\pi}\mu(s)\gamma^{\pi,\mu}$;
- if $s = t_1 \cdot t_2$ with $t_1 : \sigma \to \tau$, then $\overline{\pi}\mu(s\gamma) = @^{\sigma,\tau(')}(\overline{\pi}\mu(t_{\rho(1)}\gamma), \overline{\pi}\mu(t_{\rho(2)}\gamma))$ for some permutation ρ of $\{1, 2\}$, and by the second induction hypothesis this equals $@^{\sigma,\tau(')}(\overline{\pi}\mu(t_{\rho(1)})\gamma^{\pi,\mu}, \overline{\pi}\mu(t_{\rho(2)})\gamma^{\pi,\mu}) = \overline{\pi}\mu(s)\gamma^{\pi,\mu}$, since substitution does not affect the type of t_1;
- finally, if $s = f(s_1, \ldots, s_n)$, and $\overline{\pi}(f) = \lambda x_1 \ldots x_n.t$, then $\overline{\pi}\mu(s\gamma) = \overline{\pi}(f(\mu(s_1\gamma), \ldots, s_n\gamma)) = t[x_1 := \overline{\pi}\mu(s_1\gamma), \ldots, x_n := \overline{\pi}\mu(s_n\gamma)]$, which by the (second) induction hypothesis equals $t[x_1 := \overline{\pi}\mu(s_1)\gamma^{\pi,\mu}, \ldots, x_n := \overline{\pi}\mu(s_n)\gamma^{\pi,\mu}]$. On the other hand, $\overline{\pi}\mu(s)\gamma^{\pi,\mu} = t[x_1 := \overline{\pi}\mu(s_1), \ldots, x_n := \overline{\pi}\mu(s_n)]\gamma^{\pi,\mu}$. Since t itself contains only the variables $x_1, \ldots, x_n$, this is exactly $t[x_1 := \overline{\pi}\mu(s_1)\gamma^{\pi,\mu}, \ldots, x_n := \overline{\pi}\mu(s_n)\gamma^{\pi,\mu}]$, so both sides are equal.

Thus, if $\overline{\pi}\mu(s) \succ_\star \overline{\pi}\mu(t)$, and γ is a substitution, then also $\overline{\pi}\mu(s)$ is a pattern. Therefore (using stability of $\succ_\star$), $\overline{\pi}\mu(s\gamma) = \overline{\pi}\mu(s)\gamma^{\pi,\mu} \succ_\star \overline{\pi}\mu(t)\gamma^{\pi,\mu} = \overline{\pi}\mu(t\gamma)$ as required; stability of $\succeq_\star$ is similar.

Finally, monotonicity. We must see that if $s \succsim t$ then also $C[s] \succsim C[t]$ for all contexts C, and if π is an argument preserving function, then the same holds for

$\succ$. We can do so by induction on the form of C. Each of the cases where $C[] = \lambda x.D[]$ or $C[] = D[] \cdot q$ or $C = q \cdot D[]$ is completely straightforward (in the last two cases, note that $\overline{\pi}$ can at most exchange the two arguments of $@^{\sigma,\tau}$, so nothing exciting happens). What remains is the case where $C[] = f(q_1, \ldots, D[], \ldots, q_n)$ and by the induction hypothesis either $D[s] \succsim D[t]$ or $D[s] \succ D[t]$. Let $\overline{\pi}(f) = \lambda x_1 \ldots x_n.u$. Then $\overline{\pi}\mu(C[s]) = u[x_1 := \overline{\pi}\mu(q_1), \ldots, x_i := \overline{\pi}\mu(D[s]), \ldots, x_n := \overline{\pi}\mu(q_n)]$.

In the $\succ$ case, where π can be assumed to be an argument preserving function, we know that x_i occurs in u (possibly more than once). Since $\succ_\star$ is both monotonic and transitive (and therefore $E[v, \ldots, v] \succ_\star E[w, \ldots, w]$ if $v \succ_\star w$ and $m > 0$), this term $\succ_\star u[x_1 := \overline{\pi}\mu(q_1), \ldots, x_i := \overline{\pi}\mu(D[t]), \ldots, x_n := \overline{\pi}\mu(x_n)] = \overline{\pi}\mu(C[t])$. In the $\succeq_\star$ case it may be that x_i does not occur in u, but by reflexivity of $\succeq_\star$ we still have the required result. □

Example 5.40. Consider the AFSM `foldl`:

$$\begin{array}{rcl} \mathtt{nil} & : & \mathtt{list} \\ \mathtt{cons} & : & [\mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list} \\ \mathtt{foldl} & : & [(\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{nat} \end{array}$$

$$\begin{array}{rcl} \mathtt{foldl}(\lambda xy.F(x,y), X, \mathtt{nil}) & \Rightarrow & X \\ \mathtt{foldl}(\lambda xy.F(x,y), X, \mathtt{cons}(Y,Z)) & \Rightarrow & \mathtt{foldl}(\lambda xy.F(x,y), F(X,Y), Z) \end{array}$$

Let Σ be $\mathcal{F}_@ \cup \{\bot_\sigma \mid \sigma \in \mathcal{T}\}$, but with all base types collapsed into a single type `o`. Let $\pi(f) = \lambda\vec{x}.f(\vec{x})$ for all symbols except `foldl`, and let $\pi(\mathtt{foldl}) = \lambda xyz.\mathtt{foldl}(x,z,y)$, so we simply swap the last two arguments of `foldl`. To prove the system terminating in one go, we have the following constraints:

$$\begin{array}{rcl} \mathtt{foldl}(\lambda xy.F(x,y), \mathtt{nil}, X) & \succ_\star & X \\ \mathtt{foldl}(\lambda xy.F(x,y), \mathtt{cons}(Y,Z), X) & \succ_\star & \mathtt{foldl}(\lambda xy.F(x,y), Z, F(X,Y)) \end{array}$$

Whether we use the iterative approach or a recursive analysis, both requirements are easily satisfied if we choose $stat(\mathtt{foldl}) = Lex$.

Example 5.41. Consider the first-order system:

$$\begin{array}{rcl} \mathtt{f}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{g}(x,y) \\ \mathtt{g}(x, \mathtt{s}(y)) & \Rightarrow & \mathtt{f}(x,y) \\ \mathtt{f}(x,y) & \Rightarrow & \mathtt{g}(\mathtt{a}, \mathtt{b}) \end{array}$$

Using an argument preserving function with $\pi(\mathtt{a}) = \pi(\mathtt{b}) = \bot$ and a precedence with $\mathtt{f} \approx \mathtt{g}$ we obtain:

$$\begin{array}{rcl} \mathtt{f}(\mathtt{s}(x), y) & \succ_\star & \mathtt{g}(x,y) \\ \mathtt{g}(x, \mathtt{s}(y)) & \succ_\star & \mathtt{f}(x,y) \\ \mathtt{f}(x,y) & \succeq_\star & \mathtt{g}(\bot,\bot) = \overline{\pi}(\mathtt{g}(\mathtt{a},\mathtt{b})) \end{array}$$

Thus, with rule removal we can get rid of the first two rules. The last one is easy to orient separately. This demonstrates how minimal symbols can used to orient constraints that could otherwise not be handle with (HO)RPO.

Note that, although this is a first-order system, the recursive path ordering from Section 5.1.1 cannot handle it! However, although technically not part of the specification of RPO, minimal symbols are not entirely new. A very recent paper [120] introduces a rule that $x \succsim f$ for x a variable and f a function symbol of arity 0 which is minimal in the precedence. This feature has already been formalised in IsaFoR [121], and also implemented in AProVE [45].

5.7 CPO Versus StarHorpo

CPO and StarHorpo are truly incomparable. Each has advantages over the other.

CPO is defined only for AFSs (although a definition of HORPO has also been extended to HRSs [106]), and moreover assumes that all function symbols have a base type as output type. StarHorpo is defined for the more general class of AFSMs, which includes AFSs, as we saw in Chapter 3.4.

CPO uses a type ordering and accessibility relation, which may add significant power to the system. To some extent, a type changing function which does not just collapse all base types can be used to handle systems for which CPO uses accessibility, but for instance Example 5.8 cannot be handled with StarHorpo.

StarHorpo uses minimal symbols and a separate quasi-ordering $\succsim$, and encodes application as a function symbol. The latter cannot just be done in CPO, as the symbols $@^\sigma$ do not in general have a base type as output type. Consequently, CPO uses separate (weaker!) cases if the left-hand side is a function symbol. StarHorpo also uses argument (preserving) functions, but these could also be used with CPO.

A very important difference between CPO and StarHorpo concerns the (Select) rule. This rule is far more powerful in StarHorpo than in its counterpart in CPO. This is related to the difference in the formalisms for which the systems are defined: HORPO and CPO target AFSs, where it is actually very uncommon to have abstractions in the left-hand side of rules. In our AFSMs, where left-hand sides often contain meta-terms $\lambda\vec{x}.C[F(\vec{x})]$, it is however very likely that we will need the transitivity which is made possible by the new (Select).

Transitivity is another point where StarHorpo has the advantage. Of course $\succ^+_{\mathrm{CPO}}$ is transitive, but this relation may not be decidable, as we saw in Examples 5.34 and 5.35. In particular Example 5.35 demonstrates how the use of marked terms adds power in practice. By postponing the choice to select a term smaller than a given term $f(\vec{s})$ (and just using $f^\star(\vec{s})$) we can make multiple different choices later.

5.8 Overview

In this chapter we have seen an extension of the iterative path ordering to the higher-order case. Specifically, to the IDTS formalism, which is similar to the AFSM formalism we normally use, but considers *application-free* (meta-)terms. This iterative definition, HOIPO, provides a simple and elegant termination method for IDTS. We have also considered a recursive definition of HOIPO, and discussed how the resulting relations can be used as a strong reduction pair on AFSMs (without the application-free restriction). In addition, we have extended the method with the notion of *argument preserving functions*, which will come back in a generalised form in Chapter 6.6.3.

Compared to the computability path ordering (the most recent extension of the higher-order recursive path ordering), StarHorpo has far fewer rules. However, for optimal results, StarHorpo works best in combination with transformations, in particular the argument (preserving) functions of Section 5.6.

The power of the respective techniques is incomparable. While CPO beats StarHorpo in examples where a type ordering, and in particular an accessibility argument, are needed, the use of "minimal symbols" and the transformations from Section 5.6 are new in StarHorpo. We also gain additional power by treating application as a function symbol. Moreover, unlike both HORPO and CPO, the new StarHorpo is natively defined as a transitive relation, a great advantage since it is in general very hard to decide whether some term $s \succ^{+}_{\mathrm{CPO}} t$.

As we will see in Chapter 8.6, the techniques in this chapter can all easily be implemented in an automatic tool, which determines not only the precedence and the status automatically, but also a suitable argument function.

With an eye on future work for the techniques in this chapter, consider Figure 5.3. This diagram extends on Figure 5.3. As before, all the iterative and recursive path orderings discussed in this chapter are listed in their mutual context, with a horizontal line indicating an equivalence, and a vertical or diagonal line indicating inclusion. This time also [76] is included. In this paper, Femke van Raamsdonk and I present a more restrictive version of the higher-order iterative path ordering, which mostly corresponds to the original definition of HORPO [63] (the iterative definition of [76] is slightly stronger, but the difference is minor).

The dotted lines in the diagram suggest an obvious direction for future research: to combine the strengths of CPO and StarHorpo, perhaps along with an improvement of the iterative technique. At the moment, the main difference between the relations is the type ordering. StarHorpo has none because, in a definition as term rewriting system, types are preserved in every step. Thus, we cannot have $s \succeq_{\star} t$ if s and t do not have the same type, for that would also mean that $s \Rightarrow^{*}_{\star} t$. However, there are plenty of possible solutions to this issue which may be explored.

In an entirely different direction, we might consider alternative shapes for argument functions that could be attempted by default.

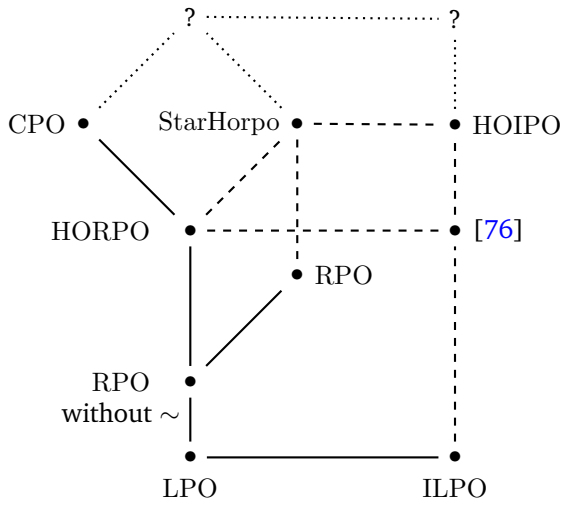


Figure 5.3: Iterative and Recursive Path Orderings – the solid lines represent existing results, the dashed lines represent theory from this chapter or [76], and the dotted lines suggest future work.

Dependency Pairs

6

Or, What shall we order?

Both the polynomial interpretations in Chapter 4 and the path ordering from Chapter 5 give *quasi-simplification orderings*. That is, $f(s_1, \ldots, s_n) \succsim s_i$ if both terms have the same base type. This property, combined with monotonicity of $\succsim$, means that quasi-simplification orderings cannot deal with a rule like:

$$\mathtt{quot}(\mathtt{s}(X), \mathtt{s}(Y)) \Rightarrow \mathtt{s}(\mathtt{quot}(\mathtt{minus}(X, Y), \mathtt{s}(Y)))$$

For if we could orient this rule with a quasi-simplification ordering, then $\mathtt{quot}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{s}(\mathtt{0}))) \succ \mathtt{s}(\mathtt{quot}(\mathtt{minus}(\mathtt{0}, \mathtt{s}(\mathtt{0})), \mathtt{s}(\mathtt{s}(\mathtt{0})))) \succsim \mathtt{s}(\mathtt{quot}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{s}(\mathtt{0}))))$ $\succsim \mathtt{quot}(\mathtt{s}(\mathtt{0}), \mathtt{s}(\mathtt{s}(\mathtt{0})))$, which contradicts well-foundedness.

The *dependency pair approach*, defined for first-order TRSs by Arts and Giesl in [9], provides a solution for systems like this. This approach transforms a term rewriting system into groups of ordering constraints, such that rewriting is terminating if and only if the groups of constraints are (separately) solvable. To solve these constraints, it suffices to use a weak reduction pair rather than a strong reduction pair. Moreover, the constraints can be simplified using for instance argument filterings and usable rules [53], which not only makes it possible to deal with rules like the `quot` rules above, but adds other advantages as well. Various optimisations of the method have been studied, see for example [46, 54].

It is not obvious how the dependency pair approach should be extended to higher-order rewriting. In particular the question of how we should deal with functional meta-variables, and with bound variables, are objects of decision. As a result, in the last few years not one but *three* dependency pair approaches for higher-order rewriting have been developed independently. The *static* approach, developed in [87, 114] by Kusakari et al., avoids dependency pairs headed by a functional meta-variable, but allows bound variables to become free. The *dynamic* approach, first defined in [112] by Sakai et al., and extended by Femke van Raamsdonk and me in [79, 80], does admit dependency pairs headed by a functional meta-variable, but avoids freeing bound variables. The *type-based* approach, by Roux [109], is based on the shape of types rather than terms.

It is the second, dynamic, approach which we will study in this chapter.

Chapter Setup. Because the method of using dependency pairs offers many possibilities, the material has been split up over two chapters. This chapter considers a basic definition of the dependency pairs approach. Furthermore, two improvements are considered: *formative rules*, a method similar to *usable rules* in the first-order approach, and a definition of *tagged dependency chains*. Both techniques are designed specifically for the higher-order setting. Chapter 7 considers a higher-order extension of the dependency pair *framework* [46]: there we will consider several ways to manipulate dependency pairs and chains. The techniques in Chapter 7 are mostly extensions of existing first-order techniques.

In this chapter, Section 6.1 briefly discusses the ideas from existing studies of dependency pairs for higher-order rewriting. Section 6.2 relates those parts of the first-order dependency pair approach most relevant to the work discussed here. The definitions are presented in a style that, although somewhat unusual, is most suitable for extension to the higher-order case. In Section 6.3 we will define a basic dependency pair approach for higher-order rewriting.

Section 6.4 discusses a first improvement: the notion of a *formative reduction*, which leads to *formative rules*, a variation of the *usable rules* which are commonly used in the first-order case. Formative rules have been designed specifically for the higher-order setting and currently have no first-order counterpart. Unlike usable rules, formative rules require a special form of dependency chains.

The initial definitions do not suffice to deal with systems where quasi-simplification orderings fail. Therefore Section 6.5 discusses an important improvement, limited to the class of *abstraction-simple* AFSMs. For such systems, we can weaken the *subterm property*, which makes life difficult in the basic definition, and go beyond quasi-simplification orderings. Finally, in Section 6.6 we will consider some systematic ways of finding a suitable reduction pair, both for the basic approach and for abstraction-simple systems.

The definitions and results from Section 6.3 onward are new in this thesis and the corresponding papers. The basic results of Section 6.3.1–6.3.3 have a counterpart in the setting of HRSs [112], but the definitions in this setting are significantly easier than in the AFSM setting. Meta-variable conditions, β-saturating, β-reduced sub-meta-terms and reduction triples do not appear in [112].

Both for polynomial interpretations and the iterative path ordering, the definition was designed to be *simple*. To gain extra power, transformations are used (such as type changing functions and argument preserving functions). With dependency pairs, this is not the case: the definitions in this chapter are as general as possible, not assuming any transformation was done on the system beforehand (other than presenting the system with maximum arity). This is because the dependency pair approach in the first-order setting provides a characterisation of termination, and is used both for termination and non-termination analysis. To have any hope for the same power in the higher-order case, the definitions must be general.

This chapter is primarily based on [79] and its journal extension [80]. However, where [79, 80] concern the AFS formalism, here of course AFSMs are considered.

6.1 Background and Related Work

As mentioned in the introduction, the extension of dependency pairs to the higher-order case is not entirely straightforward, and thus many variations exist. This work can roughly be split along two axes. On the one axis, the higher-order formalism: dependency pair definitions have been provided for applicative rewriting, rewriting modulo β (HRSs), and with β as a separate step (AFSs). On the other axis, the style of dependency pairs, with the main styles being *dynamic* and *static*. Figure 6.1 gives an overview.

	Applicative	**HRS**	**AFS**
Dynamic	[86]	[112] [78]	[79] [80]
Static	[89] [90]	[16] [111] [87] [114]	[16]
Other	[6] [7] [56] [47]	–	[109]

Figure 6.1: Papers on Higher-order Dependency Pairs

The dynamic and static approach differ in the treatment of leading variables in the right-hand sides of rules (subterms $x \cdot s_1 \cdots s_n$ with $n > 0$ and x a free variable; the corresponding notion in the AFSM formalism is meta-variable application). In the dynamic approach, such subterms lead to a dependency pair; in the static approach they do not. Consequently, first-order techniques like argument filterings, the subterm criterion and usable rules are easier to extend to a static approach, but this approach is not always applicable, and moreover incomplete.

Dependency pairs for applicative rewriting. Since many definitions of "higher-order" dependency pairs exist for applicative systems, they deserve some mention. Recall that in applicative systems there are no meta-variables or abstraction, but functional variables may be present. There are various styles of applicative rewriting; untyped, simply typed, and with alternative forms of typing.

A dynamic approach was defined both for untyped and simply-typed applicative systems in [86], along with a definition of argument filterings. A first static approach appears in [89] and is improved in [90]. The method is restricted to *plain function passing* systems where, intuitively, leading variables are harmless.

Due to the lack of binders, it is also possible to eliminate leading variables by instantiating them, as is done for simply-typed systems in [6, 7]. In [56], an uncurrying transformation from untyped applicative systems to normal first-order systems is used, which preserves and reflects termination. In [47] several first-order dependency pair processors are adapted to work better for untyped applicative systems; this paper also considers an uncurrying transformation.

However, as we have seen in Chapter 3.1, results on applicative systems cannot be transposed to rewriting with binders. Nor do these results have any parallels in the presence of λ-abstraction. Thus, let us move on to the results for HRSs and AFSs.

Dynamic Dependency Pairs for HRSs. A first definition of dependency pairs for HRSs is given in [112]. Here termination is not equivalent to the absence of infinite dependency pair chains, and a term is required to be (weakly) greater than its subterms (the *subterm property*), which makes many optimisations impossible. Consequently, most of the focus since has been on the static approach. However, in [78] (an extended abstract by Femke van Raamsdonk and me) it is discussed how the subterm property in this setting may be weakened by posing restrictions on the rules. This discussion lays the foundation for Section 6.5.

Static Dependency Pairs for HRSs. A first appearance of static dependency pairs for HRSs is in [111], where an analysis is given for non-nested and strongly linear systems. The limitations are very strong, however, and this method is subsumed by [89], where the static approach is moved to the setting of HRSs. This approach is extended with argument filterings and usable rules in [114]. The static approach omits collapsing dependency pairs (that is, dependency pairs $f^\sharp(\vec{l}) \Rrightarrow x \cdot \vec{r}$ with x a variable),[1] which avoids the need for a subterm property. The technique is restricted to *plain function passing* HRSs. A system with for instance the (terminating) rule $\mathsf{h}(\mathsf{g}(\lambda x.F(x))) \Rightarrow F(\mathsf{a})$ cannot be handled. In addition, bound variables may become free in a dependency pair. For example, the rule $\mathtt{I}(\mathtt{s}(n)) \Rightarrow \mathtt{twice}(\lambda x.I(x), n)$ generates a dependency pair $\mathtt{I}^\sharp(\mathtt{s}(n)) \Rrightarrow \mathtt{I}^\sharp(x)$ which admits an infinite static dependency pair chain, even though, as we will see, the rule itself is not problematic.

Type-based Dependency Pairs for AFSs. In *type-based termination analysis*, function symbols in terms are equipped with "sized types", a form of dependent typing. These types can themselves be seen as terms. Roux [109] considers a dependency pair approach based on the sized types rules. Although this method uses the ideas of first-order dependency pairs, it does not seem to be comparable with dependency pair approaches which consider the shape of terms rather than types. The restriction of type-based dependency pairs to the first-order setting also does not coincide with the first-order definition of dependency pairs.

Dynamic Dependency Pairs for AFSs. The definitions for HRSs [87, 112] do not immediately carry over to AFSs, since AFSs may have rules of functional type and β-reduction is a separate rewrite step. Certainly, we could transform AFSs into HRSs by η-normalising rules, and introducing a separate function symbol for application and an infinite number of β-rules (as done in Chapter 3.3), but this transformation may lose termination – and in fact, examples of systems which this transformation loses can often be proved terminating with the *dependency graph* from Chapter 7.4.

Except for a short paper by Blanqui [16] (which introduces static dependency pairs on a form of rewriting which includes AFSs, but poses some restrictions

[1] This definition of *collapsing* generalises the first-order notion ([10, Def. 9.2.3]), where a collapsing rule is a rule where the right-hand side is a variable.

such as function symbols having a base type as output type), dependency pairs for AFSs primarily appear in [79], by Femke van Raamsdonk and me. An extended version of this paper, [80], includes argument functions and a restriction of *locality*, which allows for the subterm property to be weakened.

This chapter considers dynamic dependency pairs for AFSMs. The method conservatively extends the one for first-order rewriting, and is both sound and complete. Because of the choice for a dynamic approach, the definitions are in principle not restricted to a sub-class of AFSMs. The restrictions which are needed to weaken the subterm property and for instance make it possible to use argument filterings, are optional.

To some extent, the static and dynamic approaches can be combined in the same framework. Some words about this will be said in Chapter 7.8.

6.2 First-Order Dependency Pairs

To place the ideas and definitions in this chapter in their context, let us start by considering the corresponding definitions for the first-order case. In this setting, without the complications brought on by λ-abstraction and β-reduction, the intuition behind the method is more apparent. In the rest of this chapter, these definitions and results are extended to the higher-order case.

6.2.1 Dependency Pairs

The motivation of the dependency pair approach is to be able to deal with systems which cannot be handled with a quasi-simplification ordering. For example the following TRS for division, which includes the rule we saw in the introduction.

$$\begin{array}{rcl} \mathtt{minus}(x, \mathtt{0}) & \Rightarrow & x \\ \mathtt{minus}(\mathtt{s}(x), \mathtt{s}(y)) & \Rightarrow & \mathtt{minus}(x, y) \\ \mathtt{quot}(\mathtt{0}, \mathtt{s}(y)) & \Rightarrow & \mathtt{0} \\ \mathtt{quot}(\mathtt{s}(x), \mathtt{s}(y)) & \Rightarrow & \mathtt{s}(\mathtt{quot}(\mathtt{minus}(x, y), \mathtt{s}(y))) \end{array}$$

An intuition behind the dependency pair approach is to identify those parts of the right-hand sides of rewrite rules which may give rise to an infinite reduction.

Suppose we have a *minimal non-terminating* term t_0, so a non-terminating term where all proper subterms are terminating. Consider an infinite reduction. After finitely many internal steps, there must be a step at the top, $l\gamma \to r\gamma$. If we identify a minimal non-terminating subterm t_1 of $r\gamma$, we find that its root symbol f occurs in r, and moreover, that f is a *defined symbol*, which means that a rule $f(\vec{l}) \Rightarrow r$ exists. From t_1, we can continue this reasoning, and end up with an infinite sequence which uses special "rule with subterm" steps at the top of terms.

This reasoning suggests that we should be particularly interested in subterms of right-hand sides of rewrite rules with a defined symbol at the root. Following [112], we call such subterms *candidate terms* of r.

Definition 6.1 (First-order Dependency Pairs ([9, 31])). Let $(\mathcal{F}, \mathcal{R})$ be a first-order TRS. Let $\mathcal{F}^\sharp$ denote the union of $\mathcal{F}$ with the set which contains (exactly) a fresh *marked* symbol $f^\sharp$ for every defined symbol $f \in \mathcal{F}$; $f^\sharp$ has the same arity as f. The *dependency pairs* of a rewrite rule $f(l_1, \ldots, l_n) \Rightarrow r$ are all pairs of the form $f^\sharp(l_1, \ldots, l_n) \Rrightarrow g^\sharp(p_1, \ldots, p_m)$ with $r \trianglerighteq g(p_1, \ldots, p_m)$ and g a defined symbol, where $g(\vec{p})$ is not a subterm of one of the l_i. The set of all dependency pairs of $(\mathcal{F}, \mathcal{R})$ is denoted by $\mathsf{DP}(\mathcal{R})$.

Example 6.2. The `quot`-example has the following dependency pairs:

$$\begin{array}{rcl} \mathtt{minus}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) & \Rrightarrow & \mathtt{minus}^\sharp(x, y) \\ \mathtt{quot}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) & \Rrightarrow & \mathtt{quot}^\sharp(\mathtt{minus}(x, y), \mathtt{s}(y)) \\ \mathtt{quot}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) & \Rrightarrow & \mathtt{minus}^\sharp(x, y) \end{array}$$

The first and third rewrite rule do not give dependency pairs, because their right-hand sides do not contain defined symbols. The fourth rule gives two different dependency pairs.

6.2.2 Dependency Chains

Dependency pairs are used in a *dependency chain*, a sequence $[(l_i \Rrightarrow p_i, s_i, t_i) \mid i \in \mathbb{N}]$, such that for all i:

1. $l_i \Rrightarrow p_i \in \mathsf{DP}(\mathcal{R})$;
2. $s_i = l_i\gamma_i$ and $t_i = p_i\gamma_i$ for some substitution γ_i;
3. $t_i \Rightarrow_\mathcal{R}^* s_{i+1}$;

A dependency chain is *minimal* if the strict subterms of all s_j, t_j are terminating in $\Rightarrow_\mathcal{R}$. Note that, since t_i has the form $f^\sharp(\vec{q})$ and the marked symbol $f^\sharp$ is not used in any rule, the reduction $t_i \Rightarrow_\mathcal{R}^* s_{i+1}$ takes place in strict subterms.

Termination of a TRS can be studied by examining dependency chains:

Theorem 6.3 (First-order Dependency Chains [9]). *A TRS is terminating* if and only if *it does not admit a minimal dependency chain.*

The intuition of how a minimal dependency chain is constructed was given in Section 6.2.1.

Comment: About terminology: this definition of a dependency chain, and also its naming, is not a standard. In the literature, our dependency chains are also called *infinite $\mathcal{R}$-chains* [9]. In the language of [119] we might refer to an infinite $(\mathsf{DP}(\mathcal{R}), \emptyset, \mathcal{R})$-chain.
However, in these definitions, the chains are merely chains of dependency pairs; s_i and t_i are not included. In the higher-order definitions we will need these separate terms. Hence the different naming. Since we are not interested in finite chains, the "infinite" adjective is omitted – a dependency chain is infinite by definition.

Example 6.4. Consider a TRS which enumerates the natural numbers with a rule $\mathtt{nats}(n) \Rightarrow \mathtt{cons}(n, \mathtt{nats}(\mathtt{s}(n)))$. This TRS admits a dependency chain:

$$\begin{array}{llllll} (& \mathtt{nats}^\sharp(n) \Rrightarrow \mathtt{nats}^\sharp(\mathtt{s}(n)) & , & \mathtt{nats}^\sharp(\mathtt{0}) & , & \mathtt{nats}^\sharp(\mathtt{s}(\mathtt{0})) &), \\ (& \mathtt{nats}^\sharp(n) \Rrightarrow \mathtt{nats}^\sharp(\mathtt{s}(n)) & , & \mathtt{nats}^\sharp(\mathtt{s}(\mathtt{0})) & , & \mathtt{nats}^\sharp(\mathtt{s}(\mathtt{s}(\mathtt{0}))) &), \\ & \ldots & & & & & \end{array}$$

Thus, the system is non-terminating. This chain corresponds with the reduction $\mathtt{nats}(\mathtt{0}) \Rightarrow \mathtt{cons}(\mathtt{0}, \mathtt{nats}(\mathtt{s}(\mathtt{0})) \Rightarrow \mathtt{cons}(\mathtt{0}, \mathtt{cons}(\mathtt{s}(\mathtt{0}), \mathtt{nats}(\mathtt{s}(\mathtt{s}(\mathtt{0}))) \Rightarrow \ldots$

6.2.3 Using a Reduction Pair

If a TRS has no (minimal or non-minimal) dependency chain, then it is terminating. Absence of infinite dependency chains can be demonstrated with a *weak reduction pair*, which we have previously seen in Chapter 2.4. Recall that a (weak) reduction pair for first-order rewriting is a pair $(\succsim, \succ)$ of a quasi-ordering and a compatible well-founded ordering, such that $\succsim$ and $\succ$ are both stable, and $\succsim$ is monotonic, but $\succ$ need not be monotonic.

Theorem 6.5 ([8, 88])**.** *$\mathcal{R}$ is terminating if and only if there is a reduction pair $(\succsim, \succ)$ such that $l \succ p$ for all dependency pairs $l \Rrightarrow p$, and $l \succsim r$ for all rules $l \Rightarrow r$.*

The *if* part holds because, if the requirements are satisfied, there cannot be a dependency chain: each $s_i \succ t_i \succsim s_{i+1}$, (the $\succ$ step occurs at the top, so monotonicity is not needed). As the dependency chain is infinite, this leads to an infinite decreasing $\succ$-reduction.

For the *only if* part, we can define a reduction pair based on $\Rightarrow^*_{\mathcal{R}}$ and $\Rrightarrow$.

Example 6.6. The `quot` example is terminating if there is a reduction pair satisfying:

$$\begin{array}{rcl} \mathtt{minus}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) & \succ & \mathtt{minus}^\sharp(x, y) \\ \mathtt{quot}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) & \succ & \mathtt{quot}^\sharp(\mathtt{minus}(x, y), \mathtt{s}(y)) \\ \mathtt{quot}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) & \succ & \mathtt{minus}^\sharp(x, y) \\ \mathtt{minus}(x, \mathtt{0}) & \succsim & x \\ \mathtt{minus}(\mathtt{s}(x), \mathtt{s}(y)) & \succsim & \mathtt{minus}(x, y) \\ \mathtt{quot}(\mathtt{0}, \mathtt{s}(y)) & \succsim & \mathtt{0} \\ \mathtt{quot}(\mathtt{s}(x), \mathtt{s}(y)) & \succsim & \mathtt{s}(\mathtt{quot}(\mathtt{minus}(x, y), \mathtt{s}(y))) \end{array}$$

Recall from Theorem 4.10 that polynomial interpretations without the strong monotonicity requirement give a weak reduction pair; this is also the case in first-order rewriting. Thus, we can orient the constraints with a polynomial interpretation. We consider an interpretation over the natural numbers with: $\mathcal{J}(\mathtt{minus}) = \mathcal{J}(\mathtt{minus}^\sharp) = \boldsymbol{\lambda} xy.x$, $\mathcal{J}(\mathtt{quot}) = \mathcal{J}(\mathtt{quot}^\sharp) = \boldsymbol{\lambda} xy.x + y$, $\mathcal{J}(\mathtt{s}) =$

$\boldsymbol{\lambda}x.x+1$, $\mathcal{J}(\mathtt{0})=0$. This gives the following constraints:

$$\begin{array}{rcl} x+1 & > & x \\ x+y+2 & > & x+y+1 \\ x+y+2 & > & x \end{array} \qquad \begin{array}{rcl} x & \geq & x \\ x+1 & \geq & x \\ y+1 & \geq & 0 \\ x+y+2 & \geq & x+y+2 \end{array}$$

It is clear that these constraints are satisfied for all valuations. Note that the resulting reduction pair is not a strong reduction pair, as $\succ$ is not monotonic: $[\![\mathtt{minus}(s,t)]\!] = [\![\mathtt{minus}(s,q)]\!]$ even if $t \succ q$.

The requirements for Theorem 6.5 are somewhat stronger than we need. As with rule removal, rather than orienting all dependency pairs at once, we can use a step-by-step approach. Let a set of dependency pairs $\mathcal{P}$ be called *chain-free* if there is no minimal dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ with all $\rho_i \in \mathcal{P}$. Since evidently $\emptyset$ is chain-free, the task of a termination prover is to iterate over $\mathcal{P}$ until no dependency pairs remain. This we can also do with a reduction pair:

Theorem 6.7 (Based on [43, 53])**.** *A set of dependency pairs $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ is chain-free if $\mathcal{P}_2$ is chain-free, and there is a reduction pair $(\succsim, \succ)$ such that $l \succ p$ for all dependency pairs $l \Rrightarrow p \in \mathcal{P}_1$ and $l \succsim p$ for all $l \Rrightarrow p \in \mathcal{P}_2$ and $l \succsim r$ for all rules $l \Rightarrow r \in \mathcal{R}$.*

> *Comment:* The notion *chain-free* actually does not appear in the first-order literature. It is based on the notion of *non-loopingness* in [112]. The first-order results presented in this section are adapted from their original definitions (but can also be seen as a consequence of some results in the dependency pair framework discussed in Chapter 7.1.1).

Example 6.8. Let us try to prove termination of the following TRS for addition and multiplication:

$$\begin{array}{rcl} \mathtt{add}(\mathtt{0}, y) & \Rightarrow & y \\ \mathtt{add}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{s}(\mathtt{add}(x, y)) \\ \mathtt{mul}(\mathtt{0}, y) & \Rightarrow & \mathtt{0} \\ \mathtt{mul}(\mathtt{s}(x), y) & \Rightarrow & \mathtt{add}(y, \mathtt{mul}(x, y)) \end{array}$$

This TRS has three dependency pairs:

$$\begin{array}{rcl} \mathtt{add}^\sharp(\mathtt{s}(x), y) & \Rrightarrow & \mathtt{add}^\sharp(x, y) \\ \mathtt{mul}^\sharp(\mathtt{s}(x), y) & \Rrightarrow & \mathtt{add}^\sharp(y, \mathtt{mul}(x, y)) \\ \mathtt{mul}^\sharp(\mathtt{s}(x), y) & \Rrightarrow & \mathtt{mul}^\sharp(x, y) \end{array}$$

To prove termination, Theorem 6.3 states that it suffices if the set of these three pairs is chain-free. Using Theorem 6.7, we must orient the following constraints:

$$\begin{array}{lll} \mathtt{add}^\sharp(\mathtt{s}(x),y) & {}_{(}\succsim_{)} & \mathtt{add}^\sharp(x,y) \\ \mathtt{mul}^\sharp(\mathtt{s}(x),y) & {}_{(}\succsim_{)} & \mathtt{add}^\sharp(y,\mathtt{mul}(x,y)) \\ \mathtt{mul}^\sharp(\mathtt{s}(x),y) & {}_{(}\succsim_{)} & \mathtt{mul}^\sharp(x,y) \end{array} \qquad \begin{array}{rcl} \mathtt{add}(\mathtt{0},y) & \succsim & y \\ \mathtt{add}(\mathtt{s}(x),y) & \succsim & \mathtt{s}(\mathtt{add}(x,y)) \\ \mathtt{mul}(\mathtt{0},y) & \succsim & \mathtt{0} \\ \mathtt{mul}(\mathtt{s}(x),y) & \succsim & \mathtt{add}(y,\mathtt{mul}(x,y)) \end{array}$$

Here, ${}_{(}\succsim_{)}$ indicates that the constraints may either be oriented with $\succsim$ or with $\succ$, but at least one should be oriented with $\succ$. Let's try the intuitively logical polynomial interpretation, with $\mathcal{J}(\mathtt{0}) = 0$, $\mathcal{J}(\mathtt{s}) = \boldsymbol{\lambda} n.n+1$, $\mathcal{J}(\mathtt{add}) = \boldsymbol{\lambda} nm.n+m$, $\mathcal{J}(\mathtt{mul}) = \boldsymbol{\lambda} nm.n \cdot m$. Moreover, choose $\mathcal{J}(\mathtt{add}^\sharp) = \boldsymbol{\lambda} nm.n$ and $\mathcal{J}(\mathtt{mul}^\sharp) = \boldsymbol{\lambda} nm.m$. All interpretations are weakly monotonic functions, and the constraints are satisfied:

$$\begin{array}{rcl} x+1 & > & x \\ y & \geq & y \\ y & \geq & y \end{array} \qquad \begin{array}{rcl} y & \geq & y \\ x+y+1 & \geq & x+y+1 \\ 0 & \geq & 0 \\ x \cdot y + y & \geq & x \cdot y + y \end{array}$$

The first of the dependency pairs was oriented with $\succ$ and the rest with $\succsim$, so we must prove that the set $\{\mathtt{mul}^\sharp(\mathtt{s}(x),y) \Rrightarrow \mathtt{add}^\sharp(y,\mathtt{mul}(x,y)), \mathtt{mul}^\sharp(\mathtt{s}(x),y) \Rrightarrow \mathtt{mul}^\sharp(x,y)\}$ is chain-free. To prove this, we might use any reduction pair, but let us use polynomial interpretations again, with the same interpretation for $\mathtt{0}, \mathtt{s}, \mathtt{add}$ and $\mathtt{mul}$, but now choosing $\mathcal{J}(\mathtt{add}^\sharp) = \boldsymbol{\lambda} nm.0$ and $\mathcal{J}(\mathtt{mul}^\sharp) = \boldsymbol{\lambda} nm.n$. Then $[\![l]\!]_{\mathcal{J},\alpha} = [\![r]\!]_{\mathcal{J},\alpha}$ for all rules as before, and moreover:

$$\begin{array}{rcl} [\![\mathtt{mul}^\sharp(\mathtt{s}(x),y)]\!]_{\mathcal{J},\alpha} & = & x+1 > 0 = [\![\mathtt{add}^\sharp(y,\mathtt{mul}(x,y))]\!]_{\mathcal{J},\alpha} \\ [\![\mathtt{mul}^\sharp(\mathtt{s}(x),y)]\!]_{\mathcal{J},\alpha} & = & x+1 > x = [\![\mathtt{mul}^\sharp(x,y)]\!]_{\mathcal{J},\alpha} \end{array}$$

Thus, termination is reduced to chain-freeness of $\emptyset$, which we know to hold; the system is terminating.

6.2.4 Argument Filterings

In order to obtain a reduction pair which does not necessarily have the subterm property, there are two ways we could go: either we use approaches like the polynomial interpretations given in Example 6.6 and 6.8, which directly give us a pair $(\succsim, \succ)$ where $\succ$ may be non-monotonic, or we use an existing strong reduction pair and adapt it with *argument filterings*.

An argument filtering is a function π which maps terms of the form $f(x_1, \ldots, x_n)$ with $f \in \mathcal{F}^\sharp$ either to a term $f_\pi(x_{i_1}, \ldots, x_{i_m})$ or to one of the x_i. An argument filtering is applied to a term as follows:

$$\begin{array}{rcll} \overline{\pi}(f(s_1,\ldots,s_n)) & = & f_\pi(\overline{\pi}(s_{i_1}),\ldots,\overline{\pi}(s_{i_m})) & \text{if } \pi(f(\vec{x})) = f_\pi(x_{i_1},\ldots,x_{i_m}) \\ \overline{\pi}(f(s_1,\ldots,s_n)) & = & \overline{\pi}(s_i) & \text{if } \pi(f(\vec{x})) = x_i \\ \overline{\pi}(x) & = & x & \text{if } x \text{ a variable} \end{array}$$

Phrased differently, $\overline{\pi}(f(s_1,\ldots,s_n)) = \pi(f(x_1,\ldots,x_n))[x_1 := \overline{\pi}(s_1),\ldots,x_n := \overline{\pi}(s_n)]$. Note that an argument filtering works both on unmarked and on marked symbols. Using the symbols from the `quot` example, suppose $\pi(\mathtt{minus}(x,y)) = x$ and $\pi(\mathtt{quot}(x,y)) = \mathtt{quot}_\pi(x)$ and $\pi(\mathtt{s}(x)) = \mathtt{s}_\pi(x)$. Then we for example have: $\overline{\pi}(\mathtt{s}(\mathtt{quot}(\mathtt{minus}(x,y),\mathtt{s}(y)))) = \mathtt{s}_\pi(\mathtt{quot}_\pi(x))$.

Using argument filterings, we can eliminate troublesome subterms of the dependency pair constraints. To this end, we have the following result:

Theorem 6.9 (Based on [88])**.** *Let $(\succsim, \succ)$ be a reduction pair on terms. Let π be an argument filtering, and define $s \succsim_\pi t$ iff $\overline{\pi}(s) \succsim \overline{\pi}(t)$ and $s \succ_\pi t$ iff $\overline{\pi}(s) \succ \overline{\pi}(t)$. Then $(\succsim_\pi, \succ_\pi)$ is a reduction pair.*

Argument filterings are a special case of the argument functions from Chapter 5.6. They can be used with an arbitrary reduction pair, rather than just the recursive path ordering.

Example 6.10. Recall the constraints given in Example 6.6. Using Theorem 6.9 and the argument filtering with $\pi(\mathtt{minus}(x,y)) = x$, $\pi(\mathtt{quot}(x,y)) = \mathtt{quot}_\pi(x)$ and $\pi(f(\vec{x})) = f_\pi(\vec{x})$ for all other symbols, it suffices to find a strong reduction pair satisfying:

$$\begin{array}{rcl}
x & \succsim & x \\
\mathtt{s}_\pi(x) & \succsim & x \\
\mathtt{quot}_\pi(\mathtt{0}_\pi) & \succsim & \mathtt{0}_\pi \\
\mathtt{quot}_\pi(\mathtt{s}_\pi(x)) & \succsim & \mathtt{s}_\pi(\mathtt{quot}_\pi(x))
\end{array}
\qquad
\begin{array}{rcl}
\mathtt{minus}^\sharp_\pi(\mathtt{s}_\pi(x),\mathtt{s}_\pi(y)) & \succ & \mathtt{minus}^\sharp_\pi(x,y) \\
\mathtt{quot}^\sharp_\pi(\mathtt{s}_\pi(x),\mathtt{s}_\pi(y)) & \succ & \mathtt{quot}^\sharp_\pi(x,\mathtt{s}_\pi(y)) \\
\mathtt{quot}^\sharp_\pi(\mathtt{s}_\pi(x),\mathtt{s}_\pi(y)) & \succ & \mathtt{minus}^\sharp_\pi(x,y)
\end{array}$$

These altered constraints are easily satisfied with a recursive path ordering.

6.2.5 Usable Rules

To prove that a set of dependency pairs $\mathcal{P}$ is chain-free using a reduction pair, so far we always have to show that $l \succsim r$ for all rules $l \Rightarrow r$. Thus, even when we have only a single dependency pair, we may have many ordering constraints. The method of *usable rules* comes to the rescue. The idea of this method is that in a dependency chain over $\mathcal{P}$, we can restrict attention to those rules that may actually be relevant in the reduction $t_i \Rightarrow^*_{\mathcal{R}} s_i$.

First we need some definitions. Let $f \sqsupseteq_{us} g$ denote that there is a rewrite rule $f(l_1,\ldots,l_n) \Rightarrow C[g(r_1,\ldots,r_m)]$. The reflexive-transitive closure of $\sqsupseteq_{us}$ is denoted by $\sqsupseteq^*_{us}$. Overloading notation, let $s \sqsupseteq^*_{us} g$ indicate that there is a symbol f in the term s such that $f \sqsupseteq^*_{us} g$. So if not $s \sqsupseteq^*_{us} g$, then s cannot reduce to a term containing the symbol g.

Definition 6.11. The set of *usable rules* of a term s, notation $UR(s)$, is the set of rules $g(\vec{l}) \Rightarrow r \in \mathcal{R}$, where g is any symbol such that $s \sqsupseteq^*_{us} g$. For a set of dependency pairs $\mathcal{P}$, let $UR(\mathcal{P}) = \bigcup_{l \Rrightarrow p \in \mathcal{P}} UR(p)$.

Using a reasoning originally due to Gramlich [51], and following on results from Urbain [123] and a definition for innermost rewriting [9], the authors of both [53] and [48] (independently) demonstrate that if there is a minimal dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ over $\mathcal{P}$, then there is a (not-necessarily-minimal) dependency chain $[(\rho_i, s'_i, t'_i) \mid i \in \mathbb{N}]$ over $\mathcal{P}$ where the reduction $t_i \Rightarrow^* s_{i+1}$ uses only the rules in $UR(\mathcal{P}) \cup \{\mathtt{p}(x, y) \Rightarrow x,\ \mathtt{p}(x, y) \Rightarrow y\}$ for a fresh symbol $\mathtt{p}$ (these two rules are usually considered harmless, as methods like the recursive path ordering or polynomial interpretations trivially orient them). If we can see (using for instance a reduction pair) that the latter cannot use a certain dependency pair infinitely often, then the same holds for the former.

Example 6.12. Let $\mathcal{P} = \{\mathtt{quot}^\sharp(\mathtt{s}(x), \mathtt{s}(y)) \Rrightarrow \mathtt{quot}^\sharp(\mathtt{minus}(x, y), \mathtt{s}(y))\}$. For each of the symbols f occurring in the right-hand side of the one dependency pair in this set, only $f \sqsupseteq^*_{us}$ itself. Thus, the usable rules of this set are only the two `minus` rules,

$$\begin{array}{rcl} \mathtt{minus}(x, \mathtt{0}) & \Rightarrow & x \\ \mathtt{minus}(\mathtt{s}(x), \mathtt{s}(y)) & \Rightarrow & \mathtt{minus}(x, y) \end{array}$$

Comment: Usable rules occur with slightly different definitions in different places. The definition here corresponds to the one in [53], but stronger definitions (which give fewer usable rules) are possible. All existing definitions can be seen as *approximations* of the general definition of usable rules given in [119], which we will not consider here.

6.2.6 Concluding Remarks

Thus we see a simple algorithm to prove termination using dependency pairs:

- calculate the dependency pairs, and let $\mathcal{P} := \mathsf{DP}(\mathcal{R})$;
- while $\mathcal{P}$ is non-empty:
  - find a weak reduction pair such that $l \succ p$ for one or more pair $l \Rrightarrow p \in \mathcal{P}$ and $l \succsim p$ for the rest, and also $l \succsim r$ for all rules in $UR(\mathcal{P})$ (this pair could for example be chosen with weakly monotonic polynomial interpretations, or with RPO with an argument filtering);
  - remove those pairs $l \Rrightarrow p$ which were oriented with $\succ$ from $\mathcal{P}$
- conclude that the system is terminating!

Typical implementations of dependency pairs use more sophisticated techniques, such as a graph of dependency pairs and the subterm criterion. These will be discussed in Chapter 7. For now, let us restrict ourselves to the challenge of extending this basis to the higher-order setting.

6.3 The Unrestricted Dynamic Dependency Pair approach

When trying to extend the first-order dependency pair approach to AFSMs, we run into several new issues:

- collapsing rules: non-termination might be caused by a meta-variable application. For example, the right-hand side of the non-terminating rule $\mathtt{f}(\mathtt{g}(\lambda x.F(x)), X) \Rightarrow F(X)$ doesn't even have defined symbols;
- dangling variables: given a rule $\mathtt{f}(\mathtt{0}) \Rightarrow \mathtt{g}(\lambda x.\mathtt{f}(x))$, the bound variable x may become free in the corresponding dependency pair;
- rules of functional type may lead to non-termination only because of their interaction with the (applicative) context they appear in, such as a rule $\mathtt{A}(\mathtt{B}(F)) \Rightarrow F$ (with $\mathtt{A} : [\mathtt{o}] \longrightarrow \mathtt{o} \to \mathtt{o}$ and $\mathtt{B} : [(\mathtt{o} \to \mathtt{o})] \to \mathtt{o}$);
- typing issues: to be able to use polynomial interpretations or path orderings, both sides of a dependency pair (or the constraints generated from it) should usually have the same type modulo renaming of base types.

Typing issues will be addressed in Section 6.3.4. For the other problems we have to take precautions already in the definition of dependency pairs.

Example 6.13. The following AFSM `twice` (which appears under the same name in the termination problem database v.8.0.1) is the running example of this chapter. Its signature consists of four function symbols: $\mathtt{0} : \mathtt{nat}$, $\mathtt{s} : [\mathtt{nat}] \to \mathtt{nat}$, $\mathtt{I} : [\mathtt{nat}] \to \mathtt{nat}$, and $\mathtt{twice} : [\mathtt{nat} \to \mathtt{nat}] \to \mathtt{nat} \to \mathtt{nat}$. There are three rewrite rules:

$$\begin{array}{rcl} \mathtt{I}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\ \mathtt{I}(\mathtt{s}(X)) & \Rightarrow & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\ \mathtt{twice}(F) & \Rightarrow & \lambda y.F \cdot (F \cdot y) \end{array}$$

The symbol `I` represents the identity function on natural numbers; `twice` runs a given function two times on any argument. Although this system is terminating, this is not trivial to prove: higher-order path orderings have trouble with the $\mathtt{I}(x)$ subterm of the second rule, polynomial interpretations in $\mathbb{N}$ struggle because of the $\cdot$ in the same rule, and a static dependency pair approach fails because it is impossible to prove $\mathtt{I}^\sharp(\mathtt{s}(X)) \succ \mathtt{I}^\sharp(Y)$ with a meta-stable and well-founded ordering relation.

Example 6.14. Another example that will be used regularly is the AFSM `eval`:

$$\begin{array}{lcl} \mathtt{0} & : & \mathtt{nat} \\ \mathtt{s} & : & [\mathtt{nat}] \longrightarrow \mathtt{nat} \end{array} \qquad \begin{array}{lcl} \mathtt{dom} & : & [\mathtt{nat} \times \mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{nat} \\ \mathtt{fun} & : & [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{nat} \\ \mathtt{eval} & : & [\mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{nat} \end{array}$$

$$\begin{array}{rcl}
\mathtt{dom}(\mathtt{s}(X),\mathtt{s}(Y),\mathtt{s}(Z)) & \Rightarrow & \mathtt{s}(\mathtt{dom}(X,Y,Z)) \\
\mathtt{dom}(\mathtt{0},\mathtt{s}(Y),\mathtt{s}(Y)) & \Rightarrow & \mathtt{s}(\mathtt{dom}(\mathtt{0},Y,Z)) \\
\mathtt{dom}(X,Y,\mathtt{0}) & \Rightarrow & X \\
\mathtt{dom}(\mathtt{0},\mathtt{0},Z) & \Rightarrow & \mathtt{0} \\
\mathtt{eval}(\mathtt{fun}(\lambda x.F(x),X,Y),Z) & \Rightarrow & F(\mathtt{dom}(X,Y,Z))
\end{array}$$

This system encodes a function, together with an application domain, as a natural number. The `dom` function makes sure that a function is only applied to elements of the domain it is restricted to. Note that, unlike `twice`, this AFSM has meta-variables which take arguments. This system is difficult because of the last rule: if we could orient this rule with a reduction pair that satisfies $\mathtt{dom}(X,Y,Z) \succsim Z$, we would have $\mathtt{eval}(\omega,\omega) \succ \mathtt{eval}(\mathtt{dom}(x,y,\omega),\mathtt{dom}(x,y,\omega)) \succsim \mathtt{eval}(\omega,\omega) \succ \ldots$ for $\omega := \mathtt{fun}(\lambda z.\mathtt{eval}(z,z),x,y)$.

6.3.1 Dependency Pairs

In this section we define dependency pairs, after pre-processing the rules and defining candidate terms. The complete definition is a fair bit more complicated than its first-order counterpart, and may at first seem somewhat baroque. This is partly because we have to work around the issues of functional rules and dangling variables, and partly because of several optimisations which are immediately included to obtain an easier result system, and a *complete* method.

To start, $\mathcal{F}$ is split into two subsets: the set of *defined symbols*, denoted $\mathcal{D}$, and the set of *constructor symbols*, denoted $\mathcal{C}$. Defined symbols are those symbols f such that a rule $f(l_1,\ldots,l_m)\cdot l_{m+1}\cdots l_n \Rightarrow r$ exists; all other symbols are constructor symbols.

Pre-processing. Before defining dependency pairs, we alter the system in the following way:

Definition 6.15 (β-saturating Rules)**.** An AFSM is *β-saturated* in two steps. First we add for all rules of the form $l \Rightarrow \lambda\vec{x}.((\lambda y.s)\cdot t\cdot\vec{q})$ where l has a functional type, a new rule $l \Rightarrow \lambda\vec{x}.(s[y := t]\cdot\vec{q})$, and iteratively repeat this with the new rule until it no longer has such a form. Then we add for each rule of the form $l \Rightarrow \lambda x_1\ldots x_n.r$ with r not an abstraction the following n rules: $l\cdot Z_1 \Rightarrow \lambda x_2\ldots x_n.r[x_1 := Z_1],\ \ldots,\ l\cdot Z_1\cdots Z_n \Rightarrow r[x_1 := Z_1,\ldots,x_n := Z_n]$.

Note that β-saturation has no effect on termination, since the added rules can be simulated with the original rules and some β-steps. No rules are removed.

Example 6.16. The system `twice` from Example 6.13 is β-saturated by adding the rule $\mathtt{twice}(F)\cdot X \Rightarrow F\cdot(F\cdot X)$. The system `eval` has only base-type, β-normal rules, so β-saturating does not add anything.

To understand why β-saturation is necessary, consider the following example:

Example 6.17. Let $\mathcal{R} = \{\mathtt{f}(\mathtt{0}) \Rightarrow \lambda x.\mathtt{f}(x) \cdot x\}$. The term $\mathtt{f}(\mathtt{0})$ itself is terminating. However, there *is* an infinite reduction involving this rule: $\mathtt{f}(\mathtt{0}) \cdot \mathtt{0} \Rightarrow (\lambda x.\mathtt{f}(x) \cdot x) \cdot \mathtt{0} \Rightarrow_\beta \mathtt{f}(\mathtt{0}) \cdot \mathtt{0} \Rightarrow \ldots$

Rules like the one in Example 6.17 might complicate the analysis of dependency chains, because the important $\Rightarrow_\mathcal{R}$-step does not happen at the top. The preprocessing makes sure that it could also be done with a topmost step: $f(\mathtt{0}) \cdot \mathtt{0}$ self-reduces with a single step using the new rule $f(\mathtt{0}) \cdot x \Rightarrow f(x) \cdot x$ which was added by β-saturation.

Comment: It is worth noting that we did not add new rules for rules of functional type, only for those where the right-hand side is an abstraction (or might β-reduce to an abstraction). A rule $\mathtt{f}(\mathtt{0}) \Rightarrow \mathtt{f}(\mathtt{A})$ of functional type is left alone. This is an optimisation: it would be natural to add a rule $\mathtt{f}(\mathtt{0}) \cdot x \Rightarrow \mathtt{f}(\mathtt{A}) \cdot x$, but this might give a dependency pair $\mathtt{f}(\mathtt{0}) \cdot x \Rrightarrow \mathtt{A}^\sharp$ which will not be needed. Instead of "saturating" this rule, we will later add a special dependency pair for it. Apart from optimising, this choice was made with an eye on a future extension to polymorphic systems: the given definition of β-saturation only adds finitely many rules even in the presence of polymorphic types, while the simpler alternative does not.

Candidate Terms. In the first-order definition of dependency pairs, we identified subterms that may give rise to an infinite reduction. Taking subterms in a system with binders is well-known to be problematic because bound variables may become free. However, in the setting of AFSMs this is not too troublesome, because variables are not meta-variables, and can be treated differently.

Something that we should also watch out for is β-reduction. As a design decision, our dependency pairs will always have the form $l \Rrightarrow s \cdot \vec{t}$ where s is either a functional term, or a meta-variable application, and $\vec{t}$ has length ≥ 0. Notably, we will not have dependency pairs headed by a λ-abstraction. Not because dependency pairs of the form $l \Rrightarrow (\lambda x.q) \cdot \vec{t}$ would cause insurmountable problems (they wouldn't), but because pairs like this would give an extra case every time we prove something about dependency pairs – and they are easy to avoid, by taking some care in the definition.

Moreover, terms which occur below a meta-variable may be destroyed in a substitution of the term. If we want to achieve a complete method, we must keep track of the way meta-variables are used.

These considerations lead to the following notion of candidate terms:

Definition 6.18 (Candidate Terms)**.** The set of *β-reduced sub-meta-terms* of a meta-term s, denoted $\mathsf{brsmt}_\emptyset(s)$, is given by the following inductive definition:

$$\begin{array}{rcl}
\mathsf{brsmt}_A(\lambda x.s) & = & \mathsf{brsmt}_A(s) \\
\mathsf{brsmt}_A(f(s_1,\ldots,s_n)) & = & \{f(\vec{s})\ (A)\} \cup \bigcup_{i=1}^{n} \mathsf{brsmt}_A(s_i) \\
\mathsf{brsmt}_A(Z(s_1,\ldots,s_n)) & = & \{Z(\vec{s})\ (A)\} \cup \bigcup_{i=1}^{n} \mathsf{brsmt}_{A\cup\{Z:i\}}(s_i) \\
\mathsf{brsmt}_A(s\cdot t) & = & \{s\cdot t\ (A)\} \cup \mathsf{brsmt}_A(s) \cup \mathsf{brsmt}_A(t) \\
 & & (\text{if } \mathtt{head}(s) \text{ is not an abstraction}) \\
\mathsf{brsmt}_A((\lambda x.s)\cdot t_0\cdots t_n) & = & \mathsf{brsmt}_A(s[x:=t_0]\cdot t_1\cdots t_n) \cup \mathsf{brsmt}_A(t_0) \\
\mathsf{brsmt}_A(x) & = & \emptyset
\end{array}$$

This function is defined for all terms, as we can see with induction on $\Rightarrow_\beta$ combined with the subterm relation. The set A is used to keep track of the meta-variables below which the given β-reduced sub-meta-term occurs. If s is β-normal, then $\mathsf{brsmt}_\emptyset(s)$ contains pairs $t\ (A)$ where t is a sub-meta-term of s which is not an abstraction, and A is a set of conditions which indicate the meta-variable applications inside which t occurs. The set A will be referred to as a set of *meta-variable conditions*.

The "candidate terms" of a meta-term r are those β-reduced sub-meta-terms $s\ (A)$ of r where either:

- $s = f(t_1,\ldots,t_m)\cdot q_1\cdots q_n$ with f a defined symbol and $n \geq 0$, or
- $s = F(t_1,\ldots,t_m)\cdot q_1\cdots q_n$ with F a meta-variable and $n \geq 0$, and either $n > 0$, or $t_1,\ldots,t_m$ are not all distinct variables (that is, a meta-variable application of the form $F(x_1,\ldots,x_m)$ where either all x_i are distinct variables (or $m = 0$) is not a candidate term, but all other sub-meta-terms of the form $F(\vec{t})\cdot\vec{q}$ are).

The set of candidate terms of r is denoted by $Cand(r)$. Note that a single meta-variable is never a candidate term.

Example 6.19. In the `twice` system we have:

$$Cand(F\cdot(F\cdot x)) = \{F\cdot(F\cdot x)\ (\emptyset),\ \ F\cdot x\ (\emptyset)\}$$

$$Cand(F\cdot(F\cdot X)) = \{F\cdot(F\cdot X)\ (\emptyset),\ \ F\cdot X\ (\emptyset)\}$$

Because the system is applicative, there are no meta-variable restrictions, and $F\cdot x$ is included. If these were meta-variables applications rather than normal applications, we would have:

$$Cand(F(F(x))) = \{F(F(x))\ (\emptyset)\}\ \text{(but not } F(x)\text{!)}$$

$$Cand(F(F(X))) = \{F(F(X))\ (\emptyset),\ \ F(X)\ (\{F:1\})\}$$

In the `twice` system we also have:

$$Cand(\mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X)) = \left\{ \begin{array}{ll} \mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X & (\emptyset) \\ \mathtt{twice}(\lambda x.\mathtt{I}(x)) & (\emptyset) \\ \mathtt{I}(x) & (\emptyset) \end{array} \right\}$$

If f is a defined symbol, then the candidate terms of $f(a) \cdot b \cdot c \cdot d$ are $f(a)$, $f(a) \cdot b$, $f(a) \cdot b \cdot c$, $f(a) \cdot b \cdot c \cdot d$. Note that for example $x \cdot y$ is not a candidate term of $g(\lambda x.x \cdot y)$ because x is a variable, not a meta-variable.

By α-conversion, a term with bound variables may have infinitely many candidate terms. For instance a term $f(\lambda x.g(x))$ with g a defined symbol has a candidate term $g(x)$, but also $g(y), g(z), \ldots$ For the sake of sanity, let us assume we have chosen one representative for every class of candidate terms which is equal modulo renaming of free variables. Thus, a single term only has finitely many candidate terms.

Dependency Pairs. As in the first-order case, the definition of dependency pair uses marked function symbols.

Let $\mathcal{F}^\sharp = \mathcal{F} \cup \{f^\sharp : \sigma \mid f : \sigma \in \mathcal{D}\}$, so $\mathcal{F}$ extended with for every defined symbol f a marked version $f^\sharp$ with the same type declaration. The marked counterpart of a term s, notation $s^\sharp$, is $f^\sharp(s_1, \ldots, s_n)$ if $s = f(s_1, \ldots, s_n)$ with f in $\mathcal{D}$, and just s otherwise. For example, $(\mathtt{twice}(F))^\sharp = \mathtt{twice}^\sharp(F)$ and $(\mathtt{twice}(F) \cdot m)^\sharp = \mathtt{twice}(F) \cdot m$. The application operator is not a function symbol, so is not marked.

Definition 6.20. A *dependency pair* is a triple[2] $l \Rrightarrow p\ (A)$ such that:

- l is a closed pattern of the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$;
- p is a meta-term;
- A is a set of meta-variable conditions.

The set of *dependency pairs* of a rewrite rule $l \Rightarrow r$, notation $\mathsf{DP}(l \Rightarrow r)$, consists of:

- all triples $l^\sharp \Rrightarrow p^\sharp\ (A)$ with $p\ (A) \in Cand(r)$, provided p is not a strict sub-meta-term of l,
- if l has a functional type $\sigma_1 \to \ldots \to \sigma_n \to \iota$ ($n \geq 1$) and $\mathtt{head}(r)$ has the form $F(\vec{s})$ with F a defined symbol or meta-variable: all triples $l \cdot Z_1 \cdots Z_k \Rrightarrow r \cdot Z_1 \cdots Z_k\ (\emptyset)$ where $k \in \{1, \ldots, n\}$ and all Z_i are fresh meta-variables

Let $\mathsf{DP}(\mathcal{R})$ (or just DP if $\mathcal{R}$ is clear from context) denote the set of all dependency pairs of rewrite rules of an AFSM $\mathcal{R}$.

[2]Unlike the name suggests, dependency pairs are triples, consisting of two meta-terms and a set of constraints. We will stick to the terminology "dependency pair" both because it corresponds to the first-order terminology, and because the most important part of this triple is the pair l, p.

The dependency pairs of $\mathcal{R}$ are also called *dynamic dependency pairs* of $\mathcal{R}$ to contrast the *static dependency pairs* which we will see in Chapter 7.8.

Example 6.21. The set of dependency pairs of `twice` consists of:

$$\begin{array}{lcl@{\qquad}lcl}
\mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X & \mathtt{twice}^\sharp(F) & \Rrightarrow & F \cdot (F \cdot y) \\
\mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}^\sharp(\lambda x.\mathtt{I}(x)) & \mathtt{twice}^\sharp(F) & \Rrightarrow & F \cdot y \\
\mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{I}^\sharp(x) & \mathtt{twice}(F) \cdot X & \Rrightarrow & F \cdot (F \cdot X) \\
 & & & \mathtt{twice}(F) \cdot X & \Rrightarrow & F \cdot X
\end{array}$$

Here, the set of meta-variable restrictions has been omitted where it is $\emptyset$ (that is, everywhere). We will generally stick to this notational convention. The last two dependency pairs originate from the rule added by β-saturation.

Example 6.22. The system `eval` from Example 6.14 does not need β-saturating, or dependency pairs of the second kind, because all rules already have base type. However, because the meta-variables take arguments, we do encounter variable restrictions. We have the following dependency pairs:

$$\begin{array}{rcl}
\mathtt{dom}^\sharp(\mathtt{s}(X), \mathtt{s}(Y), \mathtt{s}(Z)) & \Rrightarrow & \mathtt{dom}^\sharp(X, Y, Z) \\
\mathtt{dom}^\sharp(\mathtt{0}, \mathtt{s}(Y), \mathtt{s}(Y)) & \Rrightarrow & \mathtt{dom}^\sharp(\mathtt{0}, Y, Z) \\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x), X, Y), Z) & \Rrightarrow & F(\mathtt{dom}(X, Y, Z)) \\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x), X, Y), Z) & \Rrightarrow & \mathtt{dom}^\sharp(X, Y, Z) \ \ \{F : 1\}
\end{array}$$

The second form of dependency pair deals with functional rules whose right-hand side is not an abstraction. To illustrate why they are necessary, consider the system with function symbols $\mathtt{A} : [\mathtt{o}] \longrightarrow \mathtt{o} \to \mathtt{o}$ and $\mathtt{B} : [\mathtt{o} \to \mathtt{o}] \longrightarrow \mathtt{o}$, and one rewrite rule: $\mathtt{A}(\mathtt{B}(F)) \Rightarrow F$. This system has no dependency pairs of the first kind, but does admit a two-step loop: writing $\omega := \mathtt{B}(\lambda x.\mathtt{A}(x))$, we have $\mathtt{A}(\omega) \cdot \omega \Rightarrow (\lambda x.\mathtt{A}(x)) \cdot \omega \Rightarrow_\beta \mathtt{A}(\omega) \cdot \omega$. The rule *does* have a dependency pair of the second kind, $\mathtt{A}(\mathtt{B}(F)) \cdot x \Rrightarrow F \cdot x$.

Comparing the *dynamic* approach here to *static* dependency pairs as defined in [87], the main difference is that the dynamic approach includes *collapsing* dependency pairs, where the right-hand side is headed by a meta-variable application. Also, in the definition of a dependency chain, the free variables in the right-hand sides of dependency pairs are treated as harmless symbols; in the static approach, as we will see in Chapter 7.8, this is not the case.

Comment: The meta-variable conditions for dependency pairs are new in this work, and their use is not immediately evident. In this chapter, we will primarily use them to obtain a completeness result regarding dependency pairs. In Chapter 7.4, we will see how these conditions also affect the *dependency graph*, and thus add power for termination provers.

6.3.2 Dependency Chains

Now we have almost all the ingredients to define dependency chains. Compared to the first-order definition, we have several new aspects. To name a few: free variables (which originate from bound variables, and are therefore mostly harmless), meta-variable conditions $F : i$ and β-reduction.

Definition 6.23. A *dependency chain* is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i:

1. $\rho_i \in \mathsf{DP} \cup \{\texttt{beta}\}$;
2. if $\rho_i = l_i \Rrightarrow p_i\ (A) \in \mathsf{DP}$ then there exists a substitution γ such that:
   a) $dom(\gamma) = FMV(l_i) \cup FMV(p_i) \cup FV(p_i)$;
   b) $s_i = l_i\gamma$;
   c) if p_i is an application or functional term, then $t_i = p_i\gamma$;
   d) if $p_i = F(u_1, \ldots, u_n)$ with F a meta-variable, and $\gamma(F) = \lambda x_1 \ldots x_n.q$, then there exists a term v such that $q \trianglerighteq v$ and $\{\vec{x}\} \cap FV(v) \neq \emptyset$ and $t_i = v^\sharp[\vec{x} := \vec{u}\gamma]$, but v is not a variable;
   e) all variables in $dom(\gamma)$ are mapped to fresh variables;
   f) if $F : i \in A$ and $\gamma(F) = \lambda x_1 \ldots x_n.q$, then $x_i \in FV(q)$;
3. if $\rho_i = \texttt{beta}$ then $s_i = (\lambda x.q) \cdot u \cdot v_1 \cdots v_k$ and either
   a) $k > 0$ and $t_i = q[x := u] \cdot v_1 \cdots v_k$, or
   b) $k = 0$ and there exists a term v such that $q \trianglerighteq v$ and $x \in FV(v)$ and $t_i = v^\sharp[x := u]$, but v is not a variable;
4. $t_i \Rightarrow_{in}^* s_{i+1}$;

A dependency chain is *minimal* if moreover the strict subterms of all t_j are terminating in $\Rightarrow_{\mathcal{R}}$.

Here, a step $\Rightarrow_{in}$ is obtained by rewriting some q_i inside a term of the form $f(q_1, \ldots, q_n) \cdot q_{n+1} \cdots q_m$. The relation $\Rightarrow_{in}^*$ is reflexive, so if $t_i = s_{i+1}$, then t_i does not need to be (headed by) a functional term.

This definition has several aspects which are not necessary in the first-order case: the chain admits `beta` steps and reductions involving collapsing dependency pairs, both of which may be combined with an arbitrary subterm reduction. Even when using a normal dependency pair, γ has extra restrictions, involving the free variables and functional meta-variables. Also, we explicitly require that $t_i \Rightarrow_{in}^* s_{i+1}$, which is necessary because t_i may be an application rather than a functional term, and consequently may not be marked.

Comment: This definition allows subterm reduction steps in two different cases: following a β-reduction step, and after a collapsing dependency pair is used. This is a consequence of using the AFSM formalism, which gives us the freedom of both having β-reduction and meta-variables. In for instance a prior work where Femke van Raamsdonk and I defined dynamic dependency pairs for HRSs [78], there is only a subterm-step immediately after the use of a collapsing dependency pair, as HRSs have no separate β-steps. In the definition for AFSs [80] they are only needed following β-steps, because this formalism does not have meta-variables. The price of the generality of the AFSM framework, in this case, is that we need both.

Theorem 6.24. *$\mathcal{R}$ is non-terminating if and only if there is a minimal dependency chain over* $\mathsf{DP}(\mathcal{R})$.

Proof. Since the proof of this is quite long, it is split in two lemmas, Lemmas 6.25 and 6.27 below. □

Lemma 6.25. *If $\mathcal{R}$ is non-terminating then there is a minimal dependency chain over* $\mathsf{DP}(\mathcal{R})$.

Proof. Given any non-terminating term, let q_{-1} be a minimal-sized subterm that is still non-terminating (q_{-1} is *MNT*, or *Minimal Non-Terminating*). We make the observation:

(**) *If an MNT term is reduced at any other position than the top, then the result is either also MNT, or terminating.*

This holds because, if $q = C[s] \Rightarrow_{\mathcal{R}} C[t]$, because $s \Rightarrow_{\mathcal{R}} t$ and t is non-terminating, then so is s, which contradicts minimality of q unless C is the empty context. We also note:

(***) *If $u \Rightarrow_{in}^{*} v$, then $u^\sharp \Rightarrow_{in}^{*} v^\sharp$.*

This holds by the nature of an internal step.

Now, for any given $i \in \mathbb{N} \cup \{-1\}$, let q_i be some MNT term, and suppose $t_i = q_i^\sharp$. We consider an infinite reduction starting in q_i. The term q_i cannot be an abstraction, since abstractions can only be reduced by reducing their immediate subterm, contradicting minimality. For the same reason q_i cannot have the form $x \cdot u_1 \cdots u_n$ with x a variable, or the form $f(u_1, \ldots, u_n) \cdot u_{n+1} \cdots u_m$ with f a constructor symbol. What remains are the forms:

(A) $q_i = (\lambda x.q) \cdot u \cdot v_1 \cdots v_n$;

(B) $q_i = f(u_1, \ldots, u_n) \cdot u_{n+1} \cdots u_m$ with $f \in \mathcal{D}$.

In the first case we will choose $\rho_{i+1} = \texttt{beta}$ and satisfy requirement 3. In the second case we will choose $\rho_{i+1} \in \mathsf{DP}$ and satisfy requirement 2. Either way, requirement 1 and the minimality constraint are satisfied (the latter because always $t_i = q_i^\sharp$ with q_i MNT), and we have to take care that requirement 4 is satisfied.

(A) Let $q_i = (\lambda x.u) \cdot v \cdot w_1 \cdots w_n$. By minimality of q_i eventually a headmost step must be taken, which must be a β-step because the left-hand sides of rules have the form $f(\vec{l_1}) \cdot \vec{l_2}$. Therefore, any infinite reduction starting in q_i has the form $q_i \Rightarrow^*_{\mathcal{R}} (\lambda x.u') \cdot v' \cdot w'_1 \cdots w'_n \Rightarrow_\beta u'[x := v'] \cdot w'_1 \cdots w'_n \Rightarrow_{\mathcal{R}} \ldots$ Since also $u[x := v] \cdot w_1 \cdots w_n \Rightarrow^*_{\mathcal{R}} u'[x := v'] \cdot w'_1 \cdots w'_n$ the immediate beta-reduct of q_i is non-terminating as well. There are two sub-cases:

- If $n > 0$, this reduct is MNT by (**); in this case choose $q_{i+1} := u[x := v] \cdot w_1 \cdots w_n$ and let $\rho_{i+1}, s_{i+1}, t_{i+1} := \texttt{beta}, q_i, q_{i+1}$. Note that $s_i^\sharp = s_i$ and $t_i^\sharp = t_i$, and that case 3a of the definition of a dependency chain is satisfied.
- If $n = 0$, so $u[x := v]$ is non-terminating, then let w be a minimal-sized subterm of u where $w[x := v]$ is still non-terminating. By minimality of q_i both w and v are terminating, so $FV(w)$ contains x, but not $w = x$. Since w is not a variable we have $(w[x := v])^\sharp = w^\sharp[x := v]$. By minimality of w, also $w[x := v]$ is MNT (as its direct subterms have the form $w'[x := v]$ for some subterm w' of w). Thus, choosing $q_{i+1} := w[x := v]$ and $\rho_{i+1}, s_{i+1}, t_{i+1} := \texttt{beta}, q_i, q_{i+1}^\sharp$, case 3b is satisfied.

Note that in both sub-cases, case 4 is also satisfied, since $t_i = s_{i+1}$.

(B) Let $q_i = f(u_1, \ldots, u_n) \cdot u_{n+1} \cdots u_m$. Any infinite reduction starting in q_i must eventually take a headmost step. Therefore we can find some rule $l \Rightarrow r$ and term $q'_i = l\gamma \cdot u'_{j+1} \cdots u'_m$ such that $q_i \Rightarrow^*_{in} q'_i$, and $r\gamma \cdot u'_{j+1} \cdots u'_m$ is still non-terminating. Choose $s_{i+1} := q_i'^\sharp$; by (***), requirement 6.23(4) is satisfied, and by (**) q'_i is MNT.

Since the rules were β-saturated, we can assume that either $m = j$, or r is not an abstraction or application headed by a β-redex: if $r = \lambda x.r'$ and $m > j$ then the resulting term $r\gamma \cdot \vec{u'}$ is a β-redex, and (like above) may be reduced immediately without losing termination. The same result would have been obtained with the rule $l \cdot Z \Rightarrow r'[x := Z]$. If r is headed by a β-redex $(\lambda x.s) \cdot t \cdot \vec{q}$ while $m > j$, then l has a functional type, so there is also a rule $s[x := t] \cdot \vec{q}$ which could have been used instead to obtain a non-terminating term.

If $m > j$, then by (**) $r\gamma \cdot u'_{j+1} \cdots u'_m$ is MNT. Consequently, $\texttt{head}(r\gamma)$ cannot be a variable or a functional term $g(\vec{v})$ with g a constructor symbol: $r\gamma$ is either headed by an abstraction, or by a functional term with root symbol in $\mathcal{D}$. Since r itself is not an abstraction, nor headed by a β-redex, its head must be a meta-variable application or a functional term with defined root symbol. Either way, $l \cdot Z_{j+1} \cdots Z_m \Rrightarrow r \cdot Z_{j+1} \cdots Z_m$ $(\emptyset)$ is a dependency pair. Let ρ_{i+1} be this dependency pair, and let $\gamma' := \gamma \cup [Z_{j+1} := u'_{j+1}, \ldots, Z_m := u'_m]$. Additionally let $q_{i+1} := r\gamma \cdot u'_{j+1} \cdots u'_m$ (an MNT term as required), and $t_{i+1} := q_{i+1}$ (which equals $q_{i+1}^\sharp$ because q_{i+1} is an application). Requirement 2 is satisfied with the substitution γ': this is clear for 2a, 2b and 2c (recall that $s_{i+1} = q_i'^\sharp$); 2d is not applicable because the right-hand side of the chosen dependency pair is an appli-

cation, 2e because the right-hand side has no free variables, and 2f because ρ_{i+1} has an empty set of meta-variable requirements.

On the other hand, if $m = j$, then $q'_i = l\gamma$ and $r\gamma$ is non-terminating. Let all bound variables in r be fresh (a safe assumption by α-conversion). We choose a pair $\varphi_\emptyset(r) = p\ (A)$ in $\mathsf{brsmt}_\emptyset(r)$ as follows:

- if r is an abstraction $\lambda x.r'$, then $r'\gamma$ is also non-terminating; let $\varphi_A(r) = \varphi_A(r')$;
- if r is an application $(\lambda x.s) \cdot t \cdot \vec{q}$, and $t\gamma$ is non-terminating, then choose $\varphi_A(r) := \varphi_A(t)$, otherwise choose $\varphi_A(r) = \varphi_A(s[x := t] \cdot \vec{q})$. Note that in the latter case $(s[x := t] \cdot \vec{q})\gamma$ is non-terminating too: either some $q_i\gamma$ is non-terminating (a subterm of this term), or s is non-terminating (and therefore so is $s[x := t]$), or otherwise the β-reduct is;
- if either $r = s_1 \cdot s_2$ with $\mathtt{head}(s_1)$ not an abstraction, or $r = f(\vec{s})$, and all $s_i\gamma$ are terminating, then choose $\varphi_A(r) = r\ (A)$; if some $s_i\gamma$ is non-terminating, then choose the smallest such i and let $\varphi_A(r) = \varphi_A(s_i)$;
- if $r = F(s_1, \ldots, s_n)$ and $\gamma(F) = \lambda x_1 \ldots x_n.t$, and if $s_i\gamma$ are terminating for all i such that $x_i \in FV(t)$, then $\varphi_A(r) = r\ (A)$; if $s_i\gamma$ is non-terminating for some i with $x_i \in FV(t)$, then choose the smallest such i and let $\varphi_A(r) = \varphi_{A \cup \{F:i\}}(s_i)$.

Write $\varphi_\emptyset(r) = p\ (A)$. From the choice of $\varphi_\emptyset(r)$ it is obvious that $\varphi_\emptyset(r) \in \mathsf{brsmt}_\emptyset(r)$ and that $p\gamma$ is non-terminating. Moreover, for $F : i \in A$ we have that $\gamma(F) = \lambda\vec{x}.u$ with $x_i \in FV(u)$.

If p does not have the form $Z(\vec{v})$ for some meta-variable Z (so p is an application or functional term), note that $p'\gamma$ is terminating for all immediate subterms p' of p. Thus, $p\gamma$ is MNT. As observed before, this can only be the case if $p\gamma$ is headed by an abstraction or functional term with a defined root symbol. Taking into account that p itself is not headed by an abstraction, there is a candidate term $\tilde{p}\ (A)$ of r such that p is just $\tilde{p}$ with a variable renaming applied on it. Say $p = \tilde{p}\chi$ for some substitution χ which maps the variables in $\tilde{p}$ to the free variables occurring in p. Then $\tilde{p}$ is not a strict sub-meta-term of l: either $\tilde{p}$ contains free variables which do not occur in l, or $\tilde{p} = p$, and $p\gamma$ is not a strict subterm of $l\gamma$ because the latter is MNT. Choose $\rho_{i+1} := l^\sharp \Rrightarrow \tilde{p}^\sharp$ and let $q_{i+1} := \tilde{p}\chi\gamma$, and $t_{i+1} := q^\sharp_{i+1} = \tilde{p}^\sharp\chi\gamma$ (since p is not a variable). All parts of requirement 2 are satisfied for the substitution $\chi \cup \gamma$: 2a, 2b and 2c are evident, 2d is not applicable, 2e holds by the choice of χ (note that bound variables in r, so free variables in p, were assumed to be fresh), and we already observed that property 2f is satisfied.

Finally, if p does have the form $Z(\vec{v})$ for some meta-variable Z, the v_j cannot be all distinct bound variables: if they were, $p\gamma$ was just a strict subterm of s_i with variables renamed, and therefore terminating. Thus, some renaming of $p\ (A)$ must be a candidate term. Let $p = \tilde{p}\chi$ with $\tilde{p}\ (A) \in \mathit{Cand}(r)$ and χ a suitable variable renaming, so $\rho_{i+1} := l^\sharp \Rrightarrow \tilde{p}^\sharp$ is a dependency pair. Suppose

$\gamma(Z) = \lambda\vec{x}.u$. Clearly $u[\vec{x} := \vec{v}\gamma] = p\gamma$ is non-terminating. Consider a minimal subterm w of u such that $w[\vec{x} := \vec{v}\gamma]$ is still non-terminating. Then w cannot be one of the x_i, for if x_i occurs in q, then its substitute, $v_i\gamma$, is terminating by choice of p. Nor can it be another variable, for then $w[\vec{x} := \vec{v}\gamma] = w$ is clearly terminating. We also see that $FV(w)$ must contain some of the x_i, for otherwise $w[\vec{x} := \vec{v}\gamma] = w$, which is a subterm of $\gamma(Z)$ and therefore a strict subterm of $l\gamma = q'_i$, which contradicts minimality of q'_i.

Choose $q_{i+1} := w[\vec{x} := \vec{v}\gamma]$ and $t_{i+1} := q^\sharp_{i+1}$ and $\rho_i := l^\sharp \Rrightarrow \tilde{p}\ (A)$. For the substitution γ, all requirements of 2 are satisfied: requirement 2a is obvious, requirement 2b we have already seen, requirement 2c is not applicable, the reasoning above shows that 2d is satisfied (taking into account that $(w\delta)^\sharp = w^\sharp\delta$ if w is not a variable), 2e holds by the freshness assumption on the free variables in r, and we had already observed that property 2f is satisfied. □

Example 6.26. As we will see, neither `twice` nor `eval` admits a dependency chain. As an example of a system which does have one, consider the AFSM with the following three rules:

$$\mathtt{f}(\mathtt{0}) \Rightarrow \mathtt{g}(\lambda x.\mathtt{f}(x), \mathtt{a}) \qquad \mathtt{g}(F, \mathtt{b}) \Rightarrow F \cdot \mathtt{0} \qquad \mathtt{a} \Rightarrow \mathtt{b}$$

This system has four dependency pairs:

$$\begin{array}{rcl} \mathtt{f}^\sharp(\mathtt{0}) & \Rrightarrow & \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) \\ \mathtt{f}^\sharp(\mathtt{0}) & \Rrightarrow & \mathtt{f}^\sharp(x) \\ \mathtt{f}^\sharp(\mathtt{0}) & \Rrightarrow & \mathtt{a}^\sharp \\ \mathtt{g}^\sharp(F, \mathtt{b}) & \Rrightarrow & F \cdot \mathtt{0} \end{array}$$

The rules admit an infinite reduction: $\mathtt{f}(\mathtt{0}) \Rightarrow \mathtt{g}(\lambda x.\mathtt{f}(x), \mathtt{a}) \Rightarrow \mathtt{g}(\lambda x.\mathtt{f}(x), \mathtt{b}) \Rightarrow (\lambda x.\mathtt{f}(x)) \cdot \mathtt{0} \Rightarrow_\beta \mathtt{f}(\mathtt{0}) \Rightarrow \ldots$ Following the steps in the proof of Theorem 6.24 (starting with $\mathtt{f}(\mathtt{0})$) we obtain the following dependency chain:

$$\begin{array}{llll} (& \mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) & , \ \mathtt{f}^\sharp(\mathtt{0}) & , \ \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) \ ), \\ (& \mathtt{g}^\sharp(F, \mathtt{b}) \Rrightarrow F \cdot \mathtt{0} & , \ \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{b}) & , \ (\lambda x.\mathtt{f}(x), \mathtt{a}) \cdot \mathtt{0} \ ), \\ (& \mathtt{beta} & , \ (\lambda x.\mathtt{f}(x), \mathtt{a}) \cdot \mathtt{0} & , \ \mathtt{f}^\sharp(\mathtt{0}) \ ), \\ (& \mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) & , \ \mathtt{f}^\sharp(\mathtt{0}) & , \ \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) \ ), \\ & \ldots & & \end{array}$$

Between the first and second step, a $\Rightarrow_{in}$ step was done to reduce $\mathtt{a}$ to $\mathtt{b}$. Also note that in the third triple we used case 3b from Definition 6.23, with $v = \mathtt{f}(x)$.

If we alter the specification a little, replacing the rule $\mathtt{g}(F, \mathtt{b}) \Rightarrow F \cdot \mathtt{0}$ by $\mathtt{g}(\lambda x.F(x), \mathtt{b}) \Rightarrow F(\mathtt{0})$, then we get the following dependency chain:

$$\begin{array}{llll} (& \mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) & , \ \mathtt{f}^\sharp(\mathtt{0}) & , \ \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) \ ), \\ (& \mathtt{g}^\sharp(\lambda x.F(x), \mathtt{b}) \Rrightarrow F(\mathtt{0}) & , \ \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{b}) & , \ \mathtt{f}^\sharp(\mathtt{0}) \ ), \\ (& \mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) & , \ \mathtt{f}^\sharp(\mathtt{0}) & , \ \mathtt{g}^\sharp(\lambda x.\mathtt{f}(x), \mathtt{a}) \ ), \\ & \ldots & & \end{array}$$

Here, in the second step, case 2d is used.

As in the first-order case, the other direction of Theorem 6.24 holds, intuitively, because a dependency chain roughly defines a $\Rightarrow_{\mathcal{R}} \cup \rhd$ reduction.

Lemma 6.27. *If there is a dependency chain over* $\mathsf{DP}(\mathcal{R})$*, then* $\mathcal{R}$ *is non-terminating.*

Proof. First, consider the definition of brsmt_B: we will use that for any $t\ (A) \in \mathsf{brsmt}_\emptyset(s)$ and substitution γ which respects A: $s\gamma\ (\Rightarrow_\beta \cup \rhd)^*\ t\gamma$. Here, the phrase "$\gamma$ respects A" means that for all $F : i \in A$ the substitute $\gamma(F) = \lambda x_1 \ldots x_n.q$ has the property that $x_i \in FV(q)$.

(**) If $t\ (A) \in \mathsf{brsmt}_B(s)$ *and* γ *is a substitution with only meta-variables in its domain, which respects* A*, then* $s\gamma\ (\Rightarrow_\beta \cup \rhd)^*\ t\gamma$*.*

The proof of this claim is a straightforward induction on the derivation of $t\ (A) \in \mathsf{brsmt}_B(s)$: we are done if $s = t$, take a subterm step if s is an abstraction, application or functional term (the variable which becomes free in the abstraction case is not a problem, because $dom(\gamma)$ contains only meta-variables), and if $s = (\lambda x.u) \cdot v \cdot \vec{w}$ and $t\ (A) \in \mathsf{brsmt}(v)$ we take a subterm step, otherwise a β-step. Only the case where $s = F(s_1, \ldots, s_n)$ and $t\ (A) \in \mathsf{brsmt}_{B\cup\{F:i\}}(s_i)$ is not entirely trivial. In this case, note that whenever $t\ (A) \in \mathsf{brsmt}_B(q)$, we necessarily have $B \subseteq A$ (this is an easy induction on the definition of brsmt_B). Thus, $F : i \in A$, so $\gamma(F) = \lambda x_1 \ldots x_n.q$ with $x_i \in FV(q)$, so $s\gamma = q[x_1 := s_1\gamma, \ldots, x_n := s_n\gamma] \trianglerighteq s_i\gamma$, which by the induction hypothesis $(\Rightarrow_\beta \cup \rhd)^+\ t\gamma$. Thus, (**) holds.

Now consider a dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$. We will see that each $|s_i|\ (\Rightarrow_{\mathcal{R}} \cup \rhd)^+\ |s_{i+1}|$, where $|q|$ is q with possible $^\sharp$ marks removed. Since the existence of such an infinite reduction implies the existence of an infinite $\Rightarrow_{\mathcal{R}}$ reduction (as $a \Rightarrow_{\mathcal{R}} \cdot \rhd b$ implies $a \Rightarrow_{\mathcal{R}} C[b]$ for some context C), this is sufficient to derive non-termination of $\Rightarrow_{\mathcal{R}}$. In the subterm steps, we are not bound by any particular representation of $|s_i|$: using α-conversation we both have e.g. $\lambda x.x \rhd x$ and $\lambda x.x \rhd y$ for a fresh variable y.

For given i, consider ρ_i.

If $\rho_i = \mathtt{beta}$, then whether case 3a or case 3b applies, $|s_i| \Rightarrow_\beta \cdot \trianglerighteq |t_i| \Rightarrow_{\mathcal{R}}^* |s_{i+1}|$.

If ρ_i is a dependency pair $l^\sharp \Rrightarrow p^\sharp\ (A)$, then there are a rule $l \Rightarrow r$ and substitution γ such that $p\ (A) \in \mathsf{brsmt}_\emptyset(r)$, and γ respects A (requirement 2f of Definition 6.23). Write $\gamma = \delta \cup \chi$, where $dom(\delta)$ consists of meta-variables and χ maps the variables in $FV(p) \setminus FV(r)$ to fresh variables. We can also write $\gamma = \chi\delta$. Using α-conversion, we can replace every bound variable x in r systematically by $\gamma(x)$ (if defined), without changing the term. Write r' for this different representation of r. Then we easily see that $p\chi\ (A) \in \mathsf{brsmt}_\emptyset(r')$. By (**), $l\gamma = l\delta \Rightarrow_{\mathcal{R}} r\delta = r'\delta (\Rightarrow_{\mathcal{R}} \cup \rhd)^*\ p\chi\delta = p\gamma$.

If we are in case 2c of the definition of a dependency chain, then $p\gamma = |t_i|$, otherwise $p\gamma \trianglerighteq |t_i|$. Either way, $|s_i|\ (\Rightarrow_{\mathcal{R}} \cup \rhd)^+ |t_i|$, which $\Rightarrow_{\mathcal{R}}^* |s_{i+1}|$. □

Similar to the rule removal setting, we will consider sets of dependency pairs, and iteratively eliminate pairs. To this end, consider the following definition:

Definition 6.28. A set of dependency pairs $\mathcal{P}$ is *chain-free* if there is no minimal dependency chain where all $\rho_i \in \mathcal{P} \cup \{\texttt{beta}\}$.

We have seen that an AFSM is terminating if and only if $\mathsf{DP}(\mathcal{R})$ is chain-free.

Considering the form dependency chains might have, we could make a number of observations:

I Lemmas 6.25 and 6.27 together also prove that an AFSM is terminating if and only if it does not admit a dependency chain, regardless of minimality.

II $\emptyset$ is chain-free, because a dependency chain with only `beta` steps is an infinite $\Rightarrow_\beta \cup \rhd$ reduction, which cannot exist by the well-foundedness of β-reduction.

III If a set of dependency pairs $\mathcal{P}$ is non-collapsing, then there is a dependency chain over $\mathcal{P}$ where all $\rho_i \in \mathcal{P}$ (so no `beta` steps are taken), and which does not take subterm steps. This is clear by considering that eventually (by Observation II) some $\rho_i \in \mathcal{P}$. But then, the head of t_i can only be a functional term, so ρ_{i+1} must also be in $\mathcal{P}$, and no subterm step is possible.

The variables which are freed in the right-hand side of a dependency pair are essentially harmless: they may only be substituted with other variables. This will occasionally be reflected by substituting them with "small" symbols `c`. To this end, let C be a set consisting of fresh symbols $\mathtt{c}_i^\sigma$ for all types σ and numbers $i \in \mathbb{N}$. Let $\mathcal{F}_c$ denote the set $\mathcal{F} \uplus \mathsf{C}$, and $\mathcal{F}_c^\sharp := \mathcal{F}^\sharp \uplus \mathsf{C}$.

6.3.3 Reduction Triples

In the first-order method of Section 6.2.3 we used a reduction pair to reduce the question "is $\mathcal{P}$ chain-free" to the same question for a smaller set $\mathcal{P}'$. In the higher-order case, because of type differences, it will be advantageous to instead use a *reduction triple*. This definition is lifted from a first-order definition in [55].

Definition 6.29. A *reduction triple* consists of two quasi-orderings $\succsim$ and $\succeq$ and a well-founded ordering $\succ$, all defined on meta-terms built over $\mathcal{F}_c^\sharp$, such that:

1. $\succsim$ and $\succ$ are compatible: $\succsim \cdot \succ \subseteq \succ$; also, $\succeq$ and $\succ$ are compatible;
2. $\succsim$, $\succeq$ and $\succ$ are all *meta-stable* (if $l\ R\ r$ and l is a pattern of the form $f(\vec{s}) \cdot \vec{t}$, and γ is a substitution on domain $FMV(l) \cup FMV(r)$, then $l\gamma\ R\ r\gamma$);
3. $\succsim$ is monotonic (if $s \succsim t$ and s, t are terms of the same type, then $C[s] \succsim C[t]$ for all $C[]$);
4. $\succsim$ contains beta (always $(\lambda x.s) \cdot t \succsim s[x := t]$ if s, t are terms).

The *reduction pair* (or *weak reduction pair*) from Definition 2.24 is a pair $(\succsim, \succ)$ such that $(\succsim, \succsim, \succ)$ is a reduction triple. The reduction triple generalises this definition, by not requiring $\succeq$ to be monotonic. We will need a non-monotonic $\succeq$ in Section 6.3.4 to compare terms with different types.

To deal with subterm reduction in dependency chains, an additional definition is needed.

Definition 6.30 (Subterm Property)**.** $\succeq$ has the *subterm property* if the following requirement is satisfied: *for all terms s, t over $\mathcal{F}$ such that $s \trianglerighteq t$, there is a substitution γ with domain $FV(t) \setminus FV(s)$ such that $s \succeq t^\sharp\gamma$.*

Intuitively, the substitution γ can be used to replace free variables in t which are bound in s by corresponding constants c_i. However, we will also use a more liberal replacement of those variables, hence the general γ. Note that the subterm property concerns *all* terms s, t with $s \trianglerighteq t$, not just base-type terms.

The following theorem shows how reduction triples are used with dependency pairs.

Theorem 6.31. *A set $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ of dependency pairs is chain-free if $\mathcal{P}_2$ is chain-free, and there is a reduction triple $(\succsim, \succeq, \succ)$ such that:*

- *$l \succ p'$ for all $l \Rrightarrow p\,(A) \in \mathcal{P}_1$,*
- *$l \succeq p'$ for all $l \Rrightarrow p\,(A) \in \mathcal{P}_2$,*
- *$l \succsim r$ for all $l \Rightarrow r \in \mathcal{R}$,*
- *$\mathcal{P}$ is non-collapsing or $\succeq$ satisfies the subterm property.*

Here, p' is p with free variables replaced by arbitrary c_i^σ of suitable type.

A set of dependency pairs $\mathcal{P}$ is called *collapsing* if it contains any collapsing dependency pairs, or *non-collapsing* if it does not.

Proof. Towards a contradiction, suppose that $\mathcal{P}_2$ is chain-free, but $\mathcal{P}$ admits a minimal dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N} \mid i \geq j]$. Then infinitely many ρ_i must be in $\mathcal{P}_1$. Moreover, we can safely assume that the variables which occur in the dependency pairs do not occur in the s_i, t_i. If $\mathcal{P}$ is non-collapsing, then by observation III we can also assume that all ρ_i are dependency pairs, and that case 2d is never applicable.

Let δ_0 be the empty substitution, and $q_0 := s_0$. Now, for each i, suppose we have a term q_i and a substitution δ_i such that $q_i = s_i\delta_i$. Suppose also that the domain of δ_i contains only variables. Consider ρ_i.

- If ρ_i is a dependency pair $l \Rrightarrow p\,(A)$ and p is not a meta-variable application, then $s_i = l\gamma, t_i = p\gamma$ for some substitution γ. Let χ be a substitution such that we have $l \succ p\chi$ if $\rho_i \in \mathcal{P}_1$ or $l \succeq p\chi$ if $\rho_i \in \mathcal{P}_2$ (so χ maps variables to symbols c_i).

By meta-stability of $\succ$ or $\succeq$ either $s_i\delta_i \succ p\chi\gamma\delta_i$ (if $\rho_i \in \mathcal{P}_1$) or $s_i\delta_i \succeq p\chi\gamma\delta_i$ (if $\rho_i \in \mathcal{P}_2$).

Since the variables in the dependency pairs do not occur in t_i, and the $\mathtt{c}_i^\sigma$ symbols cannot be substituted, $p\chi\gamma = p\gamma\chi = t_i\chi$.

Let $\delta_{i+1} := \chi\delta_i$. Then $s_i\delta_i \succ t_i\delta_{i+1}$ or $s_i\delta_i \succeq t_i\delta_{i+1}$.

- If ρ_i is a dependency pair $l \Rrightarrow p\,(A)$ and $p = F(p_1, \ldots, p_m)$, then $s_i = l\gamma$ and $p\gamma \trianglerighteq |t_i|$ for some substitution γ. Here, $|t_i|$ means t_i with $\sharp$-tags removed. In this case, $\mathcal{P}$ is collapsing, so we can use the subterm property.

  As in the previous case, we can find a substitution χ mapping variables to $\mathtt{c}$-symbols such that either $s_i\gamma_i \succ (p\gamma)\chi\gamma_i$ (if $\rho_i \in \mathcal{P}_1$), or $s_i\gamma_i \succeq (p\gamma)\chi\gamma_i$ (if $\rho_i \in \mathcal{P}_2$).

  By the definition of a dependency chain, $p\gamma \trianglerighteq q$ for some term q with $q^\sharp = t_i$. Since the domains of χ and ζ_i contain only variables, and since q itself is not a variable, we also have that $p\gamma\chi\delta_i \trianglerighteq q\chi\delta_i$. The subterm property provides a substitution ζ such that $p\gamma\chi\delta_i \succeq t_i\chi\delta_i\zeta$. The domain of ζ contains only variables.

  Let $\delta_{i+1} := \chi\delta_i\zeta$. Then $s_i\delta_i \succ t_i\delta_{i+1}$ or $s_i\delta_i \succeq t_i\delta_{i+1}$ by compatibility or transitivity.

- if $\rho_i = \mathtt{beta}$, then $\mathcal{P}$ is collapsing, so we can use the subterm property. Either $s_i\delta_i \succsim t_i\delta_i$ because $\succsim$ contains beta, or there is a substitution ζ with $s_i\delta_i \succsim s_i'\delta_i \succeq t_i\delta_i\zeta$ (where s_i' is the β-reduct of s_i).

  In the first case, let $\delta_{i+1} := \delta_i$; then $s_i\delta_i \succsim t_i\delta_{i+1}$.; In the second case, let $\delta_{i+1} := \delta_i\zeta$; then $s_i\delta_i \succsim \cdot \succeq t_i\delta_{i+1}$.

In all three cases, we find a substitution δ_{i+1}, whose domain contains only variables, such that either $s_i\delta_i \succ t_i\delta_{i+1}$, or $s_i\delta_i \succeq t_i\delta_{i+1}$, or $s_i\delta_i \succsim \cdot \succeq t_i\delta_{i+1}$. Moreover, the strict case $s_i\delta_i \succ t_i\delta_{i+1}$ holds for infinitely many i.

In addition, we find that $t_i\delta_{i+1} \succsim s_{i+1}\delta_{i+1}$, because $\succsim$ is meta-stable, monotonic, transitive, and contains beta. We use for instance that if $C[l\gamma] \Rightarrow_{\mathcal{R}} C[r\gamma]$, then $l\gamma\delta_{i+1} \succsim r\gamma\delta_{i+1}$ by meta-stability, so $C[l\gamma]\delta_{i+1} = C\delta_{i+1}[l\gamma\delta_{i+1}] \succsim C\delta_{i+1}[r\gamma\delta_{i+1}] = C[r\gamma]\delta_{i+1}$.

Thus, using compatibility and transitivity, for all i we see that either $s_i\delta_i \succ s_{i+1}\delta_{i+1}$, or $s_i\delta_i \succeq \cdot \succsim s_{i+1}\delta_{i+1}$, or $s_i\delta_i \succsim \cdot \succeq \cdot \succsim s_{i+1}\delta_{i+1}$. This leads to an infinite reduction of $\succ$, $\succeq$ and $\succsim$ steps, with infinitely many occurrences of $\succ$. Using both compatibility clauses, this can be transformed into an infinite decreasing $\succ$-sequence, contradicting well-foundedness. □

Example 6.32. Termination of `twice` is proved if there is a reduction triple $(\succsim, \succeq, \succ)$ which satisfies the subterm property, and also:

$$\begin{array}{rcl}
\mathtt{I}^\sharp(\mathtt{s}(X)) & {\scriptstyle (}\succeq{\scriptstyle)} & \mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X \\
\mathtt{I}^\sharp(\mathtt{s}(X)) & {\scriptstyle (}\succeq{\scriptstyle)} & \mathtt{twice}^\sharp(\lambda x.\mathtt{I}(x)) \\
\mathtt{I}^\sharp(\mathtt{s}(X)) & {\scriptstyle (}\succ{\scriptstyle)} & \mathtt{I}^\sharp(\mathtt{c}_1^{\mathtt{nat}}) \\
\mathtt{twice}^\sharp(F) & {\scriptstyle (}\succ{\scriptstyle)} & F \cdot (F \cdot \mathtt{c}_2^{\mathtt{nat}}) \\
\mathtt{twice}^\sharp(F) & {\scriptstyle (}\succ{\scriptstyle)} & F \cdot \mathtt{c}_3^{\mathtt{nat}} \\
\mathtt{twice}(F) \cdot X & {\scriptstyle (}\succ{\scriptstyle)} & F \cdot (F \cdot X) \\
\mathtt{twice}(F) \cdot X & {\scriptstyle (}\succ{\scriptstyle)} & F \cdot X \\
\mathtt{I}(\mathtt{0}) & \succsim & \mathtt{0} \\
\mathtt{I}(\mathtt{s}(X)) & \succsim & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\
\mathtt{twice}(F) & \succsim & \lambda y.F \cdot (F \cdot y) \\
\mathtt{twice}(F) \cdot X & \succsim & F \cdot (F \cdot X)
\end{array}$$

Each of the ${\scriptstyle (}\succeq{\scriptstyle)}$ constraints should be oriented with $\succ$ or with $\succsim$, and the set of dependency pairs which were oriented with $\succsim$ must be chain-free.

At this point, we have not used certain special features of dependency chains. For example, the subterm property completely ignores that subterm reduction is only ever necessary in combination with substitution. This will be used in Section 6.5.

Another feature we have lost is completeness: there is no parallel for Lemma 6.5 which states that a set of dependency pairs is chain-free *if and only if* it has a reduction pair. This is due both to non-left-linear systems, and the fact that we ignore the meta-variable constraints which dependency pairs are equipped with. The *dependency graph*, which will be extended to the higher-order case in Chapter 7.4, can make use of differences in bound variables and the restrictions on dependency pairs. In Example 7.34 we will see a set $\mathcal{P}$ that cannot be proved chain-free with just a reduction pair.

For AFSs, where meta-variables do not take arguments, we *can* have a completeness result involving reduction triples (provided we restrict attention to left-linear systems), as demonstrated in [80]. However, taking into account the transformations of the next section, which are also likely to break completeness, the usefulness of such a result only goes so far.

6.3.4 Type Changing

The situation so far is not completely satisfactory, because both $\succeq$ and $\succ$ may have to compare terms of very different types, which neither polynomial interpretations nor path orderings are equipped to do very well. Consider for example the dependency pair $\mathtt{twice}^\sharp(F) \Rrightarrow F \cdot x$, where the left-hand side has a functional type and the right-hand side does not. Moreover, the comparison in the definition of the subterm property may concern terms of arbitrary different types.

A solution is to manipulate the ordering constraints. Let $(\succsim, \succ)$ be a weak reduction pair. Define $\gtrsim$, $\geq$ and $>$ as relations on *terms* as follows:

- $s > t$ if for all terms $q_1, \ldots, q_n$ such that $s \cdot \vec{q}$ has base type, there are terms $u_1, \ldots, u_m$ such that $t \cdot \vec{u}$ has base type, and moreover $s \cdot \vec{q} \succ t \cdot \vec{u}$;
- $s \geq t$ if for all terms $q_1, \ldots, q_n$ such that $s \cdot \vec{q}$ has base type, there are terms $u_1, \ldots, u_m$ such that $t \cdot \vec{u}$ has base type, and moreover $s \cdot \vec{q}\ R\ t \cdot \vec{u}$, where R is the union of $\succsim$ and $\succsim \cdot \succ$;
- $s \gtrsim t$ if $s \succsim t$ and s, t have the same type.

For each $R \in \{\gtrsim, \geq, >\}$, let R' be the relation on meta-terms given by: $s\ R'\ t$ if $s\gamma\ R\ t\gamma$ for all substitutions γ on domain $FMV(s) \cup FMV(t)$.

Lemma 6.33. $(\gtrsim', \geq', >')$ *as generated from a reduction pair* $(\succsim, \succ)$ *is a reduction triple.*

Proof. First note that $\gtrsim', \geq', >'$ coincide with $\gtrsim, \geq, >$ when restricted to terms. It is easy to see that each of the properties of transitivity, well-foundedness and reflexivity holds for R' if it holds for R, and similar for compatibility.

$>'$ **is transitive**: if $s > t > q$, then for all terms $\vec{u}$ such that $s \cdot \vec{u}$ has base type, there are terms $\vec{v}$ such that $t \cdot \vec{v}$ has base type, and $s \cdot \vec{u} \succ t \cdot \vec{v}$. But then there are also terms $\vec{w}$ such that $q \cdot \vec{w}$ has base type, and $t \cdot \vec{v} \succ q \cdot \vec{w}$. By transitivity of $\succ$ we see: $s \cdot \vec{u} \succ q \cdot \vec{w}$.

$>'$ **is well-founded**: if $s_0 > s_1 > \ldots$ then for all numbers i and terms $\vec{q_i}$ such that $s_i \cdot \vec{q_i}$ has base type, there are terms $\vec{q}_{i+1}$ such that $s_{i+1} \cdot \vec{q}_{i+1}$ has base type and $s_i \cdot \vec{q_i} \succ s_{i+1} \cdot \vec{q}_{i+1}$. Thus, choosing $\vec{q_0}$ a sequence of variables, we have $s_0 \cdot \vec{q_0} \succ s_1 \cdot \vec{q_1} \succ \ldots$, contradicting well-foundedness of $\succ$.

$\geq'$ **is transitive**: this follows in the same way as transitivity of $>'$, provided the union of $\succsim$ and $\succsim \cdot \succ$ is transitive. But this is easy to see with compatibility (all cases are straightforward).

$\gtrsim'$ **is transitive**: this follows from transitivity of $\succsim$.

$\geq'$ **and** $\gtrsim'$ **are reflexive**: by reflexivity of $\succsim$.

$\geq'$ **and** $>'$ **are compatible**: If $s > t \geq q$ then for all $\vec{u}$ there are $\vec{v}, \vec{w}$ such that $s \cdot \vec{u} \succ t \cdot \vec{v}\ R\ q \cdot \vec{w}$, where R is either $\succ$ or $\succsim \cdot \succ$. In the first case, $s \cdot \vec{u} \succ q \cdot \vec{w}$ by transitivity of $\succ$, in the second case this holds because $\succ \cdot \succsim \cdot \succ$ is included in $\succ$ by compatibility and transitivity.

$\gtrsim'$ **and** $>'$ **are compatible**: $\gtrsim'$ is a subrelation of $\geq'$, because $\gtrsim$ is a subrelation of $\geq$, by monotonicity of $\succsim$ (if $s \succsim t$, then for all $\vec{q}$ also $s \cdot \vec{q} \succsim t \cdot \vec{q}$). Compatibility with $>'$ follows.

$\geq', \gtrsim'$ **and** $>'$ **are all meta-stable**: let s, t be meta-terms, and suppose that $s >' t$. That is: for all substitutions γ on domain $FMV(s) \cup FMV(t)$ and terms $\vec{q}$, we have that $s\gamma > t\gamma$. Let δ be a substitution on domain $FMV(s) \cup FMV(t)$. Then indeed $s\delta >' t\delta$, because $s\delta$ and $t\delta$ are terms, so $s\delta > t\delta$ implies $s\delta >' t\delta$.

$\gtrsim'$ **is monotonic and contains beta**: obvious because $\gtrsim'$ restricted to equal-typed terms is just $\succsim$, which is monotonic and contains beta. □

The relations defined above are not necessarily computable, but they don't need to be: we will only use specific instances. To prove that $l \gtrsim' r$ for all rules, it suffices to prove $l \gtrsim r$. To prove $l \geq' p$ or $l >' p$ for dependency pair constraints, we can use the following result:

Lemma 6.34. *Given meta-terms $l : \sigma_1 \to \ldots \to \sigma_n \to \iota$ and $r : \tau_1 \to \ldots \to \tau_m \to \kappa$, where l is a pattern of the form $f(\vec{s}) \cdot \vec{t}$. If $Z_1, \ldots, Z_n$ are fresh meta-variables and $t_1, \ldots, t_m$ meta-terms, then:*

- *if $l \cdot Z_1 \cdots Z_n \succ r \cdot t_1 \cdots t_m$, then $l >' r$*
- *if $l \cdot Z_1 \cdots Z_n \gtrsim r \cdot t_1 \cdots t_m$, then $l \geq' r$*

Proof. If l is a pattern and of the right form, then also $l \cdot \vec{Z}$ is a pattern of the right form, so we can use meta-stability of the relations $\succ$ and $\gtrsim$.

If $l \cdot Z_1 \cdots Z_n \succ r \cdot t_1 \cdots t_m$, then for all substitutions γ on domain $FMV(l) \cup FMV(r)$ and terms $s_1, \ldots, s_n$:

$$\begin{array}{rcl} (l\gamma) \cdot \vec{s} & = & (l \cdot \vec{Z})\delta \text{ where } \delta = \gamma \cup [Z_1 := s_1, \ldots, Z_n := s_n] \\ & \succ & (r \cdot \vec{t})\delta \text{ by meta-stability of } \succ \\ & = & (r\gamma) \cdot (t_1\delta) \cdots (t_m\delta) \end{array}$$

And thus $l >' r$ as required. The case for $\geq'$ is similar. □

Thus, to prove that a set $\mathcal{P}$ of dependency pairs is chain-free, we can choose for every dependency pair $l \Rrightarrow p\ (A) \in \mathcal{P}$ some meta-terms $\vec{t}$ such that $p \cdot \vec{t}$ has base type, and prove either $l \cdot \vec{Z} \succ p \cdot \vec{t}$ or $l \cdot \vec{Z} \gtrsim p \cdot \vec{t}$. In the examples in this chapter, we will simply choose for each t_i some c_j^σ. This is typically a good choice because the c_j^σ can be thought of as "minimal symbols", as they occur only on the right-hand sides of constraints. However, in some cases it may for instance be preferable to let some t_i be a meta-variable.

To make sure that $\geq'$ satisfies the subterm property when necessary, let us consider the relation $\rhd^{\mathcal{F}}$. This relation can be used to satisfy the subterm property, but is defined on base-type terms, and uses β-reduction instead of allowing subterm steps $\lambda x.s \rhd s$.

Definition 6.35. $\rhd^{\mathcal{F}}$ is the relation on base-type terms (and $\trianglerighteq^{\mathcal{F}}$ its reflexive closure) generated by the following clauses:

- $(\lambda x.s) \cdot t_0 \cdots t_n \rhd^{\mathcal{F}} q$ if $s[x := t_0] \cdot t_1 \cdots t_n \trianglerighteq^{\mathcal{F}} q$;
- $f(s_1, \ldots, s_m) \cdot t_1 \cdots t_n \trianglerighteq^{\mathcal{F}} q$ if $s_i \cdot \vec{\mathsf{c}} \trianglerighteq^{\mathcal{F}} q$ and $f \in \mathcal{F}$;
- $s \cdot t_1 \cdots t_n \trianglerighteq^{\mathcal{F}} q$ if $t_i \cdot \vec{\mathsf{c}} \trianglerighteq^{\mathcal{F}} q$ (s may have any form).

Here, $s \cdot \vec{\mathsf{c}}$ is a term s applied to constants c_i^σ of the right types. We say $(\succsim, \succ)$ respects $\rhd^{\mathcal{F}}$ if $\rhd^{\mathcal{F}}$ is contained in $(\succsim \cup \succ)^*$. Note that, since $\succsim$ contains beta, the first clause is not likely to give any problems. The relation $\rhd^{\mathcal{F}}$ is interesting because if $s \rhd t$ and s has base type, then there are terms $q_1, \ldots, q_n$ and a substitution γ such that $s \rhd^{\mathcal{F}} t\gamma \cdot q_1 \cdots q_n$ (this is easy to see with induction on the size of s). Consequently, $\geq'$ satisfies the subterm property if $(\succsim, \succ)$ respects $\rhd^{\mathcal{F}}$ and $f(\vec{x}) \succsim f^\sharp(\vec{x})$ for all $f \in \mathcal{D}$ (the *marking property*).

Theorem 6.36. *A set of dependency pairs $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ is chain-free if $\mathcal{P}_2$ is chain-free and there is a reduction pair $(\succsim, \succ)$ such that:*

1. *$\tilde{l} \succ \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_1$;*
2. *$\tilde{l} \succsim \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_2$;*
3. *$l \succsim r$ for all $l \Rightarrow r \in \mathcal{R}$;*
4. *if $\mathcal{P}$ is collapsing, then $(\succsim, \succ)$ respects $\rhd^{\mathcal{F}}$, and $f(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for all $f \in \mathcal{D}$.*

Here, $\tilde{l} = l \cdot Z_1 \cdots Z_n$ for fresh meta-variables such that $l \cdot \vec{Z}$ has base type, and $\overline{p}$ may be any base-type meta-term of the form $(p\chi) \cdot t_1 \cdots t_m$, where χ replaces the free variables in p by symbols c_i^σ, and the t_j are meta-terms (which may use the meta-variables $Z_1, \ldots, Z_n$).

Proof. By Theorem 6.31, using the reduction triple generated from $(\succsim, \succ)$. □

Example 6.37. To prove termination of `twice` it suffices to find a reduction pair $(\succsim, \succ)$ such that $(\succsim, \succ)$ respects $\rhd^{\mathcal{F}}$ and satisfies the marking property, and:

$$\begin{array}{rcl}
\mathtt{I}^\sharp(\mathtt{s}(X)) & \underset{(\succsim)}{\succ} & \mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X \\
\mathtt{I}^\sharp(\mathtt{s}(X)) & \underset{(\succsim)}{\succ} & \mathtt{twice}^\sharp(\lambda x.\mathtt{I}(x)) \cdot \mathsf{c}_4^{\mathtt{nat}} \\
\mathtt{I}^\sharp(\mathtt{s}(X)) & \underset{(\succsim)}{\succ} & \mathtt{I}^\sharp(\mathsf{c}_1^{\mathtt{nat}}) \\
\mathtt{twice}^\sharp(F) \cdot X & \underset{(\succsim)}{\succ} & F \cdot (F \cdot \mathsf{c}_2^{\mathtt{nat}}) \\
\mathtt{twice}^\sharp(F) & \underset{(\succsim)}{\succ} & F \cdot \mathsf{c}_3^{\mathtt{nat}} \\
\mathtt{twice}(F) \cdot X & \underset{(\succsim)}{\succ} & F \cdot (F \cdot X) \\
\mathtt{twice}(F) \cdot X & \underset{(\succsim)}{\succ} & F \cdot X \\
\mathtt{I}(\mathtt{0}) & \succsim & \mathtt{0} \\
\mathtt{I}(\mathtt{s}(X)) & \succsim & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\
\mathtt{twice}(F) & \succsim & \lambda y.F \cdot (F \cdot y) \\
\mathtt{twice}(F) \cdot X & \succsim & F \cdot (F \cdot X)
\end{array}$$

At least one of the $\underset{(\succsim)}{\succ}$ must be oriented with $\succ$, and the remaining dependency pairs must be proved chain-free separately.

We consider a polynomial interpretation to the natural numbers. We saw in Theorem 4.10 that without the requirement of strong monotonicity for algebra interpretations, we still have a weak reduction pair. Let $\mathcal{J}(\mathsf{c}_\sigma) = 0_\sigma$ and $\mathcal{J}(@^{\sigma\to\tau}) = \boldsymbol{\lambda} f n \vec{m}.\, \max(f(n, \vec{m}), n(\vec{0}))$ (this is not a higher-order *polynomial*,

but it is a weakly monotonic functional), and furthermore:

$$\begin{aligned}
\mathcal{J}(\mathtt{0}) &= 0 \\
\mathcal{J}(\mathtt{s}) &= \boldsymbol{\lambda} n.n+1 \\
\mathcal{J}(\mathtt{I}) = \mathcal{J}(\mathtt{I}^\sharp) &= \boldsymbol{\lambda} n.n \\
\mathcal{J}(\mathtt{twice}) = \mathcal{J}(\mathtt{twice}^\sharp) &= \boldsymbol{\lambda} f n.\max(n, f(f(n)))
\end{aligned}$$

Let $(\gtrsim, \succ)$ be the reduction pair generated by this interpretation. Indeed $(\gtrsim, \succ)$ respects $\rhd^{\mathcal{F}}$, and $f(\vec{x}) = f^\sharp(\vec{x})$ for all $f \in \mathcal{D}$ (this will be discussed in a bit more detail in Section 6.6.1). Translating the requirements to polynomials, we obtain:

$$\begin{array}{lrcl}
(A) & X+1 & > & X \\
(B) & X+1 & > & 0 \\
(C) & X+1 & > & 0 \\
(D) & \max(X, F(F(X))) & \geq & \max(F(\max(F(0),0)), \max(F(0),0)) \\
(E) & \max(X, F(F(X))) & \geq & \max(F(0),0) \\
(F) & \max(X, F(F(X))) & \geq & \max(F(\max(F(X),X)), \max(F(X),X)) \\
(G) & \max(X, F(F(X))) & \geq & \max(F(X),X) \\
(H) & 0 & \geq & 0 \\
(I) & X+1 & \geq & X+1 \\
(J) & \max(n, F(F(n))) & \geq & \max(F(\max(F(n),n)), \max(F(n),n)) \\
(K) & \max(X, F(F(X))) & \geq & \max(F(\max(F(X),X)), \max(F(X),X))
\end{array}$$

Each of (A), (B), (C), (H) and (I) is obvious; (F), (J) and (K) are duplicates and (D) is implied by them; (E) is implied by (G). (G) is implied by (F), so this leaves only (F) to prove. We use a case analysis: either $X \geq F(X)$ (so by weak monotonicity of F, also $X \geq F(X) \geq F(F(X))$), or $F(X) \geq X$ (so also $F(F(X)) \geq F(X) \geq X$). In the first case, the constraint simplifies to $X \geq X$. In the second case, we get $F(F(X)) \geq F(F(X))$. Either way we are done.

Thus, the AFSM is non-terminating if the set consisting of the `twice` and $\mathtt{twice}^\sharp$ dependency pairs is chain-free. We will finish this proof in Section 6.4.

This completes the basic definition of dependency pairs for AFSMs. It is questionable whether this basis is a huge improvement over the more conventional rule removal: a reduction pair with the subterm property is still a quasi-simplification ordering, and we must always prove $l \gtrsim r$ for all rules $l \Rightarrow r$.

In the next two sections, we will strengthen the approach in two ways. First, in Section 6.4, we will study *formative rules*. This approach allows us to ignore rules which do not really "contribute" to a dependency chain. It is similar to the usable rules approach we saw in Section 6.2.5, but is based on the left-hand sides of rules rather than the right-hand sides.

Then, in Section 6.5, we will see how we can weaken the subterm property for systems where the rules are *abstraction-simple*. This is a significant improvement, which brings the method closer to its first-order counterpart (although due to collapsing dependency pairs, features like usable rules remain a problem).

In Section 6.6 we will discuss how to find a good reduction pair systematically.

6.4 Formative Rules

For the first improvement on the basic dependency pair approach of Section 6.3, let us focus on the $t_i \Rightarrow_{\mathcal{R}}^{*} s_{i+1}$ parts of a dependency chain. Because of these parts, we have to prove $l \gtrsim r$ for all rules $l \Rightarrow r$ whenever we use a reduction pair. In particular with large systems, this can be quite inconvenient. But is it really necessary?

In the first-order approach, it is not. With *usable rules* we can ignore those rules which are not really relevant to a given dependency chain. Unfortunately, this method breaks in the presence of collapsing dependency pairs (we will see a definition for non-collapsing dependency pairs in Chapter 7.6, but is has rather strong restrictions). However, we can define a method which uses the same ideas, but goes in a different direction. Where usable rules depend on the right-hand side of a dependency pair, the *formative rules* introduced here depend on the left-hand side. The intuition behind formative rules is that only the formative rules of some rule $l \Rightarrow r$ can contribute to the creation of its pattern, provided l is a *linear, fully-extended pattern*.

Aimed as this approach is at higher-order systems, we consider the symbols occurring in the left-hand side of a rule or dependency pair along with their type. We consider a fixed set of rules $\mathcal{R}$. First, let $\mathcal{R}^+$ be defined as the set:

$$\begin{aligned}&\{l \Rightarrow r \mid l \Rightarrow r \in \text{the } \beta\text{-saturated extension of } \mathcal{R} \mid r \text{ an abstraction}\} \cup\\ &\{l \cdot Z_1 \cdots Z_n \Rightarrow r \cdot Z_1 \cdots Z_n \mid l \Rightarrow r \in \text{the } \beta\text{-saturated extension of } \mathcal{R}\\ &\qquad\qquad \mid l \cdot \vec{Z} \text{ well-typed}, \ \mathtt{head}(r) \text{ not an abstraction}\}\end{aligned}$$

Note that while $\mathcal{R}^+$ may contain rules which do not occur in $\mathcal{R}$, the rewrite relation $\Rightarrow_{\mathcal{R}^+}$ is included in the transitive closure of the original, $\Rightarrow_{\mathcal{R}}^{+}$. All rules of $\mathcal{R}$ are included in $\mathcal{R}^+$ (n may be chosen 0), except those which are headed by a β-redex. Formative rules will be chosen from $\mathcal{R}^+$:

Definition 6.38 (Formative Rules)**.** For a pattern s, let $Symb(s)$ be inductively defined as follows:

$$\begin{aligned}Symb(\lambda x.s : \sigma) &= \{\langle ABS, \sigma\rangle\} \cup Symb(s)\\ Symb(f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma) &= \{\langle f, \sigma\rangle\} \cup \bigcup_{i=1}^{m} Symb(s_i)\\ Symb(x \cdot s_1 \cdots s_n : \sigma) &= \{\langle VAR, \sigma\rangle\} \cup \bigcup_{i=1}^{n} Symb(s_i) \ \ (n \geq 0)\\ Symb(Z(x_1, \ldots, x_n) : \sigma) &= \emptyset\end{aligned}$$

For $a \in \mathcal{F} \cup \{ABS, VAR\}$, a meta-term $s : \sigma$ *has shape* $\langle a, \sigma\rangle$ if:

- $a = ABS$ and s is an abstraction, *or*
- a is a function symbol and $s = a(\vec{t}) \cdot \vec{q}$, *or*
- $\mathtt{head}(s)$ is a meta-variable application.

For instance the single meta-variable $Z : \sigma$ has shapes $\langle ABS, \sigma\rangle$, $\langle VAR, \sigma\rangle$ and $\langle f, \sigma\rangle$ for all $f \in \mathcal{F}$). The pair $\langle a, \sigma\rangle$ is called a *typed symbol*.

Consider a fixed set of rules $\mathcal{R}$. For two typed symbols A, B, write $A \sqsubseteq_{fo} B$ if there is a rule $l \Rightarrow r \in \mathcal{R}^+$ such that r has shape A, and $B \in Symb(l)$, or l is not both linear and fully extended. Let $\sqsubseteq^*_{fo}$ denote the reflexive-transitive closure of $\sqsubseteq_{fo}$. Intuitively, $A \sqsubseteq^*_{fo} B$ (or: $B \sqsupseteq^*_{fo} A$) can be read as: a term containing the typed symbol B may lead to the formation of a term with shape A.

The *formative symbols* of a pattern s are the typed symbols B such that $A \sqsubseteq^*_{fo} B$ for some $A \in Symb(s)$.

The *formative rules* of a pattern s, notation $FR(s)$, are the rules $l \Rightarrow r \in \mathcal{R}^+$ such that r has shape B for some formative symbol B of s, provided s is linear and fully extended. If this is not the case, then $FR(s) = \mathcal{R}$.

The set of *formative rules of a dependency pair* $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \Rrightarrow p\,(A)$ is $FR(l_1) \cup \ldots \cup FR(l_m)$, or just $\mathcal{R}$ if any of the l_i is not both linear and fully extended. For a set $\mathcal{P}$ of dependency pairs, $FR(\mathcal{P}) = \bigcup_{\rho \in \mathcal{P}} FR(\rho)$.

Note that in a finite system, it is easy to calculate the formative symbols of a term, and consequently the formative rules can easily be found automatically. Rather than $FR(s)$ or $FR(\rho)$ we will sometimes have to write $FR(s, \mathcal{R})$ or $FR(\rho, \mathcal{R})$, when the set $\mathcal{R}$ is not clear from context.

Example 6.39. Recall the rules for the (β-saturated) system `twice`:

$$\begin{array}{lrcl}
(A) & \mathtt{I(0)} & \Rightarrow & \mathtt{0} \\
(B) & \mathtt{I(s}(X)) & \Rightarrow & \mathtt{s(twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\
(C) & \mathtt{twice}(F) & \Rightarrow & \lambda y.F \cdot (F \cdot y) \\
(D) & \mathtt{twice}(F) \cdot X & \Rightarrow & F \cdot (F \cdot X)
\end{array}$$

Here, $\mathcal{R}^+ = \mathcal{R}$, since the only rule of functional type reduces to an abstraction.

In this context, let $l = \mathtt{s}(X)$. Then $Symb(l) = \{\langle \mathtt{s}, \mathtt{nat}\rangle\}$, and:

- the right-hand sides of (B) and (D) both have shape $\langle \mathtt{s}, \mathtt{nat}\rangle$, so $\langle \mathtt{s}, \mathtt{nat}\rangle \sqsubseteq_{fo} \langle \mathtt{s}, \mathtt{nat}\rangle$, $\langle \mathtt{I}, \mathtt{nat}\rangle, \langle \mathtt{twice}, \mathtt{nat}\rangle$;
- the right-hand side of rule (D) does have shapes $\langle \mathtt{I}, \mathtt{nat}\rangle$ and $\langle \mathtt{twice}, \mathtt{nat}\rangle$, but no other rules do, so nothing further is added.

All in all, the formative symbols of l are $\langle \mathtt{s}, \mathtt{nat}\rangle$, $\langle \mathtt{I}, \mathtt{nat}\rangle$, $\langle \mathtt{twice}, \mathtt{nat}\rangle$, and thus l has (B) and (D), but not (A) and (C), as formative rules.

Formative rules are constructed in such a way that to reduce to a term of a certain form, we only need to use its formative rules. With an eye on future extensions, we split the proof into two parts: first we see that, to reduce to a term of a certain form, we can use a reduction of a certain form; second we observe that reductions of such a form use only formative rules.

Definition 6.40. For l a fixed meta-term, s a term and γ a substitution whose domain contains only meta-variables, we say that $s \Rightarrow^*_{\mathcal{R}} l\gamma$ by a *formative l-reduction* if $\lambda\vec{x}.l$ is not a fully extended linear pattern (where $\{\vec{x}\} = FV(l)$), or one of the following holds:

1. $s = l\gamma$ and l is a meta-variable application;
2. $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ and $l = f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ and each $s_i \Rightarrow^*_{\mathcal{R}} l_i\gamma$ by a formative l_i-reduction;
3. $s = x \cdot s_1 \cdots s_m$ and $l = x \cdot l_1 \cdots l_m$ for a variable x and each $s_i \Rightarrow^*_{\mathcal{R}} l_i\gamma$ by a formative l_i-reduction;
4. $s = \lambda x.s'$ and $l = \lambda x.l'$ and $s' \Rightarrow^*_{\mathcal{R}} l'\gamma$ by a formative l'-reduction (and x does not occur in domain or range of γ);
5. $s = (\lambda x.t) \cdot q \cdot \vec{u}$, and $t[x := q] \cdot \vec{u} \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative l-reduction;
6. l is not a meta-variable application and there are a rule $l' \Rightarrow r' \in \mathcal{R}^+$ and a substitution δ such that $s \Rightarrow^*_{\mathcal{R}} l'\delta$ by a formative l'-reduction and $r'\delta \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative l-reduction which does not use clause 6.

If l is a meta-variable application, a formative l-reduction can only be the empty reduction. Roughly, a formative l-reduction avoids steps which do not contribute to the pattern of l and, as we will see later, uses only formative rules of l.

Lemma 6.41. *Let $\mathcal{R}$ be a set of rules and l a closed pattern, s a terminating term and γ a substitution on domain $FMV(l)$. If $s \Rightarrow^*_{\mathcal{R}} l\gamma$, then there is a substitution δ such that $s \Rightarrow^*_{\mathcal{R}} l\delta$ by a formative l-reduction, and $\delta \Rightarrow^*_{\mathcal{R}} \gamma$ (each $\delta(Z) \Rightarrow^*_{\mathcal{R}} \gamma(Z)$).*

Proof. We can safely assume that l is fully extended and linear, for if not we might simply take $\delta := \gamma$. We can also assume that $\mathcal{R}$ is β-saturated, because $\mathcal{R}$ and its β-saturated extension $\mathcal{R}'$ generate the same (transitively closed) rewrite relation, and $\mathcal{R}^+ = \mathcal{R}'^+$.

Towards an induction hypothesis, let $X = \{x_1, \ldots, x_n\}$ be a set of variables such that $\lambda\vec{x}.l$ is a fully extended linear pattern, and the range of γ does not freely contain any variables in X. We will find some δ whose range also does not freely contain any of the variables in X, such that $s \Rightarrow^*_{\mathcal{R}} l\delta$ by a formative l-reduction, and moreover each $\delta(Z) \Rightarrow^*_{\mathcal{R}} \gamma(Z)$. The lemma follows for $X = \emptyset$.

We use induction on s, ordered with $\Rightarrow_{\mathcal{R}} \cup \rhd$ (which is well-founded because s is terminating by assumption). Since $\lambda\vec{x}.l$ is a fully extended pattern, l either has the form $Z(x_1, \ldots, x_n)$ with $X = \{x_1, \ldots, x_n\}$, or l is not a meta-variable application at all. In the first case, choosing $\delta(Z) = \lambda x_1 \ldots x_n.s$, all requirements are satisfied (as $dom(\gamma) = FMV(l) = \{Z\}$, and all variables in X are bound).

Alternatively, l is a variable, abstraction, application or functional term, and thus $l\gamma$ has the same form. Moreover, since l is a pattern, $l\gamma$ cannot be headed by a β-redex. Thus, we can transform the reduction: using the fact that $\mathcal{R}$ is

β-saturated, we can make sure that this reduction does not use any steps of the form $l\gamma \cdot t_1 \cdots t_n \Rightarrow r\gamma \cdot t_1 \cdots t_n$ where $n > 0$ and r an abstraction, or $n \geq 0$ and r headed by a β-redex (we can replace such steps one by one, and by induction on s with $\Rightarrow_{\mathcal{R}}$ we eventually obtain a reduction without such steps).

To prove the claim, we now use a second induction on the length of the resulting $\Rightarrow_{\mathcal{R}}^*$ reduction. Suppose there are no headmost steps in the reduction $s \Rightarrow_{\mathcal{R}}^* l\gamma$, so all reductions take place in a strict subterm. There are three cases:

- If $l = \lambda x.l'$ and $s = \lambda x.s'$ and $s' \Rightarrow_{\mathcal{R}}^* l'\gamma$ (with x not occurring in either domain or range of γ), then the first induction hypothesis with $X' := X \cup \{x\}$ provides δ, which x does not occur in, such that $s' \Rightarrow_{\mathcal{R}}^* l'\delta$ with a formative l'-reduction. Since x does not occur in δ we have $l\delta = \lambda x.(l'\delta)$.
- If $l = f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ and $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$, then write $\gamma = \gamma_1 \cup \ldots \cup \gamma_m$, where $dom(\gamma_i)$ is the restriction of γ to the meta-variables occurring in l_i. By linearity of l, all γ_i have disjoint domains. The first induction hypothesis provides $\delta_1, \ldots, \delta_m$ such that each $s_i \Rightarrow_{\mathcal{R}}^* l_i\delta_i$ by a formative l_i-reduction, and $\delta_i \Rightarrow_{\mathcal{R}}^* \gamma_i$. We can choose $\delta := \delta_1 \cup \ldots \cup \delta_m$ because the δ_i have disjoint domains.
- If $l = x \cdot l_1 \cdots l_m$ for some variable x, we can use a similar reasoning as in the case with a function symbol.

Finally, suppose there are headmost steps in this reduction.

If s has the form $(\lambda x.t) \cdot q \cdot u_1 \cdots u_n$, then note that the first headmost step must be a β-step. We might as well do a β-step immediately (as in the proof of Lemma 6.60) and by the first induction hypothesis we can find a suitable δ such that $s \Rightarrow_\beta t[x := q] \cdot \vec{u} =: v$ and $v \Rightarrow_{\mathcal{R}}^* l\delta$ by a formative l-reduction as required. By clause 5 we are done.

If s does not have this form, there must be at least one headmost step which is not a β-reduction. Suppose the reduction has the form $s \Rightarrow_{\mathcal{R}}^* t \Rightarrow_{\mathcal{R}} q \Rightarrow_{\mathcal{R}}^* l\gamma$, where $t = l'\gamma' \cdot u_1 \cdots u_n$ and $q = r'\gamma' \cdot u_1 \cdots u_n$ for some rule $l' \Rightarrow r'$.

Let $l'' := l' \cdot Z_1 \cdots Z_n$ and $r'' := r' \cdot Z_1 \cdots Z_n$ for suitably typed meta-variables $Z_1, \ldots, Z_n$, and choose $\gamma'' := \gamma' \cup [Z_1 := u_1, \ldots, Z_n := u_n]$. Then $t = l''\gamma''$ and $q = r''\gamma''$, and moreover $l'' \Rightarrow r'' \in \mathcal{R}^+$: r' is not headed by a β-redex (since we previously replaced such rules by a β-reduced version if possible), so if r' is not an abstraction this is evident, and if r' is an abstraction then $n = 0$ (for the same reason), so $l'' = l'$ and $r'' = r'$.

We can apply the second induction hypothesis, and obtain that $s \Rightarrow_{\mathcal{R}}^* l''\chi$ by a formative l''-reduction for some substitution χ which reduces to γ''. Since $s \Rightarrow_{\mathcal{R}}^+ r''\chi$ and $r''\chi \Rightarrow_{\mathcal{R}}^* r''\gamma'' \Rightarrow_{\mathcal{R}}^* l\gamma$, the $\Rightarrow_{\mathcal{R}}$-part of the first induction hypothesis provides that $r''\chi \Rightarrow_{\mathcal{R}}^* l\delta$ by a formative l-reduction for some substitution δ. Together, $s \Rightarrow_{\mathcal{R}}^* l\delta$ by a formative l-reduction. This holds by a separate induction: if $s \Rightarrow_{\mathcal{R}}^* l_1\gamma_1$ by a formative l_1-reduction, $l_1 \Rightarrow r_1 \in \mathcal{R}^+$, and $r_1\gamma_1 \Rightarrow_{\mathcal{R}}^* l_2\gamma_2$ by a formative l_2-reduction, then $s \Rightarrow_{\mathcal{R}}^* l_2\gamma_2$ by a formative l_2-reduction using induction on the number of topmost non-beta steps in $r_1\gamma_1 \Rightarrow_{\mathcal{R}}^* l_2\gamma_2$, and Clause 6. □

Lemma 6.41 provides the first part of the claim made before: that to reduce to a term of a certain form l, we only need the formative rules of l. The second part is provided by Lemma 6.42.

Lemma 6.42. *If $s \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative l-reduction, then this reduction only uses rules in $FR(l, \mathcal{R})$.*

That is, if a formative l-reduction uses only rules in $\mathcal{R}^+$ for some given set $\mathcal{R}$, then actually it uses only rules in $FR(l, \mathcal{R})$.

Proof. We make two observations:

(**) If $l \trianglerighteq l'$ then $FR(l') \subseteq FR(l)$.

This is obvious because all typed symbols occurring in l' also occur in l.

(***) If $l' \Rightarrow r'$ is a formative rule of l, then $FR(l') \subseteq FR(l)$.

This holds by transitivity of $\sqsubseteq^*_{fo}$.

We prove the statement by induction on the definition of a formative l-reduction. If $s = l\gamma$ then the reduction uses no rules at all. If l is not a fully extended linear pattern, then $FR(l) = \mathcal{R}$, and we are done because the relations $\Rightarrow^*_{\mathcal{R}}$ and $\Rightarrow^*_{\mathcal{R}^+}$ are the same. If $s \Rightarrow^*_{\mathcal{R}} l\gamma$ is a formative l-reduction by clauses 2, 3 or 4 of Definition 6.40, then we are done by the induction hypothesis and observation (**). If it holds by clause 5, then $s \Rightarrow^*_{\beta} t \Rightarrow^*_{\mathcal{R}} l\gamma$, where $t \Rightarrow^*_{\mathcal{R}} l\gamma$ uses only formative rules by the induction hypothesis, and the β-step uses no rules.

Finally, suppose $s \Rightarrow^*_{\mathcal{R}} l\gamma$ by clause 6 of Definition 6.40, so there are a rule $l' \Rightarrow r' \in \mathcal{R}^+$ and substitution δ such that $s \Rightarrow^*_{\mathcal{R}} l'\delta$ by a formative l'-reduction and $r'\delta \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative l-reduction without any headmost steps other than perhaps β. Suppose we can see that $l' \Rightarrow r' \in FR(l)$. Then by the induction hypothesis, the reduction $s \Rightarrow^*_{\mathcal{R}} l'\delta$ uses only rules in $FR(l')$, which also are in $FR(l)$ by (***). Also, the reduction $r'\delta \Rightarrow^*_{\mathcal{R}} l\gamma$ uses only rules in $FR(l)$ by the induction hypothesis. Thus, if we can indeed prove this, we are done.

First suppose that $\mathtt{head}(r')$ is a meta-variable application. Then note that, whatever the form of l is, $Symb(l)$ contains a pair $\langle f, \sigma\rangle$, where σ is the type of l (and also the type of s, l' and r'), and $f \in \mathcal{F} \cup \{\mathtt{ABS}, \mathtt{VAR}\}$. Since r' has *any* shape of type σ, we immediately see that $l' \Rightarrow r' \in FR(l)$.

If $\mathtt{head}(r')$ is an abstraction, then r' has shape $\langle \mathtt{ABS}, \sigma\rangle$. Since q is an abstraction, $l\gamma$ must also be one, and this can only be the case if l is an abstraction. But then $Symb(l)$ indeed contains $\langle \mathtt{ABS}, \sigma\rangle$, so $l' \Rightarrow r' \in FR(l)$ for that reason.

Finally, if $\mathtt{head}(r')$ is a function symbol, then q is not a β-redex. As the reduction $q \Rightarrow^*_{\mathcal{R}} l\gamma$ does not use other headmost steps, we must have $q \Rightarrow^*_{\mathcal{R},in} l\gamma$, and l must have the form $f(l_1, \ldots, l_k) \cdot l_{k+1} \cdots l_m$, where f is also the head-symbol of r'. But then $\langle f, \sigma\rangle \in Symb(l)$, so also $l' \Rightarrow r' \in FR(l)$. □

Definition 6.43. Let ρ be a dependency pair or $\mathtt{beta}$. We say that $s \Rightarrow^*_{\mathcal{R}} t$ by a formative ρ-reduction if either $\rho = \mathtt{beta}$ and $s = t$, or $\rho = l \Rrightarrow p$, t has the form $l\gamma$, and $s \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative l-reduction.

A dependency chain is called *formative* if for all i, $t_i \Rightarrow^*_{\mathcal{R},in} s_{i+1}$ by a formative ρ_{i+1}-reduction.

We can now see that a system is terminating, if and only if there is a minimal *formative* dependency chain. When using a reduction pair, this means we only have to prove $l \succsim r$ for all formative rules of the set $\mathcal{P}$, rather than for all rules.

Theorem 6.44. *An AFSM $\mathcal{R}$ is non-terminating if and only if* $\mathsf{DP}(\mathcal{R})$ *admits a minimal formative dependency chain.*

Proof. If there is a minimal formative dependency chain then by Lemma 6.27 the AFSM is non-terminating. For the other direction, we adapt the proof of Lemma 6.25. Given a non-terminating term a, we will construct a minimal formative dependency chain. Let q_{-1} be an MNT subterm of a.

For any $i \in \mathbb{N} \cup \{-1\}$, let q_i be some MNT term, and suppose $t_i = q_i^\sharp$. We consider an infinite reduction starting in q_i. As in the proof of Lemma 6.25, q_i can only have the form $(\lambda x.q) \cdot u \cdot v_1 \cdots v_n$ or $q_i = f(u_1, \ldots, u_n) \cdot u_{n+1} \cdots u_m$ with $f \in \mathcal{D}$. In the first case, we take $s_{i+1} = t_i$; the reduction $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ is empty, so formative. Proceeding exactly as in the proof of Lemma 6.25, we also find $\rho_{i+1},\ q_{i+1}$ and t_{i+1}.

In the second case, any infinite reduction starting in q_i must eventually take a headmost step. That is, we can find a rule $l \Rightarrow r$ and term $q_i'' := l\gamma \cdot u'_{j+1} \cdots u'_m$ such that $q_i \Rightarrow^*_{in} q_i''$, and $r\gamma \cdot u'_{j+1} \cdots u'_m$ is still non-terminating. As before, we can safely assume that either $m = j$ or r is not an abstraction or application headed by a β-redex. Let $l' := l \cdot Z_{j+1} \cdots Z_m$ for fresh meta-variables $Z_{j+1}, \ldots, Z_m$ and $r' := r \cdot Z_{j+1} \cdots Z_m$. Let $\gamma' := \gamma \cup [Z_{j+1} := u'_{j+1}, \ldots, Z_m := u'_m]$.

Recall that $q_i = f(u_1, \ldots, u_n) \cdot u_{n+1} \cdots u_m$. We can write $l' = f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$, and have that $u_i \Rightarrow^*_{\mathcal{R}} l_i\gamma'$ for all i. If l' is not linear, then the reduction $q_i \Rightarrow^*_{\mathcal{R},in} q_i''$ is a formative l'-reduction; in this case, we choose $q_i' := q_i''$. If l' is linear, then we can write $\gamma' = \gamma_1 \cup \ldots \cup \gamma_k$ where all γ_i have disjoint domains containing all meta-variables in l_i. By Lemma 6.41 we can find substitutions $\delta_1, \ldots, \delta_k$ on the same domains, such that for each i, $v_i \Rightarrow^*_{\mathcal{R}} l_i\delta_i$ by a formative l_i-reduction, and $\delta_i \Rightarrow^*_{\mathcal{R}} \gamma_i$. Taking $\delta := \delta_1 \cup \ldots \cup \delta_k$ (which is unproblematic due to the different domains), we have that $q_i \Rightarrow^*_{\mathcal{R},in} l'\delta$ using a formative l'-reduction, and $r'\delta \Rightarrow^*_{\mathcal{R}} r'\gamma' = r\gamma \cdot u'_{j+1} \cdots u'_m$, so is still non-terminating. We take $q_i' := l''\delta$.

The rest of the proof continues exactly as in Lemma 6.25: the proof is not affected by the exact choice for q_i', it suffices if q_i' admits a headmost step without losing termination. The resulting dependency pair ρ_{i+1} has left-hand side $l'^\sharp$, and $s_{i+1} = q_i'^\sharp$. We thus indeed have that $t_i \Rightarrow^*_{\mathcal{R},in} s_{i+1}$ by a formative ρ_{i+1}-reduction. □

Defining the notion of a *formative chain-free* set of dependency pairs as a set $\mathcal{P}$ which does not admit a minimal formative dependency chain, we quickly obtain improved variations of Theorems 6.31 and 6.36. I will only pose the formative version of Theorem 6.36, as this is the one we will primarily use:

Theorem 6.45. *A set of dependency pairs $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ is formative chain-free if $\mathcal{P}_2$ is formative chain-free and there is a reduction pair $(\succsim, \succ)$ such that:*

1. *$\tilde{l} \succ \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_1$;*
2. *$\tilde{l} \succsim \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_2$;*
3. *$l \succsim r$ for all $l \Rightarrow r \in FR(\mathcal{P}, \mathcal{R})$;*
4. *if $\mathcal{P}$ is collapsing, then $(\succsim, \succ)$ respects $\rhd^{\mathcal{F}}$, and $f(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for all $f \in \mathcal{D}$.*

Here, $\tilde{l} = l \cdot Z_1 \cdots Z_n$ for fresh meta-variables such that $l \cdot \vec{Z}$ has base type, and $\overline{p}$ may be any base-type meta-term of the form $(p\chi) \cdot t_1 \cdots t_m$, where χ replaces the free variables in p by symbols $\mathtt{c}_i^\sigma$, and the t_j are meta-terms (which may use the meta-variables $Z_1, \ldots, Z_n$).

Proof. In the proof of Theorem 6.31 (resp. 6.36), the constraint $l \succsim r$ is only used to obtain that $t_i \succsim s_{i+1}$. When considering formative dependency chains, we have $t_i \succsim FR(\mathcal{P}, \mathcal{R}) s_{i+1}$ for all i: if $\rho_{i+1} = \mathtt{beta}$ because the reduction is empty, if $\rho_{i+1} = l \Rrightarrow p$ with l non-linear because $\mathcal{R} \subseteq FR(\mathcal{P}, \mathcal{R})$ and if $\rho_{i+1} = f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ with a linear left-hand side, then we merely need the formative rules of all l_i by Lemma 6.42 – all of these are included in $FR(\mathcal{P})$. □

> *Comment:* Note that we do *not* have the result that a set $\mathcal{P}$ is chain-free if and only if it is formative chain-free and certain requirements are met: Theorem 6.45 can only be used at the start of a termination proof using dependency pairs. I have not been able to prove or disprove that it is possible to transform (minimal) dependency chains into formative chains.

Example 6.46. Consider the set $\mathcal{P}$ consisting of the collapsing dependency pair of the `eval` system:

$$\mathcal{P} = \{\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x), X, Y), Z) \Rrightarrow \mathtt{dom}^\sharp(X, Y, Z)\ \ \{F : 1\}\}$$

The formative rules of this set are:

$$\begin{array}{rcl} \mathtt{dom}(X, Y, \mathtt{0}) & \Rightarrow & X \\ \mathtt{dom}(\mathtt{0}, \mathtt{0}, Z) & \Rightarrow & \mathtt{0} \\ \mathtt{eval}(\mathtt{fun}(F, X, Y), Z) & \Rightarrow & F \cdot \mathtt{dom}(X, Y, Z) \end{array}$$

Thus, to prove that this set is formative chain-free, we merely need to find a reduction pair with $l \succsim r$ for these three rules, and $l \succ p$ for the dependency pair. This saves two additional constraints.

Example 6.47. Recall the result from Example 6.37, which goes through with formative chain-freeness as well as chain-freeness. To prove termination of `twice`, it suffices to prove that $\mathcal{P}$ is formative chain-free, where $\mathcal{P}$ consists of the four dependency pairs involving `twice`. Since $Symb(l) = \emptyset$ for all argument positions of the left-hand sides of these dependency pairs, $FR(\mathcal{P}) = \emptyset$. Thus, using the type changing transformation, it suffices to orient the following constraints:

$$\begin{array}{rcl} \mathtt{twice}^\sharp(F) \cdot X & \succ & F \cdot (F \cdot \mathtt{c}_2^{\mathtt{nat}}) \\ \mathtt{twice}^\sharp(F) \cdot X & \succ & F \cdot \mathtt{c}_3^{\mathtt{nat}} \\ \mathtt{twice}(F) \cdot X & \succ & F \cdot (F \cdot X) \\ \mathtt{twice}(F) \cdot X & \succ & F \cdot X \end{array}$$

We do not need to consider any rules! These constraints are easily handled with for example a recursive path ordering (which follows the constraints from Chapter 6.6.2). For instance the first clause, choosing $\mathtt{twice}^\sharp \blacktriangleright @^\sigma$ for all σ:

$$\begin{array}{ll} & \overline{\pi}(\mu(\mathtt{twice}^\sharp(F) \cdot X)) \\ = & @^\star(\mathtt{twice}^\sharp(F), X) \\ \Rightarrow_{\mathtt{put}} & @^\star(\mathtt{twice}^{\sharp\star}(F), X) \\ \Rightarrow_{\mathtt{abs}} & @^\star(\lambda x.\mathtt{twice}^{\sharp\star}_{\mathtt{nat}}(F, x), X) \\ \Rightarrow_{\mathtt{select}} & \mathtt{twice}^{\sharp\star}(F, @^\star(\lambda x.\mathtt{twice}^{\sharp\star}_{\mathtt{nat}}(F, x), X)) \\ \Rightarrow_{\mathtt{select}} & \mathtt{twice}^{\sharp\star}(F, X) \\ \Rightarrow_{\mathtt{copy}} & @(\mathtt{twice}^{\sharp\star}_{\mathtt{nat}\to\mathtt{nat}}(F, X), \mathtt{twice}^{\sharp\star}(F, X)) \\ \Rightarrow_{\mathtt{select}} & @(F, \mathtt{twice}^{\sharp\star}(F, X)) \\ \Rightarrow_{\mathtt{copy}} & @(F, @(\mathtt{twice}^{\sharp\star}_{\mathtt{nat}\to\mathtt{nat}}(F, X), \mathtt{twice}^{\sharp\star}(F, X))) \\ \Rightarrow_{\mathtt{select}} & @(F, @(F, \mathtt{twice}^{\sharp\star}(F, X))) \\ \Rightarrow_{\mathtt{bot}} & @(F, @(F, \bot_{\mathtt{nat}})) \\ = & \overline{\pi}(\mu(F \cdot (F \cdot \mathtt{c}_2^{\mathtt{nat}}))) \end{array}$$

Note that both the `eval` and the `twice` examples are unfortunate when using formative rules, because there is only one base type. In systems with only one base type, if there are formative rules, then every collapsing rule is formative. Nevertheless, even in such cases, there is something to gain, as the last two examples have demonstrated. In the case of `twice` the use of formative rules was even essential: if we had to prove $l \succsim r$ for all rules in addition to the dependency pair constraints, we could not succeed with StarHorpo; nor have I been able to find any proof using polynomial interpretations.

Comment: The use of formative rules is by no means restricted to the higher-order setting. In particular for many-sorted TRSs (which use types), or innermost rewriting (where types may be added by [39]), the method might also give advantages in the first-order case. However, at the moment the relative power of formative rules in this setting has not been investigated.

6.5 The Dynamic Dependency Pair Approach for Abstraction-simple AFSMs

Now let us move on to a very important improvement: a way to weaken the subterm property. Both in Theorem 6.36 and in Theorem 6.45, the reduction pair used must respect $\rhd^{\mathcal{F}}$, a rather strong restriction. This is due to subterm reduction steps which may occur in a dependency chain.

However, an important observation about these subterm reduction steps is the following: they *only* happen when substituting a previously bound variable. The subterm property of]Definition 6.30 does not use this fact at all. But it does create some potential, which we will exploit in this section.

To use the potential from this observation, we will pay special attention to *abstraction-simple* systems. In an abstraction-simple AFSM we can (mostly) avoid reducing terms below an abstraction. Knowing this, a subterm step in a dependency chain only concerns terms *which have never been reduced*. Thus, we could for instance use argument filterings, provided we do not filter inside abstractions.

6.5.1 Intuition

The ideas behind the abstraction-simplicity restriction originates in the notion of *weak reductions*, defined in [26] (following a definition from Howard in 1968). A weak reduction in the λ-calculus does not use steps between a λ-abstraction and its binder. This notion generalises to AFSMs in the obvious way.

Consider AFSMs where the left-hand sides of all rules are *linear* (so no free variables occur more than once), and *free of abstractions* (so the λ symbol does not occur in them, except perhaps in the form $\lambda\vec{x}.F(\vec{x})$). As it turns out, we can prove the following statement:

Claim: *in a left-linear, left-abstraction-free AFSM, if there is a minimal dependency chain, then there is one where the reduction $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ is a weak reduction.*

To see why this matters, let us consider a colouring of the function symbols. In a given term s, make all symbol occurrences either red or green: red if the symbol occurs between an abstraction and its binder, green otherwise. So if $s = C[f(t_1, \ldots, t_n)]$, make the f red if some t_i contains freely a variable which is bound in s, green if not. We say s is well-coloured if it uses this colouring. Colour the rules in the same way; by the restrictions, the left-hand sides are entirely green, while the right-hand sides may contain red symbols.

Now consider a weak reduction step on a well-coloured term. If the term is reduced by a coloured rule, then the result is also well-coloured. If the term is reduced with a β-step, then the result may have some red symbols outside an abstraction. However, it can become well-coloured again by painting these red symbols green. We never have to paint green symbols red. Inventing notation, we can summarise this as follows:

Claim: *if $s \Rightarrow_{\mathcal{R},weak} t$, then $colour(s) \Rightarrow_{\mathcal{R}_{colour}} \cdot \Rightarrow^*_{make\_green} colour(t)$.*

Combining the two claims, we can colour dependency chains. In the two

cases where we need a subterm step (2d and 3b), we take a term which was originally below an abstraction, reduce it to a subterm which still contains the bound variable, and substitute it. Importantly, the subterm clause $q \trianglerighteq u$ can be derived with steps $\lambda x.s \triangleright s,\ s_1 \cdot s_2 \triangleright s_i$ and $f(s_1, \ldots, s_n) \triangleright s_i$, *where the* f *is always a red symbol*.

Considering the red and green symbols as different symbols altogether (related only by the $make\_green$ rules) we thus see that it will not give problems to use an argument filtering, provided we use it only for the green symbols!

This summarises the ideas which we shall use to simplify the limited subterm property. Since colours do not work so well in black-and-white print, let us use tags instead: a red symbol f corresponds with a symbol f^-, and a green symbol remains unchanged. Moreover, if we focus on the colours, and forget about the weak reductions, it turns out that we do not need to require that the left-hand sides of rules contain no λ-abstractions at all: it suffices if the rules are *abstraction-simple*.

6.5.2 Abstraction-simple AFSMs

To avoid reductions below an abstraction, we will have to restrict attention to systems where, intuitively, matching is local. That is, we should not need to use a reduction somewhere deep inside a term (possibly inside an abstraction) to make it possible to do a reduction at the top. Additionally, we should avoid matching on an abstraction.

Formally, the AFSM should be *left-linear*, *fully extended* and have *simple meta-applications*. Each of these restrictions we have seen before, and they are all satisfied by many, if not most, common examples. In the termination problem database 8.0.1 (which only supports AFSs, so systems which have simple meta-variables by definition), 143 out of 156 systems satisfy all requirements.

Moreover, to restrict to weak reductions, AFSs should satisfy one additional technical requirement, which is automatically satisfied by AFSMs which originate from an AFS, HRS, CRS or CRSX: all rules should have base type, or all meta-variables in the rules take at most one argument. This excludes systems like:

$$\begin{array}{rcl} \mathtt{g}(\mathtt{0},\mathtt{0}) & \Rightarrow & \mathtt{h}(\lambda x.\mathtt{f}(x)) \\ \mathtt{f}(Z) & \Rightarrow & \lambda y.\mathtt{g}(Z,y) \\ \mathtt{h}(\lambda xy.F(x,y)) & \Rightarrow & F(\mathtt{0},\mathtt{0}) \end{array}$$

In the infinite reduction $\mathtt{g}(\mathtt{0},\mathtt{0}) \Rightarrow \mathtt{h}(\lambda x.\mathtt{f}(x)) \Rightarrow \mathtt{h}(\lambda xy.\mathtt{g}(x,y)) \Rightarrow \mathtt{g}(\mathtt{0},\mathtt{0}) \Rightarrow \ldots$ we cannot avoid reducing below an abstraction, because the `f`-step is needed to create the abstraction required for the `h`-rule. However, systems like this, if they ever occur in practice (which is unlikely, given that systems in all common formalisms satisfy the clause), can be η-expanded to obtain a system where the clause *is* satisfied by Transformation 2.14.

Definition 6.48. An AFSM with rules $\mathcal{R}$ is *abstraction-simple* if for all left-hand-sides l of a rule:

- l *is linear:* no meta-variable occurs twice in l;
- l *is fully extended and has simple meta-variable applications:* meta-variable applications occur only in the form $\lambda x_1 \ldots x_n.Z(x_1, \ldots, x_n)$, and do not occur below another abstraction;
- either all meta-variable applications in l take 0 or 1 arguments, or all rules in $\mathcal{R}$ have base type.

Example 6.49. Both the systems `twice` and `eval` are abstraction-simple.

6.5.3 Tagging Unreducable Symbols

Obviously, when there are rules where the left-hand side contains an abstraction, such as $\mathsf{f}(\lambda x.\mathsf{g}(x), F) \Rightarrow r$, it may be impossible to avoid reducing inside an abstraction in order to create a redex. However, the colouring intuition still goes through; we merely need to "paint symbols green" a few times more.

Following the colouring intuition, we will mark all function symbols which occur between a λ-abstraction and its binder with a special tag ("colouring red"). The symbol can only be reduced by removing its tag first ("painting green").

Definition 6.50. Let $\mathcal{F}^-$ be the set $\{f^- : \sigma \mid f : \sigma \in \mathcal{F}\}$, so a set containing a "tagged" symbol f^- for all function symbols $f \in \mathcal{F}$. For a set of variables A and a meta-term s, define $\mathsf{tag}_A(s)$ as follows:

$$\begin{array}{rcl}
\mathsf{tag}_A(x) & = & x \\
\mathsf{tag}_A(s \cdot t) & = & \mathsf{tag}_A(s) \cdot \mathsf{tag}_A(t) \\
\mathsf{tag}_A(\lambda x.s) & = & \lambda x.\mathsf{tag}_{A \cup \{x\}}(s) \\
\mathsf{tag}_A(f(s_1, \ldots, s_n)) & = & \begin{cases} f(\mathsf{tag}_A(s_1), \ldots, \mathsf{tag}_A(s_n)) & \text{if } FV(f(\vec{s})) \cap A = \emptyset \\ f^-(\mathsf{tag}_A(s_1), \ldots, \mathsf{tag}_A(s_n)) & \text{if } FV(f(\vec{s})) \cap A \neq \emptyset \end{cases} \\
\mathsf{tag}_A(Z(s_1, \ldots, s_n)) & = & Z(\mathsf{tag}_A(s_1), \ldots, \mathsf{tag}_A(s_n))
\end{array}$$

We denote $\mathsf{tag}(s) := \mathsf{tag}_\emptyset(s)$. Define $\mathcal{R}^{\mathsf{tag}} := \{l \Rightarrow \mathsf{tag}_\emptyset(r) \mid l \Rightarrow r \in \mathcal{R}\} \cup \{f^-(Z_1, \ldots, Z_n) \Rightarrow f(Z_1, \ldots, Z_n) \mid f^- \in \mathcal{F}^-\}$. For a meta-term of the form $s := f^\sharp(\vec{t})$ also $\mathsf{tag}(s)$ is defined: since at the top-level no variables are bound, $\mathsf{tag}(s) = f^\sharp(\mathsf{tag}(t_1), \ldots, \mathsf{tag}(t_n))$.

Note that, apart from the untagging rules, $\mathcal{R}^{\mathsf{tag}}$ is not all that different from $\mathcal{R}$: $\mathsf{tag}(r)$ is almost exactly r, only the symbols below an abstraction may be marked with a $-$ sign.

Example 6.51. $\mathsf{tag}(\mathsf{f}(\lambda x.\mathsf{g}(x, \mathsf{g}(\mathsf{0})))) = \mathsf{f}(\lambda x.\mathsf{g}^-(x, \mathsf{g}(\mathsf{0})))$.

Example 6.52. Consider our running example `twice` (with β-saturated rules):

$$\begin{array}{rcl} \mathtt{I}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\ \mathtt{I}(\mathtt{s}(X)) & \Rightarrow & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\ \mathtt{twice}(F) & \Rightarrow & \lambda y.F \cdot (F \cdot y) \\ \mathtt{twice}(F) \cdot X & \Rightarrow & F \cdot (F \cdot X) \end{array}$$

These rules are left-linear, fully extended and have simple meta-variable applications. Moreover, all meta-variables take at most one argument. Thus, the system is abstraction-simple. $\mathcal{R}^{\mathsf{tag}}$ consists of the following rules:

$$\begin{array}{rcl} \mathtt{I}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\ \mathtt{I}(\mathtt{s}(X)) & \Rightarrow & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}^-(x)) \cdot X) \\ \mathtt{twice}(F) & \Rightarrow & \lambda y.F \cdot (F \cdot y) \\ \mathtt{twice}(F) \cdot X & \Rightarrow & F \cdot (F \cdot X) \end{array} \qquad \begin{array}{rcl} \mathtt{0}^- & \Rightarrow & \mathtt{0} \\ \mathtt{s}^-(Z) & \Rightarrow & \mathtt{s}(Z) \\ \mathtt{I}^-(Z) & \Rightarrow & \mathtt{I}(Z) \\ \mathtt{twice}^-(F) & \Rightarrow & \mathtt{twice}(F) \end{array}$$

Note that, when we prove termination, we would typically ignore the symbols $\mathtt{0}^-, \mathtt{s}^-, \mathtt{twice}^-$: they do not occur in any other rules, so we could just merge them with their untagged symbol.

In the proofs later on in this section, we will use a number of properties of $\mathcal{R}^{\mathsf{tag}}$, given by Lemmas 6.53–6.60.

Lemma 6.53. $\mathsf{tag}_{X \cup Y}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}_X(s)$ *for any set of rules* $\mathcal{R}$*. If the variables in* Y *do not occur in* s*, then even* $\mathsf{tag}_{X \cup Y}(s) = \mathsf{tag}_X(s)$.

Proof. Easy induction on the size of s. We only use the untagging rules $f^-(\vec{Z}) \Rightarrow f(\vec{Z})$. □

Lemma 6.54. *Let* s *be a term and* γ *a substitution whose domain contains only variables. Then* $\mathsf{tag}(s)\gamma^{\mathsf{tag}} = \mathsf{tag}(s\gamma)$*. Here,* $\gamma^{\mathsf{tag}} = [x := \mathsf{tag}(\gamma(x)) \mid x \in dom(\gamma)]$.

Proof. We prove by induction on the size of s: for any set of variables A whose elements do not occur in either domain or range of γ, we have $\mathsf{tag}_A(s)\gamma^{\mathsf{tag}} = \mathsf{tag}_A(s\gamma)$.

If s is a variable not in $dom(\gamma)$, both sides are just s.

If s is a variable in $dom(\gamma)$, we must see that $\mathsf{tag}(\gamma(s)) = \mathsf{tag}_A(\gamma(s))$, which holds by the second part of Lemma 6.53.

If s is an application $t \cdot q$, then $\mathsf{tag}_A(s)\gamma^{\mathsf{tag}} = (\mathsf{tag}_A(t)\gamma^{\mathsf{tag}}) \cdot (\mathsf{tag}_A(q)\gamma^{\mathsf{tag}})$, which by the induction hypothesis equals $\mathsf{tag}(t\gamma) \cdot \mathsf{tag}(q\gamma) = \mathsf{tag}((t\gamma) \cdot (q\gamma)) = \mathsf{tag}(s\gamma)$.

If s is a functional term $f(s_1, \ldots, s_n)$ then also the induction hypothesis on each of the s_i suffices, because $A \cap FV(s) = A \cap FV(s\gamma)$, which is easy to see by the requirements on A.

Finally, if s is an abstraction $\lambda x.t$, then $\mathsf{tag}_A(s)\gamma^{\mathsf{tag}} = \lambda x.\mathsf{tag}_{A \cup \{x\}}(s)\gamma^{\mathsf{tag}}$. By α-conversion, we can assume the x is fresh, so does not occur in domain or range of γ. Thus, by the induction hypothesis this term equals $\lambda x.\mathsf{tag}_{A \cup \{x\}}(t\gamma) = \mathsf{tag}(\lambda x.t\gamma) = \mathsf{tag}(s\gamma)$. □

Lemma 6.55. *Let s be a term, and γ a substitution on domain A, which contains only variables. Let B be a set of variables disjoint from A. Let $\gamma' := [x := \mathsf{tag}_B(\gamma(x)) \mid x \in A]$. Then $\mathsf{tag}_{A\cup B}(s)\gamma' \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}_B(s\gamma)$.*

Proof. By induction on the size of s.

If s is a variable not in A, then both sides are just s.

If s is a variable in A, then $\mathsf{tag}_{A\cup B}(s)\gamma' = \gamma'(s) = \mathsf{tag}_B(\gamma(s)) = \mathsf{tag}_B(s\gamma)$.

If s is an application, we merely apply the induction hypothesis.

If s is an abstraction $\lambda x.t$, then $\mathsf{tag}_{A\cup B}(s)\gamma' = \lambda x.(\mathsf{tag}_{A\cup B\cup\{x\}}(t)\gamma')$. By Lemma 6.53 (note that x is fresh), this equals $\lambda x.(\mathsf{tag}_{A\cup B\cup\{x\}}(t)\gamma'')$, where $\gamma'' = [x := \mathsf{tag}_{B\cup\{x\}}(\gamma(x)) \mid x \in A]$. By the induction hypothesis this term reduces to $\lambda x.\mathsf{tag}_{B\cup\{x\}}(t\gamma) = \mathsf{tag}_B(s\gamma)$.

If s is a function application $f(s_1,\ldots,s_n)$, consider two possibilities: either $A \cap FV(s) = \emptyset$ or $A \cap FV(s) \neq \emptyset$.

In the first case, $\mathsf{tag}_{A\cup B}(s)\gamma' = \mathsf{tag}_B(s)\gamma'$ by Lemma 6.53, $= \mathsf{tag}_B(s) = \mathsf{tag}_B(s\gamma)$ because the variables in $dom(\gamma)$ do not occur in s.

In the second case, $\mathsf{tag}_{A\cup B}(s)\gamma' = f^-(\mathsf{tag}_{A\cup B}(s_1)\gamma',\ldots,\mathsf{tag}_{A\cup B}(s_n)\gamma')$, and by the induction hypothesis this $\Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} f^-(\mathsf{tag}_B(s_1\gamma),\ldots,\mathsf{tag}_B(s_n\gamma))$. If $B \cap FV(s) \neq \emptyset$ this term equals $\mathsf{tag}_B(s\gamma)$, otherwise it reduces to it in one untagging step. □

Lemma 6.56. *If $s \Rightarrow_\beta t$ by a headmost step, then $\mathsf{tag}_B(s) \Rightarrow_\beta \cdot \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}_B(t)$ for all rule sets $\mathcal{R}$ and sets of variables B.*

Proof. We can write $s = (\lambda x.q)\cdot u\cdot\vec{v}$, and $t = q[x := u]\cdot\vec{v}$. Thus:

$$\begin{array}{lll} \mathsf{tag}_B(s) & = & (\lambda x.\mathsf{tag}_{B\cup\{x\}}(q))\cdot\mathsf{tag}_B(u)\cdot\mathsf{tag}_B(\vec{v}) \\ & \Rightarrow_\beta & \mathsf{tag}_{B\cup\{x\}}(q)[x := \mathsf{tag}(u)]\cdot\mathsf{tag}_B(\vec{v}) \\ & \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} & \mathsf{tag}_B(q[x := u])\cdot\mathsf{tag}_B(\vec{v}) = \mathsf{tag}_B(t) \text{ by Lemma 6.55} \end{array}$$

□

Lemma 6.57. *Let s be a meta-term, and γ a substitution whose domain contains all meta-variables in $FMV(s)$, and no variables. Then $\mathsf{tag}(s)\gamma^{\mathsf{tag}} \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}(s\gamma)$.*

Proof. We prove the following claim by induction on the size of s: for any set of variables A, which do not occur in the range of γ: $\mathsf{tag}_A(s)\gamma^{\mathsf{tag}} \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}_A(s\gamma)$. We use only the untagging rules $f^-(\vec{Z}) \Rightarrow f(\vec{Z})$. Consider the form of s.

If s is a variable, both sides are just s.

If s is an abstraction, application or functional term, we use the induction hypothesis as was done in Lemma 6.54.

Finally, if s is a meta-variable application $Z(s_1,\ldots,s_n)$, then let $\gamma(Z) = \lambda x_1\ldots x_n.t$. We have:

$$
\begin{array}{ll}
& \mathsf{tag}_A(s)\gamma^{\mathsf{tag}} \\
= & Z(\mathsf{tag}_A(s_1),\ldots,\mathsf{tag}_A(s_n))\gamma^{\mathsf{tag}} \\
= & \mathsf{tag}_{\{x_1,\ldots,x_n\}}(t)[x_1 := \mathsf{tag}_A(s_1)\gamma^{\mathsf{tag}},\ldots,x_n := \mathsf{tag}_A(s_n)\gamma^{\mathsf{tag}}] \\
\Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} & \mathsf{tag}_{\{x_1,\ldots,x_n\}}(t)[x_1 := \mathsf{tag}_A(s_1\gamma),\ldots,x_n := \mathsf{tag}_A(s_n\gamma)] \\
& \text{(by the induction hypothesis)} \\
= & \mathsf{tag}_{\{\vec{x}\}\cup A}(t)[\vec{x} := \mathsf{tag}_A(\vec{s}\gamma)] \quad \text{(by Lemma 6.53)} \\
\Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} & \mathsf{tag}_A(t[\vec{x} := \vec{s}\gamma]) \quad \text{(by Lemma 6.55)} \\
= & \mathsf{tag}_A(s\gamma) \text{ as required.}
\end{array}
$$

□

We now have most of the preparations for Lemma 6.60, which expresses that a reduction in a term of a certain form l can be done by only reducing subterms headed by untagged ("green") symbols. Later on, we will use this to see that the reduction $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ in a dependency chain can be assumed to reduce only untagged symbols. To immediately combine the result with formative rules, we will need the following extension of the notion of a formative reduction.

Definition 6.58. For l a fixed meta-term, s a term and γ a substitution whose domain contains only meta-variables, we say that $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ by a *tagged formative l-reduction* if one of the following clauses holds:

1. $\mathsf{tag}(s) = l\gamma^{\mathsf{tag}}$ and l has the form $\lambda\vec{x}.Z(\vec{x})$;
2. $s = f(s_1,\ldots,s_n)\cdot s_{n+1}\cdots s_m$ and $l = f(l_1,\ldots,l_n)\cdot l_{n+1}\cdots l_m$ and each $\mathsf{tag}(s_i) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l_i\gamma^{\mathsf{tag}}$ by a tagged formative l_i-reduction;
3. $s = x\cdot s_1\cdots s_m$ and $l = x\cdot l_1\cdots l_m$ for a variable x and each $\mathsf{tag}(s_i) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l_i\gamma^{\mathsf{tag}}$ by a tagged formative l_i-reduction;
4. $s = \lambda x.s'$ and $l = \lambda x.l'$ and $\mathsf{tag}(s') \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l'\gamma^{\mathsf{tag}}$ by a tagged formative l'-reduction, and x does not occur in domain or range of γ;
5. $s = (\lambda x.t)\cdot q\cdot\vec{u}$, and $\mathsf{tag}(t[x := q]\cdot\vec{u}) \Rightarrow^*_{\mathcal{R}} l\gamma^{\mathsf{tag}}$ by a tagged formative l-reduction;
6. l is not a single meta-variable Z, and there are a rule $l' \Rightarrow r' \in \mathcal{R}^+$ and a substitution δ such that $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l'\delta^{\mathsf{tag}}$ by a tagged formative l'-reduction and $\mathsf{tag}(r'\delta) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma$ by a tagged formative l-reduction which does not use clause 6;
7. $l = \lambda x_1\ldots x_n.Z(\vec{x})$ and $s = \lambda x_1\ldots x_i.(\lambda y.q)\cdot u\cdot\vec{v}$ and $\mathsf{tag}(\lambda x_1\ldots x_i.q[x := u]\cdot\vec{v}) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ is a tagged formative l-reduction which only uses clauses 1 and maybe 7, and $i \leq n$.

This definition extends on Definition 6.40 and is somewhat technical. Let us first make some observations:

Lemma 6.59. *If* $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ *by a tagged formative* l*-reduction, then indeed* $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}} l\gamma^{\mathsf{tag}}$*. Moreover, we have* $s \Rightarrow^*_{\mathcal{R}} l\gamma$ *by a formative* l*-reduction, and the tagged formative* l*-reduction can be done using only rules in* $FR(l,\mathcal{R})^{\mathsf{tag}}$*.*

Proof. For the first claim; we use induction on the length of the reduction. It is obvious if $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ by case 1, in case 2 we note that $\mathsf{tag}(f(s_1,\dots,s_n)) = f(\mathsf{tag}(s_1),\dots,\mathsf{tag}(s_n))$, and in case 3 $\mathsf{tag}(s) = x \cdot \mathsf{tag}(s_1)\cdots\mathsf{tag}(s_m)$. In case 4 $\mathsf{tag}(s) = \lambda x.\mathsf{tag}_{\{x\}}(s')$ reduces to $\lambda x.\mathsf{tag}(s')$ by the untagging rules (according to Lemma 6.53), and in case 6 it holds because for $l' \Rightarrow r' \in \mathcal{R}^+$ we have $l'\chi \Rightarrow^+_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}(r')\chi$ for any substitution χ (by Lemma 6.56 if $l' \Rightarrow r'$ was obtained from the underlying rule using β-reduction, otherwise immediately), and $\mathsf{tag}(r')\delta^{\mathsf{tag}} \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}(r'\delta)$ using the untagging rules by Lemma 6.57. As for clauses 5 and 7, we have that $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}(t[x := q] \cdot \vec{u})$ and $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}(\lambda x_1 \dots x_i.q[x := u] \cdot \vec{v})$ respectively, by Lemma 6.56.

For the second claim, we also use induction on the definition of a tagged formative l-reduction, using that whenever a rule $l \Rightarrow r$ is used in the corresponding (untagged) formative l-reduction, then $l \Rightarrow \mathsf{tag}(r) \in FR(l,\mathcal{R})^{\mathsf{tag}}$ by Lemma 6.42. The only non-obvious part is when using clause 7, so $l = \lambda x_1 \dots x_n.Z(x_1,\dots,x_n)$ and $s = \lambda x_1 \dots x_i.t$ with $i \leq n$. In this case, the reduction $s \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} t$ uses only β-steps, and always at the highest headmost position where this is possible. Using clauses 1, 4 and 5 of Definition 6.40, also $s \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative l-reduction. □

We could now prove a result very similar to Lemma 6.41: if we can reduce to a term of a certain form l, then we can do this with a tagged formative reduction. But we have even more: a formative reduction can be *transformed* into a tagged formative reduction. This will become very relevant in the next chapter, when we consider transformations of dependency chains.

Lemma 6.60. *Let* $\mathcal{R}$ *be an abstraction-simple AFSM and* l *a pattern satisfying the constraints from Definition 6.48. If* $s \Rightarrow^*_{\mathcal{R}} l\gamma$ *by a formative* l*-reduction, then* $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ *by a tagged formative* l*-reduction.*

Proof. We prove this by induction on the definition of a formative l-reduction; we additionally show that the tagged formative reduction $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ does not use clause 6 if the original reduction does not use the corresponding clause.

If $s = l\gamma$ by clause 1 of Definition 6.40, then l can only be a single meta-variable, and $\mathsf{tag}(s) = l\gamma^{\mathsf{tag}}$ by clause 1 of Definition 6.58. If the reduction uses clause 2, $s = f(\vec{s})$, $l = f(\vec{l})$ and each $s_i \Rightarrow^*_{\mathcal{R}} l_i\gamma$, then by the induction hypothesis each $\mathsf{tag}(s_i) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l_i\gamma_i^{\mathsf{tag}}$, where γ_i is the restriction of γ to $FMV(l_i)$. We conclude that $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ by case 2 of Definition 6.58. Clause 3 is similar. If $s \Rightarrow_\beta s' \Rightarrow^*_{\mathcal{R}} l\gamma$ by clause 5, then $\mathsf{tag}(s') \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ by a tagged formative l-reduction by the induction hypothesis, so we are done with clause 5 of Definition 6.58.

Alternatively, suppose $s \Rightarrow_{\mathcal{R}} l\gamma$ by clause 6 of Definition 6.40. Then l is not a meta-variable and there are $l' \Rightarrow r' \in \mathcal{R}^+$ and a substitution δ such that $s \Rightarrow^*_{\mathcal{R}} l'\delta$ by a formative reduction and $r'\delta \Rightarrow^*_{\mathcal{R}} l\gamma$ by a formative reduction which does

not use clause 6. By the induction hypothesis, $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l'\delta^{\mathsf{tag}}$ by a tagged formative l'-reduction, and $\mathsf{tag}(r'\delta) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\delta^{\mathsf{tag}}$ by a tagged formative l-reduction without clause 6. Thus, $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ by a tagged formative l-reduction.

Only case 4 remains. Suppose first that $l = \lambda x_1 \dots x_n.Z(x_1, \dots, x_n)$. Then both s and l are abstractions; there are two possibilities:

- $s = \lambda x_1 \dots x_n.t$. Since the only formative l-reductions starting in an abstraction use clause 4, and the only formative $Z(\vec{x})$-reductions use clause 1 or 5, we can find t' such that $t \Rightarrow^*_\beta t'$ with headmost β-steps, and $\gamma = [Z := s']$, where $s' = \lambda\vec{x}.t']$. Thus, $\mathsf{tag}(s') = \lambda\vec{x}.\mathsf{tag}_{\{\vec{x}\}}(t') = l\gamma^{\mathsf{tag}}$, so we have a tagged formative l-reduction by clause 1 and possibly 7.
- $s = \lambda x_1 \dots x_i.q$, where $1 \leq i < n$ and q is not an abstraction. Therefore $n \geq 2$, so we are in the situation that $\mathcal{R}$ does not contain functional rules. Since q reduces to an abstraction, but is itself not an abstraction, it must head-reduce at some point. Since it cannot head-reduce with a rule step, $\mathtt{head}(q)$ is a β-redex. The shape of a formative reduction dictates that a β-step is taken immediately: $q = (\lambda y.u) \cdot v \cdot \vec{w} \Rightarrow_\beta u[x := v] \cdot \vec{w} \Rightarrow^*_{\mathcal{R}} \lambda x_{i+1} \dots x_n.Z(\vec{x})\gamma$ by a formative l-reduction. Continuing this argument, the reduction uses *only* $\Rightarrow_\beta$ steps. We conclude: $\mathsf{tag}(s) \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l\gamma^{\mathsf{tag}}$ by a tagged formative l-reduction, by clause 7.

Either way, the induction hypothesis holds.

Alternatively, suppose l does not have this form. Since l is an abstraction (we are still dealing with clause 4), by the restrictions from Definition 6.48 the only alternative is that $l = \lambda\vec{x}.t$ with $FMV(t) = \emptyset$, so γ and γ^{tag} are empty. Thus, $s = \lambda x.s'$ and $l = \lambda x.l'$ and $s' \Rightarrow^*_{\mathcal{R}} l'$ by a formative l'-reduction. By the induction hypothesis, $\mathsf{tag}(s') \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} l'$ by a tagged formative l'-reduction as well, so $\mathsf{tag}(s) = \lambda x.\mathsf{tag}_{\{x\}}(s') \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \lambda x.\mathsf{tag}(s)$ (by Lemma 6.53) $\Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \lambda x.l'$ (by the induction hypothesis), $= l$. □

6.5.4 Revised Dependency Pairs for Abstraction-simple AFSMs

Using Lemma 6.60 we will now define *tagged* dependency chains, and see that non-termination corresponds to the existence of a tagged dependency chain.

Definition 6.61. A *tagged dependency chain* is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i:

1. $\rho_i \in \mathsf{DP} \cup \{\mathtt{beta}\}$;
2. if $\rho_i = l_i \Rrightarrow p_i\ (A) \in \mathsf{DP}$ then there exists a substitution γ such that:
   a) $dom(\gamma) = FMV(l_i) \cup FMV(p_i) \cup FV(p_i)$;
   b) $s_i = l_i\gamma^{\mathsf{tag}}$;
   c) if p_i is an application or functional term, then $t_i = \mathsf{tag}(p_i\gamma)$;

   d) if $p_i = F(u_1, \ldots, u_n)$ with F a meta-variable, and $\gamma(F) = \lambda x_1 \ldots x_n.q$, then there exists a term v such that $q \trianglerighteq v$ and $\{\vec{x}\} \cap FV(v) \neq \emptyset$ and $t_i = \mathsf{tag}(v^\sharp[\vec{x} := \vec{u}\gamma])$, but v is not a variable;
   e) all variables in $dom(\gamma)$ are mapped to fresh variables;
   f) if $F : i \in A$ and $\gamma(F) = \lambda x_1 \ldots x_n.q$, then $x_i \in FV(q)$;

3. if $\rho_i = \mathtt{beta}$ then $s_i = \mathsf{tag}((\lambda x.q) \cdot u \cdot v_1 \cdots v_k)$ and either
   a) $k > 0$ and $t_i = \mathsf{tag}(q[x := u] \cdot v_1 \cdots v_k)$, or
   b) $k = 0$ and there exists a term v such that $q \trianglerighteq v$ and $x \in FV(v)$ and $t_i = \mathsf{tag}(v^\sharp[x := u])$, but v is not a variable;

4. $t_i \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}},in} s_{i+1}$;

A tagged dependency chain is *minimal* if moreover the strict subterms of all s_j, with tags removed, are terminating in $\Rightarrow_{\mathcal{R}}$, and is *formative* if the reduction $t_i \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}},in} s_{i+1}$ is a tagged formative ρ-reduction (that is, a tagged formative l-reduction if $\rho = l \Rrightarrow p$ and an empty reduction otherwise).

This definition is similar to the original definition of a dependency chain, but uses tags for s_i and t_i, and tagged rules in the $\Rightarrow_{\mathcal{R},in}$ reduction. By Lemma 6.60 we can transform an existing (minimal) formative dependency chain into a tagged (minimal) formative dependency chain.

In the following, we say that a set of dependency pairs $\mathcal{P}$ is abstraction-simple if all left-hand sides of dependency pairs in $\mathcal{P}$ satisfy the requirements of Definition 6.48. This is for example the case if $\mathcal{P} \subseteq \mathsf{DP}(\mathcal{R})$.

Theorem 6.62. *Let $\mathcal{R}$ be an abstraction-simple AFSM and $\mathcal{P}$ an abstraction-simple set of dependency pairs. Then there is a formative dependency chain over $(\mathcal{P}, \mathcal{R})$ if and only if there is a tagged formative dependency chain over $(\mathcal{P}, \mathcal{R})$, and the former is minimal if and only if the latter is.*

Proof. If we remove the tags from a tagged formative dependency chain we obtain a normal formative dependency chain, so one direction follows by Lemma 6.27. For the other direction, we take an existing formative dependency chain, replace t_i by $\mathsf{tag}(t_i)$, and if originally $\rho_i = l_i \Rrightarrow p_i$ and $s_i = l_i\gamma_i$, then we replace s_i by $l_i\gamma_i^{\mathsf{tag}}$. If $\rho_i = \mathtt{beta}$ then we replace s_i by $\mathsf{tag}(s_i)$. This gives a dependency chain with the required properties, minimal if and only if the original dependency chain is minimal. □

Thus, an abstraction-simple AFSM $\mathcal{R}$ is terminating if and only if there is a minimal tagged formative dependency chain over $\mathsf{DP}(\mathcal{R})$. More than that, an existing formative dependency chain can be transformed into a tagged formative dependency chain, and back.

Example 6.63. We consider once more the system from Example 6.26:

$$\mathtt{f}(\mathtt{0}) \;\Rightarrow\; \mathtt{g}(\lambda x.\mathtt{f}(x), \mathtt{a}) \qquad \mathtt{g}(F, \mathtt{b}) \;\Rightarrow\; F \cdot \mathtt{0} \qquad \mathtt{a} \;\Rightarrow\; \mathtt{b}$$

Noting that $\mathcal{R}^{\mathsf{tag}}$ consists of the rules

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{f}(\mathtt{0}) & \Rightarrow & \mathtt{g}(\lambda x.\mathtt{f}^-(x), \mathtt{a}) & \mathtt{g}(F, \mathtt{b}) & \Rightarrow & F \cdot \mathtt{0} \\ \mathtt{a} & \Rightarrow & \mathtt{b} & \mathtt{f}^-(x) & \Rightarrow & \mathtt{f}(x) \end{array}$$

as well as some other rules $h^-(\vec{x}) \Rightarrow h(\vec{x})$, we have the following tagged dependency chain:

$$\begin{array}{lll} (\; \mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{f}^-(x), \mathtt{a}), & f^\sharp(\mathtt{0}), & \mathtt{g}^\sharp(\lambda x.\mathtt{f}^-(x), \mathtt{a}) \;) \\ (\; \mathtt{g}^\sharp(F, \mathtt{b}) \Rrightarrow F \cdot \mathtt{0}, & \mathtt{g}(\lambda x.\mathtt{f}^-(x), \mathtt{b}), & (\lambda x.\mathtt{f}^-(x)) \cdot \mathtt{0} \;) \\ (\; \mathtt{beta}, & (\lambda x.\mathtt{f}^-(x)) \cdot \mathtt{0}, & \mathtt{f}^\sharp(\mathtt{0}) \;) \\ (\; \mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{f}^-(x), \mathtt{a}), & f^\sharp(\mathtt{0}), & \mathtt{g}^\sharp(\lambda x.\mathtt{f}^-(x), \mathtt{a}) \;) \\ \quad \ldots & & \end{array}$$

Here, the `beta` step uses case 3b with $v = \mathtt{f}(x)$.

By Theorem 6.62 we can prove termination of an abstraction-simple system by showing that it does not admit a minimal formative tagged dependency chain. As before, we can use a reduction triple, but we have a different subterm property:

Definition 6.64 (Tagged Subterm Property)**.** $\succeq$ has the *tagged subterm property* if: for all variables $x_1, \ldots, x_n$ and terms $s, q, t_1, \ldots, t_n$ such that $s \trianglerighteq q$ and $FV(q) \cap \{\vec{x}\} \neq \emptyset$ and q not a variable, there is a substitution γ on domain $FV(q) \setminus FV(s)$ such that: $\mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \succeq \mathsf{tag}(q^\sharp[\vec{x} := \vec{t}]\gamma)$

As we will see shortly, the tagged subterm property is an improvement over the limited subterm property because we do not have to take the subterms of untagged functional terms $f(\vec{s})$. We can adapt the proof of Theorem 6.31 to obtain the following result:

Theorem 6.65. *An abstraction-simple set $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ of dependency pairs is formative chain-free if $\mathcal{P}_2$ is formative chain-free and there is a reduction triple $(\succsim, \succeq, \succ)$ such that:*

- $l \succ \mathsf{tag}(p')$ *for all* $l \Rrightarrow p\ (A) \in \mathcal{P}_1$,
- $l \succeq \mathsf{tag}(p')$ *for all* $l \Rrightarrow p\ (A) \in \mathcal{P}_2$,
- $l \succsim \mathsf{tag}(r)$ *for all* $l \Rightarrow r \in FR(\mathcal{P}, \mathcal{R})$,
- $f^-(\vec{Z}) \succsim f(\vec{Z})$ *for all* $f^- \in \mathcal{F}$,
- $\mathcal{P}$ *is non-collapsing, or* $\succsim$ *satisfies the tagged subterm property.*

Here, p' is p with free variables replaced by arbitrary $\mathtt{c}^i_\sigma$ of suitable type.

Proof. This proof is derived in much the same way as Theorem 6.31, using that a minimal formative dependency chain can be transformed into a tagged minimal formative dependency chain.

Towards a contradiction, assume that $\mathcal{P}_2$ is formative chain-free while $\mathcal{P}$ is not. By Lemma 6.60 this means a minimal formative tagged dependency chain exists over $\mathcal{P}$ – say $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ – but not over $\mathcal{P}_2$, so this chain contains infinitely many $\rho_i \in \mathcal{P}_1$. As before, we assume that the variables in the dependency pairs do not freely occur in any of the s_i, t_i, and that if $\mathcal{P}$ is non-collapsing then all $\rho_i \in \mathcal{P}$. We will define for all i some substitution δ_i such that $s_i\delta_i^{\mathsf{tag}} \succ s_{i+1}\delta_{i+1}^{\mathsf{tag}}$, or $s_i\delta_i^{\mathsf{tag}} \succeq s_{i+1}\delta_{i+1}^{\mathsf{tag}}$ or $s_i\delta_i^{\mathsf{tag}} \succsim s_{i+1}\delta_{i+1}^{\mathsf{tag}}$ or $s_i\delta_i^{\mathsf{tag}} \succsim \cdot \succeq \cdot \succsim s_{i+1}\delta_{i+1}^{\mathsf{tag}}$; the strict inequality occurs whenever $\rho_i \in \mathcal{P}_1$. We distinguish three cases:

- If ρ_i is a non-collapsing dependency pair $l \Rrightarrow p\ (A)$, then $s_i = l\gamma^{\mathsf{tag}}$ and $t_i = \mathsf{tag}(p\gamma)$ for some substitution γ. Let χ be the substitution such that $p' = p\chi$.

  Then, if $\rho_i \in \mathcal{P}_1$ we have: $s_i\delta_i^{\mathsf{tag}} = l\gamma^{\mathsf{tag}}\delta_i^{\mathsf{tag}} \succ \mathsf{tag}(p\chi)\gamma^{\mathsf{tag}}\delta_i^{\mathsf{tag}} \succeq \mathsf{tag}(p\chi\gamma)\delta_i^{\mathsf{tag}}$ by Lemma 6.57, which is equal to $\mathsf{tag}(p\gamma\chi)\delta_i^{\mathsf{tag}}$ (swapping the substitutions as before), and by Lemma 6.54 this is equal to $\mathsf{tag}(p\gamma)\chi^{\mathsf{tag}}\delta_i^{\mathsf{tag}} = t_i\chi^{\mathsf{tag}}\delta_i^{\mathsf{tag}}$. If $\rho_i \in \mathcal{P}_2$ we have the same, but with $\succeq$ instead of $\succ$.

  We choose $\delta_{i+1} := \chi\delta_i$.

- If ρ_i is a collapsing dependency pair $l \Rrightarrow F(u_1, \ldots, u_m)\ (A)$, then $\mathcal{P}$ is collapsing, so $\succeq$ satisfies the tagged subterm property. There is a substitution γ such that $s_i = l\gamma^{\mathsf{tag}}$ and $\gamma(F) = \lambda x_1 \ldots x_m.q$, and there is a subterm v of q such that $\{\vec{x}\} \cap FV(v) \neq \emptyset$ and $t_i = \mathsf{tag}(v^\sharp[\vec{x} := \vec{u}\gamma])$, and v is not a variable. We can also write $p' = p\chi$ as in the previous case.

  If $\rho_i \in \mathcal{P}_1$ then $s_i\delta_i^{\mathsf{tag}} = l\gamma^{\mathsf{tag}}\delta_i^{\mathsf{tag}} \succ \mathsf{tag}(p\chi)\gamma^{\mathsf{tag}}\delta_i^{\mathsf{tag}} = \mathsf{tag}(p)\chi^{\mathsf{tag}}\gamma^{\mathsf{tag}}\delta_i^{\mathsf{tag}}$ by Lemma 6.54, $= \mathsf{tag}(p)\gamma^{\mathsf{tag}}\chi^{\mathsf{tag}}\delta_i^{\mathsf{tag}}$ (swapping the substitutions is safe as before), and this we can rewrite to $\mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{u})\gamma^{\mathsf{tag}}]\chi^{\mathsf{tag}}\delta_i^{\mathsf{tag}}$. Using Lemma 6.57, each $\mathsf{tag}(u_i)\gamma^{\mathsf{tag}} \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} \mathsf{tag}(u_i\gamma)$, so we have: $s_i\delta_i \succ \cdot \succeq \mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{u}\gamma)]\chi^{\mathsf{tag}}\delta_i^{\mathsf{tag}}$. By Lemma 6.54 and the observation that $FV(q) = \{\vec{x}\}$ this equals $\mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{u}\gamma\chi\delta_i)]$.

  Now the tagged subterm property gives us a substitution ζ such that: $\mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{u}\gamma\chi\delta_i)] \succeq \mathsf{tag}(v^\sharp[\vec{x} := \vec{u}\gamma\chi\delta_i]\zeta) = \mathsf{tag}(v^\sharp[\vec{x} := \vec{u}\gamma]\chi\delta_i\zeta)$. Since this is a substitution of variables, this term is equal to $\mathsf{tag}(v^\sharp[\vec{x} := \vec{q}\gamma])\chi^{\mathsf{tag}}\delta_i^{\mathsf{tag}}\zeta^{\mathsf{tag}}$ by Lemma 6.54. This is exactly $t_i(\chi\delta_i\zeta)^{\mathsf{tag}}$.

  We choose $\delta_{i+1} := \chi\delta_i\zeta$.

  Then, by transitivity of $\succeq$ and compatibility with $\succ$, we find: $s_i\delta_i^{\mathsf{tag}} \succ t_i\delta_{i+1}^{\mathsf{tag}}$.

  The case where $\rho_i \in \mathcal{P}_2$ is parallel, but has $\succeq$ in the place of $\succ$.

- If $\rho_j = \mathtt{beta}$, then $s_i\delta_i^{\mathsf{tag}} = \mathsf{tag}(((\lambda x.q)\cdot u\cdot\vec{v})\delta_i) = (\lambda x.\mathsf{tag}_{\{x\}}(q\delta_i))\cdot\mathsf{tag}(u\delta_i)\cdot \mathsf{tag}(\vec{v}\delta_i) \succsim \mathsf{tag}_{\{x\}}(q\delta_i)[x := \mathsf{tag}(u\delta_i)]\cdot\mathsf{tag}(\vec{v}\delta_i)$ because $\succsim$ contains beta.

If this was not a topmost step, so $t_i = \mathsf{tag}(q[x := u] \cdot \vec{v})$, then note that by Lemma 6.55 this term $\succsim \mathsf{tag}(q\delta_i[x := u\delta_i]) \cdot \mathsf{tag}(\vec{v}\delta_i) = \mathsf{tag}(t_i\delta_i) = \mathsf{tag}(t_i)\delta_i^{\mathsf{tag}}$. In this case, choose $\delta_{i+1} := \delta_i$.

If this was a topmost step, so $|\vec{v}| = 0$, then we can write $t_i = \mathsf{tag}(w^\sharp[x := u])$ for some subterm w of q which is not a variable, but which contains x. By the tagged subterm property, $\mathsf{tag}_{\{x\}}(q\delta_i)[x := \mathsf{tag}(u\delta_i)] \succeq \mathsf{tag}(w^\sharp\delta_i[x := u\delta_i]\zeta)$ for some substitution ζ, and since the domain of δ_i and ζ contains only variables, this is equal to $\mathsf{tag}(t_i)\delta_i^{\mathsf{tag}}\zeta^{\mathsf{tag}}$. In this case choose $\delta_{i+1} := \delta_i\zeta$. We have $s_i\delta_i^{\mathsf{tag}} \succeq \mathsf{tag}(t_i)\delta_{i+1}^{\mathsf{tag}}$

It is not hard to derive – using Lemma 6.59 to observe that only formative rules and untagging rules are used in the reduction $t_i \Rightarrow^*_{\mathcal{R}^{\mathsf{tag}}} s_i$ – that moreover each $t_i\delta_{i+1} \succsim s_{i+1}\delta_{i+1}$. Thus, we complete as before. □

6.5.5 Type Changing

Theorem 6.65 is comparable to Theorem 6.31, and as before, the result is likely not immediately usable due to typing problems. Moreover, it is not evident that the tagged subterm property is really weaker than the normal subterm property. Thus, to finish the work of this section, we should re-examine the results of Section 6.3.4. To start, let us reconsider the definition of $\rhd^{\mathcal{F}}$.

Definition 6.66 (Refinement of $\rhd^{\mathcal{F}}$)**.** Let S be a set of function symbols. $\rhd^S$ is the relation on base-type terms generated by the following clauses:

- $(\lambda x.s) \cdot t_0 \cdots t_n \rhd^S q$ if $s[x := t_0] \cdot t_1 \cdots t_n \trianglerighteq^S q$
- $f(s_1, \ldots, s_m) \cdot t_1 \cdots t_n \trianglerighteq^S q$ if $s_i \cdot \vec{c} \trianglerighteq^S q$ and $f \in S$ ← *here we differ from* $\rhd^{\mathcal{F}}$
- $s \cdot t_1 \cdots t_n \trianglerighteq^S q$ if $t_i \cdot \vec{c} \trianglerighteq^S q$ (s may have any form)

Here, $\trianglerighteq^S$ is the reflexive closure of $\rhd^S$. Note that our original definition of $\rhd^{\mathcal{F}}$ is just a special case of this definition. For abstraction-simple AFSMs, we can limit ourselves to $\rhd^{\mathcal{F}^-}$, shortly denoted $\rhd^-$. We obtain the following result:

Theorem 6.67. *An abstraction-simple set of dependency pairs $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ is formative chain-free if $\mathcal{P}_2$ is formative chain-free and there is a reduction pair $(\succsim, \succ)$ such that:*

1. $\tilde{l} \succ \overline{\mathsf{tag}(p)}$ *for* $l \Rrightarrow p\ (A) \in \mathcal{P}_1$*;*
2. $\tilde{l} \succsim \overline{\mathsf{tag}(p)}$ *for* $l \Rrightarrow p\ (A) \in \mathcal{P}_2$*;*
3. $l \succsim \mathsf{tag}(r)$ *for all* $l \Rightarrow r \in FR(\mathcal{P}, \mathcal{R})$*;*
4. $f^-(Z_1, \ldots, Z_n) \succsim f(Z_1, \ldots, Z_n)$ *for all* $f^- \in \mathcal{F}^-$*;*
5. *if $\mathcal{P}$ is collapsing, then $(\succsim, \succ)$ respects $\rhd^-$, and $f^-(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for all $f \in \mathcal{D}$.*

Here, $\tilde{l} = l \cdot Z_1 \cdots Z_n$ for fresh meta-variables such that $l \cdot \vec{Z}$ has base type, and $\overline{\mathsf{tag}(p)}$ may be any base-type meta-term of the form $(\mathsf{tag}(p)\chi) \cdot t_1 \cdots t_m$, where χ replaces the free variables in $\mathsf{tag}(p)$ by symbols c_i^σ, and the t_j are meta-terms (which may use the meta-variables $Z_1, \ldots, Z_n$).

Proof. Let $(\succsim, \succ)$ be a reduction pair satisfying the requirements in the theorem, and let $(\geq', \gtrsim', >')$ be the reduction triple generated by $(\succsim, \succ)$ as was defined in Section 6.5.5. This triple clearly satisfies the first four requirements of Theorem 6.65 (using the observation that $\mathsf{tag}(p\chi) = \mathsf{tag}(p)\chi$ if χ maps variables to symbols c_i^σ). For the last requirement, let $\mathcal{P}$ be collapsing. We must see that $\geq'$ satisfies the tagged subterm property.

That is, given terms $s, q, t_1, \ldots, t_n$ and variables $\{x_1, \ldots, x_n\}$ such that $s \trianglerighteq q$ and $FV(q) \cap \{\vec{x}\} \neq \emptyset$ and q is not a variable, and given terms $u_1, \ldots, u_m$ such that $s \cdot \vec{y}$ has base type, there must be a substitution γ on domain $FV(q) \setminus FV(s)$ and terms $v_1, \ldots, v_k$ such that:

$$\mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \gtrsim \mathsf{tag}(q^\sharp[\vec{x} := \vec{t}]\gamma) \cdot \vec{v}$$

We prove this statement by induction on $s \cdot \vec{u}$, ordered with $\rhd^-$ (a relation which it is not hard to prove well-founded, using for instance a computability argument).

Base Case: suppose $s = q \cdot w_1 \cdots w_i$.

Then $\mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} = \mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot w_1' \cdots w_k' \cdot u_1 \cdots u_m$, where each $w_i' = \mathsf{tag}_{\{\vec{x}\}}(w_i)[\vec{x} := \mathsf{tag}(\vec{t})]$. Let $\vec{v}$ denote the sequence $w_1', \ldots, w_k', u_1, \ldots, u_m$. Then we thus have: $\mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} = \mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{v}$.

Now, if q does not have the form $f(\vec{q})$ with $f \in \mathcal{D}$, then $q = q^\sharp$, so by Lemma 6.55 and the assumption that always $f^-(\vec{Z}) \gtrsim f(\vec{Z})$ we have: $\mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \gtrsim \mathsf{tag}(q^\sharp[\vec{x} := \vec{t}]) \cdot \vec{v}$ which suffices if we take for γ the empty substitution.

If q does have this form, then note that by assumption some x_i occurs in q, so $\mathsf{tag}_{\{\vec{x}\}}(q) = f^-(\mathsf{tag}_{\{\vec{x}\}}(\vec{q})) \gtrsim f^\sharp(\mathsf{tag}_{\{\vec{x}\}}(\vec{q}))$. Thus, $\mathsf{tag}_{\{\vec{x}\}}(q)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{v} \gtrsim f^\sharp(\mathsf{tag}_{\{\vec{x}\}}(\vec{q})[\vec{x} := \mathsf{tag}(\vec{t})]) \cdot \vec{v}$, and by Lemma 6.55 this term $\gtrsim f^\sharp(\mathsf{tag}(\vec{q}[\vec{x} := \vec{t}])) \cdot \vec{v} = \mathsf{tag}(q^\sharp[\vec{x} := \vec{t}]) \cdot \vec{v}$, which suffices by taking for γ the empty substitution.

Induction Case: We consider all ways where q is a subterm, and not just the head of s:

1. $s = (\lambda y.v) \cdot w_0 \cdot w_1 \cdots w_m$ and $v \trianglerighteq q$;
2. $s = f(v_1, \ldots, v_m) \cdot w_1 \cdots w_n$ and some $v_i \trianglerighteq q$.
3. $s = w \cdot v_1 \cdots v_n$ and one of the $v_i \trianglerighteq q$;

These are the only forms s can have. In very general terms, each of these cases is easy because $\succ \cup \gtrsim$ includes $\rhd^-$ (in case 2 we use that $FV(f(\vec{v}))$ contains x, so the tagging function replaces f by f^-). Precise derivations are given below.

In case 1,

$$\begin{array}{ll}
& \mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \\
\succsim & (\mathsf{tag}_{\{\vec{x},y\}}(v)[y := \mathsf{tag}_{\{\vec{x}\}}(w_0)] \cdot \mathsf{tag}_{\{\vec{x}\}}(\vec{w}))[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \\
& \text{(since } \succsim \text{ contains beta)} \\
\succsim & (\mathsf{tag}_{\{\vec{x}\}}(v[y := w_0]) \cdot \mathsf{tag}_{\{x\}}(\vec{w}))[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \text{ by Lemma 6.55} \\
= & \mathsf{tag}_{\{\vec{x}\}}(v[y := w_0] \cdot \vec{w})[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{q}
\end{array}$$

Here, $\vec{w}$ denotes $w_1, \ldots, w_m$, so it does not include w_0.

Since $v[y := w_0] \trianglerighteq q[y := w_0]$ we can use the induction hypothesis to find $\gamma', \vec{v'}$ such that this last term $(\succ \cup \succsim)^* \ \mathsf{tag}(q[y := w_0]^\sharp \gamma'[\vec{x} := \vec{t}]) \cdot \vec{v'}$. This proves the lemma for $\gamma := [y := w_0]\gamma'$ because $q[y := w_0]^\sharp = q^\sharp[y := w_0]$ when q is not a variable.

In case 2,

$$\begin{array}{ll}
& \mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \\
= & f^-(\mathsf{tag}_{\{\vec{x}\}}(\vec{v})[\vec{x} := \mathsf{tag}(\vec{t})]) \cdot (\mathsf{tag}_{\{\vec{x}\}}(\vec{w})[\vec{x} := \mathsf{tag}(\vec{t})]) \\
& \text{because } FV(v_i) \cap \{\vec{x}\} \neq \emptyset \\
(\succ \cup \succsim)^* & \mathsf{tag}_{\{\vec{x}\}}(v_i)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{c} \text{ because } (\succsim, \succ) \text{ respects } \rhd^- \\
(\succ \cup \succsim)^* & \mathsf{tag}(q^\sharp[\vec{x} := t]\gamma) \cdot \vec{v'} \text{ for some } \vec{v'} \text{ and } \gamma \\
& \text{by the induction hypothesis}
\end{array}$$

Finally, case 3.

$$\begin{array}{ll}
& \mathsf{tag}_{\{\vec{x}\}}(s)[\vec{x} := \mathsf{tag}(\vec{t})] \cdot \vec{u} \\
= & (\mathsf{tag}_{\{\vec{x}\}}(w)\delta) \cdot (\mathsf{tag}_{\{\vec{x}\}}(v_1)\delta) \cdots (\mathsf{tag}_{\{\vec{x}\}}(v_n)\delta) \cdot (\vec{u}\delta) \\
& \text{where } \delta := [\vec{x} := \mathsf{tag}(\vec{t})] \\
(\succ \cup \succsim)^* & \mathsf{tag}_{\{\vec{x}\}}(v_i)\delta \cdot \vec{c} \text{ because } (\succsim, \succ) \text{ respects } \rhd^- \\
(\succ \cup \succsim)^* & \mathsf{tag}(q^\sharp[\vec{x} := t]\gamma) \cdot \vec{v'} \text{ for some } \vec{v'} \text{ and } \gamma \\
& \text{by the induction hypothesis}
\end{array}$$

□

Example 6.68. Recall the rules of `eval` from Example 6.14, whose dependency pairs were calculated in Example 6.22 and its formative rules in Example 6.46. This system is already β-saturated, and is abstraction-simple. To prove termination we must see that $\mathsf{DP}(\mathcal{R})$ is formative chain-free, and by Theorem 6.67 it suffices to find a reduction pair which respects $\rhd^-$ and orients (choosing $\mathcal{P}_1 = \mathsf{DP}(\mathcal{R})$ and $\mathcal{P}_2 = \emptyset$):

$$\begin{array}{rcl}
\mathtt{dom}^\sharp(\mathtt{s}(X), \mathtt{s}(Y), \mathtt{s}(Z)) & \succ & \mathtt{dom}^\sharp(X, Y, Z) \\
\mathtt{dom}^\sharp(\mathtt{0}, \mathtt{s}(Y), \mathtt{s}(Y)) & \succ & \mathtt{dom}^\sharp(\mathtt{0}, Y, Z) \\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x), X, Y), Z) & \succ & \mathtt{dom}^\sharp(X, Y, Z) \\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x), X, Y), Z) & \succ & F(\mathtt{dom}(X, Y, Z)) \\
\mathtt{dom}(X, Y, \mathtt{0}) & \succsim & X \\
\mathtt{dom}(\mathtt{0}, \mathtt{0}, Z) & \succsim & \mathtt{0} \\
\mathtt{eval}(\mathtt{fun}(\lambda x.F(x), X, Y), Z) & \succsim & F(\mathtt{dom}(X, Y, Z)) \\
f^-(\vec{Z}) & \succsim & f(\vec{Z}) \quad \text{for all } f \in \mathcal{F} \\
f^-(\vec{Z}) & \succsim & f^\sharp(\vec{Z}) \quad \text{for all } f \in \mathcal{D}
\end{array}$$

As in Example 6.37, we consider a polynomial interpretation. Since the f^- do not occur in the right-hand side of any requirement (in the original system, there are no abstractions in the right-hand side), the last two sets of constraints are easy. They are satisfied with $\mathcal{J}(f^-) = \boldsymbol{\lambda}\vec{x}.\mathcal{J}(f)(\vec{x}) + \mathcal{J}(f^\sharp)(\vec{x}) + x_1 + \ldots + x_n$. As in Example 6.37, we choose $\mathcal{J}(\mathtt{c}_\sigma) = 0_\sigma$ and $\mathcal{J}(@^\tau) = \boldsymbol{\lambda} f n \vec{m}.\ \max_{\tau\to\mathtt{o}\to\tau}(f(n,\vec{m}), n(\vec{0}))$, which together with the choice for $\mathcal{J}(f^-)$ guarantees that $(\succsim, \succ)$ respects $\rhd^-$. Having fixed these values, we can choose the remaining $\mathcal{J}(f)$ as arbitrary (so not necessarily strongly monotonic!) weakly monotonic functionals. All requirements are oriented with the following interpretation:

$$\begin{array}{rclcrcl}
\mathcal{J}(\mathtt{0}) & = & 0 & & \mathcal{J}(\mathtt{dom}) & = & \boldsymbol{\lambda} nmk.n+m \\
\mathcal{J}(\mathtt{s}) & = & \boldsymbol{\lambda} n.n+1 & & \mathcal{J}(\mathtt{dom}^\sharp) & = & \boldsymbol{\lambda} nmk.m \\
\mathcal{J}(\mathtt{fun}) & = & \boldsymbol{\lambda} f nm.f(n+m)+n+m & & \mathcal{J}(\mathtt{eval}) & = & \boldsymbol{\lambda} nm.n \\
 & & & & \mathcal{J}(\mathtt{eval}^\sharp) & = & \boldsymbol{\lambda} nm.n+1
\end{array}$$

$$\begin{array}{rcl}
Y+1 & > & Y \\
Y+1 & > & Y \\
F(X+Y)+X+Y+1 & > & Y \\
F(X+Y)+X+Y+1 & > & F(X+Y) \\
X+Y & \geq & X \\
0 & \geq & 0 \\
F(X+Y)+X+Y & \geq & F(X+Y)
\end{array}$$

Thus, the system is indeed terminating.

Theorem 6.67 is a real improvement over Theorem 6.45 because (A) the requirement that $\rhd^-$ is included in $\succ \cup \succsim$ is significantly weaker than the requirement for $\rhd^{\mathcal{F}}$ to be included, and (B) the requirement that $f(\vec{Z}) \succsim f^\sharp(\vec{Z})$ was replaced by a requirement that $f^-(\vec{Z}) \succsim f^\sharp(\vec{Z})$, which removes the direct relationship between a defined symbol and its marked version.

Thus, we can use a weak reduction pair rather than a strong reduction pair, and deal with systems which were previously beyond reach, as demonstrated by Example 6.68. In the next section we will see precisely how to use weakly monotonic algebras and also path orderings in such a way that $\rhd^-$ is respected. Moreover, we will generalise the notion of argument functions from Chapter 5.6, to be used with any strong reduction pair (which satisfies a number of constraints).

6.6 Finding a Reduction Pair

Both in the basic dependency pair approach and its improvements, we use a weak reduction pair to prove that a given set of dependency pairs is (formative) chain-free. In Chapters 4 and 5 we have seen two different ways to create a weak reduction pair: either by using weakly monotonic algebras without the strong monotonicity constraint (Theorem 4.10), or by using path orderings with an argument function which need not be argument preserving (Theorem 5.39).

For a set of non-collapsing dependency pairs, these results suffice. But for collapsing dependency pairs, both the basic (formative) dependency pair approach (Theorems 6.36 and 6.45) and the tagged dependency pair approach (Theorem 6.67) pose additional constraints on the reduction pair that is used. In this section, we will see some systematic ways to satisfy these additional constraints.

First, let us put the results of Theorems 6.45 and 6.67 in a different form.

Definition 6.69. Let $\mathcal{R}$ be a set of rules, and $\mathcal{P}_1$ and $\mathcal{P}_2$ sets of triples $l \Rrightarrow p\ (A)$, where l is a pattern of the form $f(\vec{s})\cdot\vec{t}$, r a meta-term and A a set of meta-variable conditions.

A *standard reduction pair* for $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$ is a weak reduction pair $(\succsim, \succ)$ for which we can find $\tilde{l}, \overline{p}$ for all $l \Rrightarrow p\ (A) \in \mathcal{P}_1 \cup \mathcal{P}_2$, such that the following inequalities are satisfied:

- $\tilde{l} \succ \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_1$;
- $\tilde{l} \succsim \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_2$;
- $l \succsim r$ for $l \Rightarrow r \in \mathcal{R}$;
- all pairs in $\mathcal{P}_1 \cup \mathcal{P}_2$ are non-collapsing, or:
  - $f(s_1, \ldots, s_m) \cdot t_1 \cdots t_n \succsim s_i \cdot \vec{c}$ for all $f \in \mathcal{F}$ if both sides have base type (note that $f \in \mathcal{F}$, so marked symbols are not included);
  - $s \cdot t_1 \cdots t_n \succsim t_i \cdot \vec{c}$ if both sides have base type;
  - $f(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for all $f \in \mathcal{D}$.

Each $\tilde{l} \Rrightarrow \overline{p}$ must be a pair of base-type meta-terms, such that $\tilde{l} = l \cdot Z_1 \cdots Z_n$ for fresh meta-variables $Z_1, \ldots, Z_n$, and $\overline{p}$ has the form $(p\chi) \cdot s_1 \cdots s_m$ where the s_i are arbitrary meta-terms, and χ is a substitution δ which maps free variables in p to symbols c_i^σ.

Definition 6.70. A *tagged reduction pair* for $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$ is a weak reduction pair $(\succsim, \succ)$ for which we can find $\tilde{l}, \overline{p}$ for all $l \Rrightarrow p\ (A) \in \mathcal{P}_1 \cup \mathcal{P}_2$, such that the following inequalities are satisfied:

- $\tilde{l} \succ \overline{\mathsf{tag}(p)}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_1$;
- $\tilde{l} \succsim \overline{\mathsf{tag}(p)}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}_2$;
- $l \succsim \mathsf{tag}(r)$ for $l \Rightarrow r \in \mathcal{R}$;
- $f(s_1, \ldots, s_m) \cdot t_1 \cdots t_n \succsim s_i \cdot \vec{c}$ for all $f \in \mathcal{F}^-$ if both sides have base type;
- $s \cdot t_1 \cdots t_n \succsim t_i \cdot \vec{c}$ if both sides have base type;
- $f^-(\vec{Z}) \succsim f(\vec{Z})$ for all $f \in \mathcal{F}$ and $f^-(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for all $f \in \mathcal{D}$.

Each $\tilde{l} \Rrightarrow \overline{\mathsf{tag}(p)}$ must be a pair of base-type meta-terms, such that $\tilde{l} = l \cdot Z_1 \cdots Z_n$ for fresh meta-variables $Z_1, \ldots, Z_n$ and $\overline{\mathsf{tag}(p)}$ has the form $(\mathsf{tag}(p)\chi) \cdot s_1 \cdots s_m$ where the s_i are arbitrary meta-terms, and χ is a substitution which maps free variables in p to symbols c_i^σ.

With these two definitions, we can simplify the phrasing both for the basic and for the tagged result, and combine them in one theorem.

Theorem 6.71 (Reduction Pairs)**.** *Let $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$ be a set of dependency pairs for the system $(\mathcal{F}, \mathcal{R})$.*

The set $\mathcal{P}$ is formative chain-free if $\mathcal{P}_2$ is formative chain-free, and a standard reduction pair exists for $(\mathcal{P}_1, \mathcal{P}_2, FR(\mathcal{P}, \mathcal{R}))$.

If $\mathcal{P}$ and $\mathcal{R}$ are abstraction-simple, then $\mathcal{P}$ is formative chain-free if $\mathcal{P}_2$ is formative chain-free, and a tagged reduction pair exists for $(\mathcal{P}_1, \mathcal{P}_2, FR(\mathcal{P}, \mathcal{R}))$.

Proof. This is a consequence of the combination of Theorems 6.45 and 6.67. □

Note that in the definition of a tagged reduction pair, there is no special case for non-collapsing $\mathcal{P}_1 \cup \mathcal{P}_2$. This is because in a non-collapsing system, there are no subterm steps anyway, so we might as well use the standard reduction pair and not have to deal with tagged constraints.

To find a suitable reduction pair, if the set $\mathcal{P}$ under consideration is non-collapsing, is simply a matter of choosing the right interpretation function (in the case of weakly monotonic algebras), or the right precedence, status function and argument function (in the case of path orderings).

In the case of a collapsing set, $\mathcal{P}$, we have potentially infinitely many constraints. Apart from the constraints of the form $l \succ r$ and $l \succsim r$ for dependency pairs and rules, we also have:

A. constraints of the form $f(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for $f \in \mathcal{D}$ in the case of a standard reduction pair, or otherwise $f^-(\vec{Z}) \succsim f^\sharp(\vec{Z})$ for $f \in \mathcal{D}$ and $f^-(\vec{Z}) \succsim f(\vec{Z})$ for $f \in \mathcal{F}$;

B. base-type constraints $f(s_1, \ldots, s_m) \cdot \vec{t} \succsim s_i \cdot \vec{\mathsf{c}}$ for all f either in $\mathcal{F}$ (in the case of a standard reduction pair), or in $\mathcal{F}^-$ (in the case of a tagged reduction pair);

C. base-type constraints of the form $s \cdot t_1 \cdots t_n \succsim t_i \cdot \vec{\mathsf{c}}$.

The aim, now, is as follows: to find a (finite!) number of constraints which an algebra interpretation or path ordering should satisfy, which immediately give us A–C. Ideally, these constraints should be as few and simple as possible.

Sections 6.6.1 and 6.6.2 will provide standard choices and constraints when using algebra interpretations and path orderings respectively. In Section 6.6.3 we will generalise the argument functions from Chapter 5.6 to be used with an arbitrary strong reduction pair, and discuss the requirements needed for the resulting pair to satisfy A–C.

6.6.1 Weakly Monotonic Algebras

In Examples 6.37 and 6.68, we used weakly monotonic algebras to orient the dependency pair constraints. The theory for this is supplied by Theorem 4.10, which provides a weak reduction pair; the fact that strong monotonicity is not required gives us more freedom than we have with rule removal. To some extent this was already used in these examples: the interpretation for the application symbols, $\mathcal{J}(@^\tau) = \boldsymbol{\lambda} f n \vec{m}.\, \max(f(n, \vec{m}), n(\vec{0}))$, is not strongly monotonic.

As in Chapter 4, we will focus on interpretations in $\mathbb{N}$. To start, let us fix a number of requirements for the interpretation functions.

Definition 6.72 (Default Requirements)**.** An interpretation function $\mathcal{J}$ satisfies the *default requirements* for $\mathcal{F}$ and $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{F})$ if the following constraints are satisfied:

1. $\mathcal{J}(\mathsf{c}_i^\sigma) = 0_\sigma$ for all types σ (that is, the constant function in $\mathcal{WM}_\sigma$ that returns 0 for all inputs);
2. in the case of a tagged reduction pair, for all $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau_1 \to \ldots \to \tau_m \to \iota \in \mathcal{F}$ such that f^- does not occur in $\mathcal{P}_1,\ \mathcal{P}_2$ or $\mathcal{R}$:
   a) $\mathcal{J}(f^-) = \boldsymbol{\lambda} \vec{x}\vec{y}.\, \max(\mathcal{J}(f)(\vec{x}, \vec{y}), x_1(\vec{0}), \ldots, x_n(\vec{0}))$ if $f \notin \mathcal{D}$;
   b) $\mathcal{J}(f^-) = \boldsymbol{\lambda} \vec{x}\vec{y}.\, \max(\mathcal{J}(f)(\vec{x}, \vec{y}), \mathcal{J}(f^\sharp)(\vec{x}, \vec{y}), x_1(\vec{0}), \ldots, x_n(\vec{0}))$ if $f \in \mathcal{D}$;
3. for all functional types σ:
   a) $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} f x.\, f(x)$ for all σ (as required by Theorem 4.10 to orient the β-rule);
   b) $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} f x \vec{y}.x(\vec{0})$ for all σ;
   c) $\mathcal{J}(@^\sigma)(f, 0_{\sigma_1}) = f(0_{\sigma_1})$.

Choosing the interpretations for the symbols c_i^σ as small as possible seems reasonable, as they only appear on the right-hand side of constraints. The interpretations for unused symbols are not very important; especially in hand-written termination proofs, we would ideally ignore such symbols. By fixing their interpretation immediately, we can do just that. As for the interpretation of $@^\sigma$, the requirements are satisfied for example by the choice $\boldsymbol{\lambda} f x \vec{y}. f(x, \vec{y}) + x(\vec{0})$, which appeared in Example 4.11. Alternatively, we might choose $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} f x \vec{y}.\, \max(f(x, \vec{y}), x(\vec{0}))$, as in Examples 6.37 and 6.68.[3]

[3] Recall that $\mathcal{J}(@^\sigma)$ must be a function in $\mathcal{WM}_{\sigma\to\sigma}$. Thus, if for example $\sigma = \mathsf{o} \to \mathsf{o} \to \mathsf{o}$, then the interpretation function is a weakly monotonic function in $\mathcal{WM}_{\mathsf{o}\to\mathsf{o}\to\mathsf{o}} \to \mathcal{WM}_{\mathsf{o}} \to \mathcal{WM}_{\mathsf{o}} \to \mathcal{WM}_{\mathsf{o}}$, which takes three arguments. With the interpretation of application as addition, we have e.g. $[\![s \cdot t \cdot q]\!] = \mathcal{J}(@^{\mathsf{o}\to\mathsf{o}}([\![s \cdot t]\!], [\![q]\!])) = [\![s \cdot t]\!]([\![q]\!]) + [\![q]\!] = \mathcal{J}(@^{\mathsf{o}\to\mathsf{o}\to\mathsf{o}})([\![s]\!], [\![t]\!])([\![q]\!]) + [\![q]\!] = (\boldsymbol{\lambda} y.[\![s]\!]([\![t]\!], y) + [\![t]\!])([\![q]\!]) + [\![q]\!] = [\![s]\!]([\![t]\!], [\![q]\!]) + [\![t]\!] + [\![q]\!]$.

Using the default requirements, we can make two observations which will prove very useful. First, for any term $s \cdot t_1 \cdots t_n$, we can see that $[\![s \cdot \vec{t}]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![s]\!]_{\mathcal{J},\alpha}([\![t_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![t_n]\!]_{\mathcal{J},\alpha})$. This holds by induction on n, using clause 3a.

In addition, for any term $s : \sigma_1 \to \ldots \to \sigma_n \to \iota$ we have that $[\![s \cdot \mathsf{c}_1^{\sigma_1} \cdots \mathsf{c}_j^{\sigma_j}]\!]_{\mathcal{J},\alpha} = \boldsymbol{\lambda} x_{j+1} \ldots x_n.[\![s]\!]_{\mathcal{J},\alpha}(0_{\sigma_1}, \ldots, 0_{\sigma_j}, x_{j+1}, \ldots, x_n)$. This holds by induction on j, and clauses 1 and 3c. Consequently, $[\![s \cdot \vec{\mathsf{c}}]\!]_{\mathcal{J},\alpha} = [\![s]\!]_{\mathcal{J},\alpha}(\vec{0})$.

Constraint A. Here, let us distinguish between the cases for a tagged reduction pair and a standard reduction pair.

When creating a *standard reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$, we simply add the following constraints:

$$\text{For all } f \in \mathcal{D}\text{: } \mathcal{J}(f) \sqsupseteq \mathcal{J}(f^\sharp).$$

Since $\mathcal{D}$ is finite if $\mathcal{R}$ is finite, this only adds finitely many constraints, unless we already had infinitely many to deal with. Obviously, choosing $\mathcal{J}$ like this, all constraints of the form $f(\vec{Z}) \gtrsim f^\sharp(\vec{Z})$ are satisfied.

When creating a *tagged reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$, we add the following constraints:

$$\begin{array}{c}\text{For all } f^- \text{ which actually occur in } \mathcal{P}_1,\ \mathcal{P}_2 \text{ or } \mathcal{R}\text{:}\\ \mathcal{J}(f^-) \sqsupseteq \mathcal{J}(f);\\ \text{if } f \in \mathcal{D}\text{: } \mathcal{J}(f^-) \sqsupseteq \mathcal{J}(f^\sharp).\end{array}$$

Again, this only adds finitely many constraints if $\mathcal{R}$ is finite, even if $\mathcal{F}$ is infinite. With these constraints, certainly $f^-(\vec{Z}) \gtrsim f(\vec{Z})$ if f^- occurs in $\mathcal{P}_1$, $\mathcal{P}_2$ or $\mathcal{R}$, and similarly $f^-(\vec{Z}) \gtrsim f^\sharp(\vec{Z})$ (if necessary). For the unused symbols, the constraints also hold, provided $\mathcal{J}$ satisfies the default requirements: $\mathcal{J}(f^-) \sqsupseteq \boldsymbol{\lambda}\vec{x}\vec{y}.\mathcal{J}(f)(\vec{x},\vec{y}) = \mathcal{J}(f)$ by the nature of the $\max$ function (and similar for $f^\sharp$).

Constraint B. Again, we distinguish depending on the setting.

When creating a *standard reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$, we add the following constraints:

$$\mathcal{J}(f) \sqsupseteq \boldsymbol{\lambda} x_1 \ldots x_n \vec{y}.x_i(\vec{0}) \text{ for all } f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}$$

Alternatively, we can choose $\mathcal{J}$ in such a way that this constraint is systematically satisfied. For example, a polynomial interpretation with the restrictions of Theorem 4.20 has this property.

If $\mathcal{J}$ satisfies the default requirements, and additionally these constraints, then $[\![f(s_1, \ldots, s_n) \cdot t_1 \cdots t_m]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![f(s_1, \ldots, s_n)]\!]_{\mathcal{J},\alpha}([\![t_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![t_m]\!]_{\mathcal{J},\alpha}) = \mathcal{J}([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![t_m]\!]_{\mathcal{J},\alpha}) \sqsupseteq s_i(\vec{0}) = [\![s_i \cdot \vec{\mathsf{c}}]\!]_{\mathcal{J},\alpha}$, using the (in-)equalities we previously derived.

When creating a *tagged reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$, we add the following constraints:

$$\mathcal{J}(f^-) \sqsupseteq \boldsymbol{\lambda} x_1 \ldots x_n \vec{y}.x_i(\vec{0})$$
for all $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}$ such that f^- occurs in $\mathcal{P}_1,\ \mathcal{P}_2$ or $\mathcal{R}$

This choice works for the same reasons as before, and is also satisfied by the fixed interpretation for the symbols f^- which do not occur in $\mathcal{P}_1,\ \mathcal{P}_2$ or $\mathcal{R}$.

Constraint C. The last constraint is directly satisfied with any interpretation function which satisfies the default requirements:

$[\![s \cdot t_1 \cdots t_n]\!]_{\mathcal{J},\alpha}$
$\sqsupseteq [\![s \cdot t_1 \cdots t_i]\!]_{\mathcal{J},\alpha}([\![t_{i+1}]\!]_{\mathcal{J},\alpha}, \ldots, [\![t_n]\!]_{\mathcal{J},\alpha})$ as we saw before
$= \mathcal{J}(@^\sigma)([\![s \cdot t_1 \cdots t_{i+1}]\!]_{\mathcal{J},\alpha}, [\![t_i]\!]_{\mathcal{J},\alpha})(\ldots)$ for some type σ
$= \max([\![s \cdot t_1 \cdots t_{i+1}]\!]_{\mathcal{J},\alpha}(\ldots), [\![t_i]\!]_{\mathcal{J},\alpha}(\vec{0}))$ by clause 3b
$\sqsupseteq [\![t_i]\!]_{\mathcal{J},\alpha}(\vec{0})$
$= [\![t_i \cdot \vec{0}]\!]_{\mathcal{J},\alpha}$ as we saw before

Thus, using weakly monotonic algebra interpretations in $\mathbb{N}$ we can find a weak reduction pair that satisfies all the requirements for Theorem 6.71, simply by making sure that the interpretation function satisfies the default requirements and including a few additional constraints (only finitely many, provided $\mathcal{R}$ is finite). Moreover, we can forget about unused tagged symbols f^-, as their requirements are automatically satisfied!

Since we no longer have the requirement that $\mathcal{J}$ is strongly monotonic, we have a far greater freedom in the choice of interpretations. We saw this used in Example 6.68, where a polynomial was used that was not strongly monotonic. But also, we can go beyond polynomials. One example is the $\max_\sigma$ function which we used for interpreting application. Or we could use repeated function application, such as $\boldsymbol{\lambda} f n m.\, \max(m, f^n(m))$ (a case analysis whether $m \geq f(m)$ or $f(m) \geq m$ shows that this function is indeed weakly monotonic).

6.6.2 Path Orderings

Next, let us consider the (recursive or iterative) path ordering. The definition of Chapter 5 natively includes argument functions, an extension of the argument filterings discussed in Section 6.2.4, and thus provides a weak reduction pair. To satisfy constraints A–C, we consider the reduction pair generated by Theorem 5.39 using an argument function π, a precedence $\blacktriangleright$ with a well-founded strict part, and a status function $stat$ with the following properties:

1. $\pi(\mathsf{c}_i^\sigma) = \bot_\sigma$ for all i, σ;
2. $@^{\sigma\to\tau} \blacktriangleright @^\sigma, @^\tau$ for all types σ, τ (or similar for $@^\rho_{swap}$);
3. in the case of a standard reduction pair, let $S := \mathcal{F}$, and in the case of a tagged reduction pair, let S consist of those symbols f^- which actually occur in the rules; then for all $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in S$: $\pi(f)$ has the form $\lambda x_1 \ldots x_n.g_f(\vec{s})$ with $\{\vec{x}\} \subseteq \{\vec{s}\}$ and for all $i \in \{1, \ldots, n\}$ $g_f \blacktriangleright @^{\sigma_i}$;

4. in the case of a tagged reduction pair, for all f^- which do not occur in $\mathcal{P}_1,\ \mathcal{P}_2$ or $\mathcal{R}$, if $\pi(f) = \lambda x_1 \dots x_n.s$ and, if f is a defined symbol, $\pi(f^\sharp) = \lambda x_1 \dots x_n.t$:

   a) fix $\pi(f^-) = \lambda x_1 \dots x_n.\mathtt{group}_f(x_1, \dots, x_n, s)$ if $f \notin \mathcal{D}$;

   b) fix $\pi(f^-) = \lambda x_1 \dots x_n.\mathtt{group}_f(x_1, \dots, x_n, s, t)$ if $f \in \mathcal{D}$.

   Here, $\mathtt{group}_f$ is some otherwise unused symbol, such that $\mathtt{group}_f \blacktriangleright @^\sigma$ for all types σ.

By fixing the argument function used for the unused symbols f^-, we guarantee that all constraints involving them are satisfied, so the termination prover (which may be a person or an automatic tool) can essentially ignore these symbols.

Constraint A. We simply add the following constraints:

$$\begin{gathered}\text{For all } f \in \mathcal{D}\text{: } f(Z_1, \dots, Z_n) \gtrsim f^\sharp(Z_1, \dots, Z_n).\\ \text{For all } f \in \mathcal{D} \text{ such that } f^- \text{ occurs in } \mathcal{P}_1,\ \mathcal{P}_2 \text{ or } \mathcal{R}\text{:}\\ f^-(Z_1, \dots, Z_n) \gtrsim f^\sharp(Z_1, \dots, Z_n).\\ \text{For all } f \in \mathcal{F} \text{ such that } f^- \text{ occurs in } \mathcal{P}_1,\ \mathcal{P}_2 \text{ or } \mathcal{R}\text{:}\\ f^-(Z_1, \dots, Z_n) \gtrsim f(Z_1, \dots, Z_n).\end{gathered}$$

These are only finitely many constraints if $\mathcal{R}$ is finite. Depending on the choice of π these constraints can also be guaranteed by a restriction on $\blacktriangleright$, for example choosing $f^- \blacktriangleright f, f^\sharp$ if the argument function at most permutes arguments. It is obvious, using the (Select) clause, that the constraints are also satisfied for those symbols f^- which do not occur in any of $\mathcal{P}_1,\ \mathcal{P}_2$ or $\mathcal{R}$.

Constraint B. We must see that $f(s_1, \dots, s_n) \cdot t_1 \cdots t_m \gtrsim s_i \cdot \vec{c}$ for a function f such that $\pi(f)$ has the form $\lambda x_1 \dots x_n.g_f(\vec{q})$ with $x_i \in \{\vec{q}\}$ and $g_f \blacktriangleright @^{\sigma_i}$, which is the type of s_i (note that these constraints are satisfied by definition if $f \in S$, but are also satisfied by the fixed choice of $\pi(f^-)$ for unused symbols f^-).

Using the definition of Theorem 5.39, we first observe: if $\overline{\pi}(\mu(u)) \succeq_\star g^\star(\vec{v})$, then for all w we have: $\overline{\pi}(\mu(u \cdot w)) = @^\sigma(\overline{\pi}(\mu(u)), \overline{\pi}(\mu(w))) \succeq_\star g^\star(\overline{\pi}(\mu(\vec{v})), \overline{\pi}(\mu(w)))$, or similar with $@^\sigma_{swap}$. Thus, writing $\vec{t'}$ for $\overline{\pi}(\mu(t_1)), \dots, \overline{\pi}(\mu(t_m))$, we have:

$$\overline{\pi}(\mu(f(\vec{s}) \cdot \vec{t})) \gtrsim g^\star_f(\vec{q}, \vec{t'}) \text{ with } \overline{\pi}(\mu(s_i)) \in \{\vec{q}\}$$

Let $\sigma_i = \tau_1 \to \dots \to \tau_m \to \kappa$. Then $\overline{\pi}(\mu(s_i \cdot \vec{c})) = @^{\tau_m \to \kappa}(\dots @^{\sigma_i}(s_i, \bot_{\tau_1}) \dots \bot_{\tau_m})$. By clause 2, $g_f \blacktriangleright$ each instance of $@$ used here, and therefore the (Copy) rule provides that $g^\star_f(\vec{q}, \vec{t}) \succeq_\star \overline{\pi}(\mu(s \cdot \vec{c}))$.

Constraint C. Finally, to see that always $s \cdot t_1 \cdots t_n \gtrsim t_i \cdot \vec{c}$, we might observe, similar to the reasoning in the previous constraint:

$$\overline{\pi}(\mu(s \cdot \vec{t})) \succeq_\star @^{\sigma \to \tau\star}(\overline{\pi}(\mu(s \cdot t_1 \cdots t_{i-1})), \overline{\pi}(\mu(t_i)), \dots, \overline{\pi}(\mu(t_n)))$$

Here, σ is the type of t_i, and also as before, we see that $@^{\sigma\to\tau} \blacktriangleright @^{\rho}$ for all occurrences of some $@^{\rho}$ in $\overline{\pi}(\mu(t_i \cdot \vec{c})) = @^{\tau_m\to\kappa}(\ldots @^{\sigma}(t_i, \mathtt{c}_{\tau_1}) \ldots \mathtt{c}_{\tau_m})$. Thus, using the (Copy) clause m times, and the observation that both $@^{\sigma\to\tau\star}_{\vec{\rho},\sigma}(\ldots, \overline{\pi}(\mu(t_i)), \ldots) \succeq_\star \overline{\pi}(\mu(t_i))$ by (Select), we find the required inequality.

As with the weakly monotonic algebra case, we can satisfy the additional requirements merely by posing some standard restrictions on π and $\underline{\blacktriangleright}$, and adding a handful of constraints. We can forget about the unused tagged symbols f^-.

Compared to the theory of Chapter 5, we have gained the ability to use argument functions which are not argument preserving. Most importantly, we can use *argument filterings* as defined in Section 6.2.4.

Example 6.73. In Example 6.68 we proved termination of `eval` with a polynomial interpretation. Let us now try the same thing with StarHorpo, using argument filterings. We have the following constraints:

$$\begin{aligned}
\mathtt{dom}^\sharp(\mathtt{s}(X),\mathtt{s}(Y),\mathtt{s}(Z)) &\succ \mathtt{dom}^\sharp(X,Y,Z)\\
\mathtt{dom}^\sharp(\mathtt{0},\mathtt{s}(Y),\mathtt{s}(Y)) &\succ \mathtt{dom}^\sharp(\mathtt{0},Y,Z)\\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x),X,Y),Z) &\succ \mathtt{dom}^\sharp(X,Y,Z)\\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x),X,Y),Z) &\succ F(\mathtt{dom}(X,Y,Z))\\
\mathtt{dom}(X,Y,\mathtt{0}) &\succsim X\\
\mathtt{dom}(\mathtt{0},\mathtt{0},Z) &\succsim \mathtt{0}\\
\mathtt{eval}(\mathtt{fun}(\lambda x.F(x),X,Y),Z) &\succsim F(\mathtt{dom}(X,Y,Z))
\end{aligned}$$

Note that the tagged symbols f^- do not occur in any of these constraints at all. Thus, we can forget about them, and merely need to choose an argument function which maps each $\mathtt{c}_i^\sigma$ to $\bot_\sigma$, and a precedence which has $@^{\sigma\to\tau} \blacktriangleright @^{\sigma}, @^{\tau}$. We choose an argument function which maps $\mathtt{dom}(x,y,z)$ to $\mathtt{dom}'(x,y)$, and otherwise maps $f(\vec{x})$ to itself (except the $\mathtt{c}_i^\sigma$, which are mapped to $\bot_\sigma$ as required).

This leaves the following ordering constraints:

$$\begin{aligned}
\mathtt{dom}^\sharp(\mathtt{s}(X),\mathtt{s}(Y),\mathtt{s}(Z)) &\succ_\star \mathtt{dom}^\sharp(X,Y,Z)\\
\mathtt{dom}^\sharp(\mathtt{0},\mathtt{s}(Y),\mathtt{s}(Y)) &\succ_\star \mathtt{dom}^\sharp(\mathtt{0},Y,Z)\\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x),X,Y),Z) &\succ_\star \mathtt{dom}^\sharp(X,Y,Z)\\
\mathtt{eval}^\sharp(\mathtt{fun}(\lambda x.F(x),X,Y),Z) &\succ_\star F(\mathtt{dom}'(X,Y))\\
\mathtt{dom}'(\mathtt{s}(X),\mathtt{s}(Y)) &\succeq_\star \mathtt{s}(\mathtt{dom}'(X,Y))\\
\mathtt{dom}'(\mathtt{0},\mathtt{s}(Y)) &\succeq_\star \mathtt{s}(\mathtt{dom}'(\mathtt{0},Y,))\\
\mathtt{dom}'(X,Y) &\succeq_\star X\\
\mathtt{dom}'(\mathtt{0},\mathtt{0}) &\succeq_\star \mathtt{0}\\
\mathtt{eval}(\mathtt{fun}(\lambda x.F(x),X,Y),Z) &\succeq_\star F(\mathtt{dom}'(X,Y))
\end{aligned}$$

These constraints are oriented using a symbol precedence $\mathtt{fun} \blacktriangleright \mathtt{dom}' \blacktriangleright \mathtt{s},\mathtt{0}$. To demonstrate for example the last one:

$$
\begin{array}{ll}
& \mathtt{eval}(\mathtt{fun}(\lambda x.F(x), X, Y), Z) \\
\Rightarrow_{\mathtt{put}} & \mathtt{eval}^\star(\mathtt{fun}(\lambda x.F(x), X, Y), Z) \\
\Rightarrow_{\mathtt{select}} & \mathtt{fun}(\lambda x.F(x), X, Y) \\
\Rightarrow_{\mathtt{put}} & \mathtt{fun}^\star(\lambda x.F(x), X, Y) \\
\Rightarrow_{\mathtt{select}} & F(\mathtt{fun}^\star(\lambda x.F(x), X, Y)) \\
\Rightarrow_{\mathtt{copy}} & F(\mathtt{dom}(\mathtt{fun}^\star(\lambda x.F(x), X, Y), \mathtt{fun}^\star(\lambda x.F(x), X, Y))) \\
\Rightarrow_{\mathtt{select}} & F(\mathtt{dom}(X, \mathtt{fun}^\star(\lambda x.F(x), X, Y))) \\
\Rightarrow_{\mathtt{select}} & F(\mathtt{dom}(X, Y))
\end{array}
$$

6.6.3 General Argument Functions

The argument functions from Chapter 5.6 are not just useful in combination with path orderings; they could be used with any reduction pair. We could for example simplify a termination proof with polynomial interpretations by using argument functions first, or add argument filterings to a reduction pair that is not discussed in this thesis. Argument functions for AFSMs are defined almost exactly as before. In this definition, we consider terms over some signature $\mathcal{F}_{old}$. In the case of a standard reduction pair, this is the set $\mathcal{F}_c^\sharp$, in the case of a tagged reduction pair the set $\mathcal{F}_c^\sharp \cup \{f^- \mid f \in \mathcal{F}\}$.

Definition 6.74 (Argument Function)**.** Let $\mathcal{F}_{new}$ be a set of function symbols. An *argument function* is a function π which maps every function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}_{old}$ to a term $\lambda x_1 \ldots x_n.s : \sigma_1 \to \ldots \to \sigma_n \to \tau$ over $\mathcal{F}_{new}$ such that $FV(s) \subseteq \{x_1, \ldots, x_n\}$. An argument function is extended to meta-terms as follows:

$$
\begin{array}{rcll}
\overline{\pi}(x) & = & x & \text{for } x \in \mathcal{V} \\
\overline{\pi}(\lambda x.s) & = & \lambda x.\overline{\pi}(s) & \\
\overline{\pi}(s \cdot t) & = & \overline{\pi}(s) \cdot \overline{\pi}(t) & \\
\overline{\pi}(Z(s_1, \ldots, s_n)) & = & Z(\overline{\pi}(s_1), \ldots, \overline{\pi}(s_n)) & \text{for } Z \in \mathcal{M} \\
\overline{\pi}(f(s_1, \ldots, s_n)) & = & t[x_1 := \overline{\pi}(s_1), \ldots, x_n := \overline{\pi}(s_n)] & \text{if } \pi(f) = \lambda x_1 \ldots x_n.t
\end{array}
$$

Note that the substitution in the last clause is actually a meta-substitution.

An *argument filtering* is an argument function where each $\pi(f)$ has the form $\lambda x_1 \ldots x_n.x_i$ or $\lambda x_1 \ldots x_n.f'(x_{i_1}, \ldots, x_{i_k})$, where $i_1, \ldots, i_k$ is a sub-sequence of $1, \ldots, n$.

Theorem 6.75. *Let $(\succsim, \succ)$ be a reduction pair on terms over $\mathcal{F}_{new}$, and let $(\succsim_\pi, \succ_\pi)$ be given by: $s \succsim_\pi t$ iff $\overline{\pi}(s) \succsim \overline{\pi}(t)$, and $s \succ_\pi t$ iff $\overline{\pi}(s) \succ \overline{\pi}(t)$.*

Then $(\succsim_\pi, \succ_\pi)$ is a reduction pair.

Proof. We first make the following observation (**): $\overline{\pi}(s\gamma) = \overline{\pi}(s)\gamma^\pi$, where $\gamma^\pi(x) = \overline{\pi}(\gamma(x))$. This holds by induction on the form of s. The only non-obvious case is when $s = f(s_1, \ldots, s_n)$ and $\pi(f) = \lambda x_1 \ldots x_n.p$. Then $\overline{\pi}(s\gamma) = p[x_1 := \overline{\pi}(s_1\gamma), \ldots, x_n := \overline{\pi}(s_n\gamma)] = p[x_1 := \overline{\pi}(s_1)\gamma^\pi, \ldots, x_n := \overline{\pi}(s_n)\gamma^\pi]$ by the induction hypothesis, $= \overline{\pi}(s)\gamma^\pi$.

Having this, it is easy to see that $\succsim_\pi$ and $\succ_\pi$ are both meta-stable, and compatibility, well-foundedness, transitivity and (anti-)reflexivity are inherited from the corresponding properties of $\succsim$ and $\succ$. As for monotonicity of $\succsim_\pi$, the only non-trivial question is whether $f(s_1, \ldots, s_n) \succsim_\pi f(s'_1, \ldots, s'_n)$ when each $s_i \succsim_\pi s'_i$, but this is clear by monotonicity of $\succsim$: writing $\pi(f) = \lambda x_1 \ldots x_n.p$, if each $\overline{\pi}(s_i) \succsim \overline{\pi}(s'_i)$, then $\overline{\pi}(f(\vec{s})) = p[\vec{x} := \overline{\pi}(\vec{s})] \succsim p[\vec{x} := \overline{\pi}(\vec{s'})] = \overline{\pi}(f(\vec{s'}))$. Also $\succsim$ contains beta by (**): $\overline{\pi}((\lambda x.s) \cdot t) = (\lambda x.\overline{\pi}(s)) \cdot \overline{\pi}(t) \succsim \overline{\pi}(s)[x := \overline{\pi}(t)]$ (as $\succsim$ contains beta), and this equals $\overline{\pi}(s[x := t])$ as required. Thus, $(\succsim, \succ)$ is a reduction pair. □

Moreover, if always $s \cdot \vec{t} \succsim t_i \cdot \vec{c}$, and $\pi(\mathsf{c}^\sigma_i) = \mathsf{c}^\sigma_i$, then also $s \cdot \vec{t} \succsim_\pi t_i \cdot \vec{c}$: this is because $\overline{\pi}(s \cdot \vec{t}) = \overline{\pi}(s) \cdot \overline{\pi}(t_1) \cdots \overline{\pi}(t_n) \succsim \overline{\pi}(t_i) \cdot \vec{c} = \overline{\pi}(t_i) \cdot \overline{\pi}(\vec{c})$. To satisfy constraints A and B, we might pose restrictions either on the argument function, or on the underlying reduction pair. Alternatively, we can add these requirements as ordering constraints. As typically $\mathcal{F}$ is finite when $\mathcal{R}$ is finite, this is not likely to lead to problems.

6.7 Overview

In this chapter we have explored a definition of dependency pairs for AFSMs. This definition follows the *dynamic* style, where collapsing dependency pairs are included. The method is *complete*: an AFSM is terminating if and only if it does not admit an infinite dependency chain.

The higher-order case provides many challenges not present in the first-order case: collapsing dependency pairs, dangling variables, functional rules, typing issues... we have seen how to solve these issues, and gain a method which makes use of weak reduction pairs rather than strong ones. Moreover, we have discussed a restriction to the class of *abstraction-simple* systems where we can strengthen the method, which for example allows us to use almost unrestricted argument filterings. We have considered *formative reductions*, which allow us to drop certain rules from dependency chains. And we have seen how the methods of Chapters 4 and 5 can be used with dependency pairs, both in the basic and the abstraction-simple setting, and considered a set of standard constraints for both methods to satisfy the (tagged) subterm property.

It is worth noting that the "basic" definitions in this chapter are a fair bit more complicated than in the first-order case. They could be significantly simplified, for example by η-normalising the system beforehand, not keeping track of the meta-variable conditions (which we have not even used so far!) and replacing free variables in dependency pairs by symbols c^σ immediately. The reason not to do so is twofold. First, the completeness result depends on the full definition, which includes all these features – and since dependency pairs in the first-order setting are often used for non-termination [47], this is very important for future extensions. Moreover, in the next chapter we will consider the *dependency graph*, which can make use of these features.

7 Improving Dependency Pairs

Or, Can we do even better?

In Chapter 6 we have seen a basic extension of the dependency pair approach to the higher-order setting. Although this method already provides many advantages over simply using rule removal, it is not yet as powerful as it could be.

In the first-order world, the original definition of dependency pairs has been extended and improved in a variety of ways. For example, the dependency graph [9], usable rules [48, 53] and the subterm criterion [53] all provide methods to manipulate dependency chains, and make it easier to prove chain-freeness. Modern definitions feature the dependency pair *framework* [46], where groups of tuples $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathit{flag})$ are iteratively transformed by dependency pair *processors*, until either termination of all groups is proved, or a proof of non-termination is obtained. The framework is used both for innermost and full termination.

Several of these first-order techniques can be lifted to the higher-order setting, although some to a greater extent than others. For another direction in the higher-order world, we might look at the *static dependency pair approach* (which was discussed to some extent in Chapter 6.1). This method avoids collapsing dependency pairs, and therefore has the potential to add significant extra power.

This chapter seeks to build on the work we did in Chapter 6 and provide improvements in three ways:

- (partially) extending the first-order dependency pair framework to the higher-order setting;
- extending existing first-order processors to the higher-order setting;
- transposing the static dependency pair approach to AFSMs, and discussing how it can be analysed in the same framework as the dynamic approach.

Where Chapter 6 mostly introduced new ideas, in this chapter we will primarily adapt existing ideas to the current setting. Of course, significant changes to these techniques are still needed, and a few new methods are discussed as well.

Chapter Setup. To start off, Section 7.1 discusses a simplified version of the first-order dependency pair framework, as well as a number of existing techniques within this framework. In Section 7.2 I will introduce a dependency pair framework for higher-order rewriting. We will also rephrase the results so far as *dependency pair processors*.

As a first new result, Section 7.3 lists some modifications to collapsing dependency pairs, phrased as dependency pair processors.

Next, we study a number of techniques lifted from the first-order dependency pair approach: the dependency graph (Section 7.4), the subterm criterion (Section 7.5), and usable rules (Section 7.6). These techniques, especially the latter two, have limited applicability in the presence of collapsing dependency pairs, but they are useful when we consider a set of non-collapsing dependency pairs. In particular, usable rules offer additional power when considering the first-order part of an AFSM. This is explored more fully in Section 7.7.

Finally, in Section 7.8, I will briefly present the *static* dependency pair approach, which was originally defined for HRSs in [87]. As we will see, the static and dynamic approaches can be combined in the same general framework.

This chapter is primarily based on [80]; the dependency graph already appears in the conference version of this paper, [79]. However, where [79, 80] concern higher-order rewriting in the AFS formalism, here of course AFSMs are considered. Section 7.7 is based on a separate paper, [41].

7.1 The First-order Dependency Pair Framework

Let us start this chapter by considering the first-order dependency pair framework, and those processors for which we will derive higher-order extensions.

As in Chapter 6.2, we will consider as an example the system `quot`:

$$\begin{array}{rcl} \mathtt{minus}(x, \mathtt{0}) & \Rightarrow & x \\ \mathtt{minus}(\mathtt{s}(x), \mathtt{s}(y)) & \Rightarrow & \mathtt{minus}(x, y) \\ \mathtt{quot}(\mathtt{0}, \mathtt{s}(y)) & \Rightarrow & \mathtt{0} \\ \mathtt{quot}(\mathtt{s}(x), \mathtt{s}(y)) & \Rightarrow & \mathtt{s}(\mathtt{quot}(\mathtt{minus}(x, y), \mathtt{s}(y))) \end{array}$$

7.1.1 The Dependency Pair Framework

So far, we have considered the notion of *chain-free* (and, in the higher-order case, *formative chain-free*) sets of dependency pairs. That is, we study the question whether a minimal dependency chain over a given set $\mathcal{P}$ exists. The dependency pair framework takes this a step further, and considers *dependency pair problems*, where also the underlying set of rules $\mathcal{R}$ is a parameter. Dependency pair problems are transformed and simplified by *dependency pair processors*.

Definition 7.1. A *(first-order) dependency pair problem* is a triple $(\mathcal{P}, \mathcal{R}, \mathit{flag})$[1] of a set of dependency pairs, a set of rules, and a flag that is either `minimal` or `arbitrary`. A dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathit{flag})$ is *finite* if there is no dependency chain, such that:

- all $\rho_i \in \mathcal{P}$;
- each $s_i \Rightarrow^*_{\mathcal{R}} t_i$;
- if $\mathit{flag} = \texttt{minimal}$, then the dependency chain is minimal.

A dependency problem $(\mathcal{P}, \mathcal{R}, \mathit{flag})$ is *infinite* if it is not finite, or $\mathcal{R}$ is non-terminating.[2]

A *dependency pair processor* is a function which takes a dependency pair problem and returns either a set of dependency pair problems, or `NO`. A processor Proc is called *sound* if for all dependency pair problems A: if $\mathit{Proc}(A)$ is not `NO`, and all elements of $\mathit{Proc}(A)$ are finite, then also A is finite. A processor Proc is called *complete* if for all dependency pair problems A: if $\mathit{Proc}(A)$ is `NO`, or any of the elements of $\mathit{Proc}(A)$ is infinite, then A is infinite. To prove termination, we must use sound processors; for non-termination complete ones are necessary.

We could use roughly the following algorithm to prove termination of a first-order term rewriting system $\mathcal{R}$:

1. let A be the set $\{(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal})\}$;
2. while A is non-empty:
   - select a problem P from A;
   - choose a sound processor S and let $A := A \setminus \{P\} \cup S(P)$ (if $S(A) = \texttt{NO}$, then this processor cannot be applied);
3. if step 2 completes, conclude termination.

The first step in this algorithm defines a set A such that $\mathcal{R}$ is terminating if all dependency pair problems in A are finite; the second step iteratively transforms A while always preserving this property.

Similarly, for non-termination we can use the following algorithm:

1. let A be the set $\{(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal})\}$;
2. while A is non-empty:

[1] The usual definition of the dependency pair framework in e.g. [46] considers an additional parameter $\mathcal{Q}$, which is used for instance for innermost termination. We will ignore this parameter because this thesis considers only full termination. Moreover, since in the first-order setting dependency pairs, like rules, are just pairs of terms where the right-hand side contains all variables in the left-hand side, and the left-hand side is not a variable, $\mathcal{P}$ is there simply assumed to be a TRS.

[2] Thus, a DP problem can be both finite and infinite. For a discussion of this I refer to [46].

- select a problem P from A;
- choose a complete processor S; if $S(P) = \texttt{NO}$ then conclude non-termination, otherwise let $A := A \setminus \{P\} \cup S(P)$.

Here, A always has the property that if $\mathcal{R}$ is terminating, there is an infinite DP problem in A.

An example of a dependency pair processor is the following:

Theorem 7.2 (Empty Set Processor)**.** *The function which maps a dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathit{flag})$ to $\emptyset$ if $\mathcal{P} = \emptyset$ is a sound and complete processor.*

Note: In this theorem, as in all other processors that will be presented in this chapter, a processor is phrased as "a function which maps a problem C with certain properties to S". This should be read as: "a function which maps a problem C to S if C satisfies the given properties, and otherwise to $\{P\}$".

7.1.2 Processors with a Reduction Pair

Theorem 6.7, which allows us to use a reduction pair to prove that a set of dependency pairs is chain-free, can be generalised to the following processor:

Theorem 7.3 (Reduction Pair Processor)**.** *The function which maps a dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathit{flag})$ to $\{(\mathcal{P}_2, \mathcal{R}, \mathit{flag})\}$ if:*

- *$\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$,*
- *there is a weak reduction pair $(\succsim, \succ)$ such that $l \succ p$ for all $l \Rrightarrow p \in \mathcal{P}_1$, $l \succsim p$ for $l \Rrightarrow p \in \mathcal{P}_2$ and $l \succsim r$ for all rules $l \Rightarrow r \in \mathcal{R}$*

is a sound and complete processor.

Reduction pairs can also be used to simplify a dependency pair problem by only altering $\mathcal{R}$: if we can orient the rules and dependency pairs with a *strong* reduction pair, then we can remove all those rules which were oriented strictly.

Theorem 7.4 (Rule Removal Processor)**.** *([119]) The function which maps a dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathit{flag})$ to $(\mathcal{P}_2, \mathcal{R}_2, \mathit{flag})$ provided:*

- *$\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$, $\mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2$,*
- *there is a strong reduction pair $(\succsim, \succ)$ such $l \succ r$ for all $l \Rrightarrow r \in \mathcal{P}_1$ and $l \Rightarrow r \in \mathcal{R}_1$, and $l \succsim r$ for all $l \Rrightarrow r \in \mathcal{P}_2$ and $l \Rightarrow r \in \mathcal{R}_2$.*

is a sound and complete processor.

This processor could not be phrased in the old "chain-free" terminology as it involves a (possible) change to $\mathcal{R}$.

Another method for the first-order setting which we have seen so far are the *usable rules* from Chapter 6.2.5, an idea which found a parallel (although not an extension) in the formative rules we saw later on. The idea that a minimal dependency chain can be transformed into a dependency chain which uses only usable rules can be reflected in a processor which alters the set $\mathcal{R}$:

Theorem 7.5 (Usable Rules Processor)**.** *([119]) The function which maps a dependency pair problem* $(\mathcal{P}, \mathcal{R}, \mathtt{minimal})$ *to* $\{(\mathcal{P},\ UR(\mathcal{P}) \cup \{\mathtt{p}(x,y) \Rightarrow x,\ \mathtt{p}(x,y) \Rightarrow y\}, \mathtt{arbitrary})\}$ *is a sound processor.*

This processor is not, however, complete: the new dependency pair problem may be infinite even when the original one is not. And since it loses minimality, it is probably better to use the following processor instead:

Theorem 7.6 (Reduction Pair Processor with Usable Rules)**.** *The function which maps a dependency pair problem* $(\mathcal{P}, \mathcal{R}, \mathtt{minimal})$ *to* $\{(\mathcal{P}_2, \mathcal{R}, \mathtt{minimal})\}$ *if:*

- $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$*,*
- *there is a reduction pair* $(\succsim, \succ)$ *such that* $l \succ p$ *for all* $l \Rrightarrow p \in \mathcal{P}_1,\ l \succsim p$ *for* $l \Rrightarrow p \in \mathcal{P}_2,\ l \succsim r$ *for all rules* $l \Rightarrow r \in UR(\mathcal{P})$ *and* $\mathtt{p}(x,y) \succsim x, y$ *for a fresh symbol* $\mathtt{p}$

is a sound and complete processor.

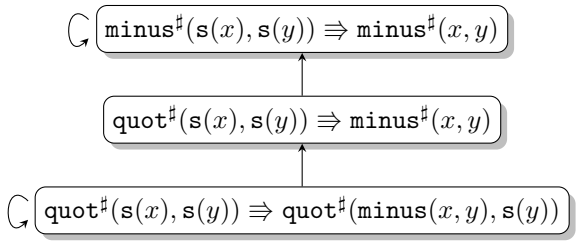
The notion of *chain-free* used so far corresponds to finiteness of the dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal})$, where $\mathcal{R}$ is the original set of rules. As demonstrated by for instance the rule removal processor, the dependency pair framework can go further. We can mix and match processors and alter each of the three components of the problem (and also the fourth, for innermost termination, which is omitted here).

In the rest of Section 7.1 we will study two more extensions, in the form of dependency pair processors: the *dependency graph* and the *subterm criterion*.

7.1.3 The Dependency Graph

To see whether a system has an infinite dependency chain, it makes sense to ask what form such a chain would have. This question is studied by means of a *dependency graph*.

The dependency graph of a pair $(\mathcal{P}, \mathcal{R})$ of a set of dependency pairs and a set of rules, is a directed graph whose nodes are the elements of $\mathcal{P}$, and which has an edge from $l \Rrightarrow p$ to $u \Rrightarrow v$ if there are substitutions γ and δ such that $p\gamma \Rightarrow_{\mathcal{R}}^{*} u\delta$. For example, the dependency graph of $(\mathsf{DP}(\mathcal{R}_{\mathtt{quot}}), \mathcal{R}_{\mathtt{quot}})$ is as follows:



Now, if there is a dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$, then there will always be an edge in the graph from dependency pair ρ_i to ρ_{i+1}. Since the graph is finite (which always holds if the original TRS is finite), an infinite chain corresponds to a cycle in the graph. Here, a *cycle* is a non-empty set $\mathcal{P}$ of dependency pairs such that for all $\rho_1, \rho_2 \in \mathcal{P}$ there is a (non-empty) path in the graph from ρ_1 to ρ_2 which only traverses pairs from $\mathcal{P}$. We might phrase this idea as follows:

Theorem 7.7. *(Adapted from [9, 46]) A set of dependency pairs is chain-free if and only if all cycles in its dependency graph are chain-free.*

The dependency graph is not in general computable, which is why *approximations* are often used. An approximation of the dependency graph G is a graph with the same nodes as G, but which may have additional edges.

The dependency graph of our running `quot` example has two cycles. In order to prove termination, it is sufficient to find a reduction pair such that $\mathtt{minus}^\sharp(\mathtt{s}(x),\mathtt{s}(y)) \succ \mathtt{minus}^\sharp(x,y)$ and $l \succsim r$ for all rules, and a(nother) reduction pair such that $\mathtt{quot}^\sharp(\mathtt{s}(x),\mathtt{s}(y)) \succ \mathtt{quot}^\sharp(\mathtt{minus}(x,y),\mathtt{s}(y))$ and $l \succsim r$ for all rules. The fact that we can deal with groups of dependency pairs separately can make it significantly simpler to find reduction pairs.

It is worth observing that, if a set of dependency pairs $\mathcal{P}$ is chain-free, the same holds for all subsets $\mathcal{P}'$ of $\mathcal{P}$. Moreover, a dependency graph might have exponentially many cycles, as demonstrated by the graph on the right (which has 30 cycles and originates from a system with only 5 rules). Therefore modern approaches consider only *maximal cycles*, also called *strongly connected components*. In the graph on the right, the strongly connected components are $\{7, 8, 9, 10\}$ and $\{11, 12, 13, 14\}$.

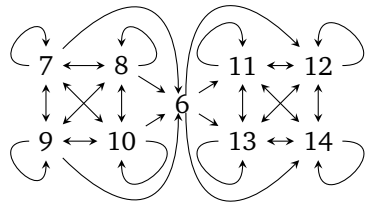


Figure 7.1: Dependency Graph in [53]

This reasoning leads to the following processor:

Theorem 7.8. *([46]) The function which maps $(\mathcal{P}, \mathcal{R}, \mathit{flag})$ to $(\{C \mid C$ is a strongly connected component of an approximation of the dependency graph of $(\mathcal{P}, \mathcal{R})\}, \mathcal{R}, \mathit{flag})$ is a sound and complete processor.*

7.1.4 The Subterm Criterion

Rather than going to the trouble of finding a reduction pair, most automated tools which implement dependency pairs first attempt to prove that a given set of dependency pairs does not admit a dependency chain by using the *subterm criterion*. Finding out whether a set of dependency pairs satisfies the subterm criterion is typically easy.

Definition 7.9. Let ν be a function which assigns, to every n-ary dependency pair symbol $f^\sharp$, one of its argument positions $i \in \{1, \ldots, n\}$ (ν is a *projection function*). We extend ν to a function on terms by defining $\overline{\nu}(f^\sharp(s_1, \ldots, s_n)) = s_i$ if $\nu(f^\sharp) = i$.

The phrase *subterm criterion* refers to the follow theorem:

Theorem 7.10 (Subterm Criterion Processor)**.** *(Adapted from [53]) The function which maps $(\mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{R}, \texttt{minimal})$ to $\{(\mathcal{P}_2, \mathcal{R}, \texttt{minimal})\}$ provided a projection function ν exists such that $\overline{\nu}(l) \rhd \overline{\nu}(p)$ for $l \Rrightarrow p \in \mathcal{P}_1$ and $\overline{\nu}(l) = \overline{\nu}(p)$ for $l \Rrightarrow p \in \mathcal{P}_2$, is a sound and complete processor.*

This holds because for all s_i, t_i in a minimal dependency chain, $\overline{\nu}(s_i)$ and $\overline{\nu}(t_i)$ are strict subterms of s_i and t_i, and therefore terminating.

The subterm criterion on itself is not sufficient to show termination of the `quot` example, but at least we can use it to eliminate some dependency pairs: choosing $\nu(\texttt{minus}^\sharp) = \nu(\texttt{quot}^\sharp) = 2$ we obtain:

$$\begin{array}{rcccccl}
\overline{\nu}(\texttt{minus}^\sharp(\texttt{s}(x), \texttt{s}(y))) & = & \texttt{s}(y) & \rhd & y & = & \overline{\nu}(\texttt{minus}^\sharp(x, y)) \\
\overline{\nu}(\texttt{quot}^\sharp(\texttt{s}(x), \texttt{s}(y))) & = & \texttt{s}(y) & = & \texttt{s}(y) & = & \overline{\nu}(\texttt{quot}^\sharp(\texttt{minus}(x, y), \texttt{s}(y))) \\
\overline{\nu}(\texttt{quot}^\sharp(\texttt{s}(x), \texttt{s}(y))) & = & \texttt{s}(y) & \rhd & y & = & \overline{\nu}(\texttt{minus}^\sharp(x, y))
\end{array}$$

This shows that the TRS `quot` is terminating if and only if there is no dependency chain where every step uses the dependency pair $\texttt{quot}^\sharp(\texttt{s}(x), \texttt{s}(y)) \Rrightarrow \texttt{quot}^\sharp(\texttt{minus}(x, y), \texttt{s}(y))$.

7.1.5 A Termination Proof with the Framework

To demonstrate how the dependency pair framework works in practice, let us try it out on a small example. We consider the TRS with symbols $\texttt{f}, \texttt{g}, \texttt{h}, \texttt{a}$ and $\texttt{b}$, and rules:

$$\begin{array}{rcl}
\texttt{g}(\texttt{h}(\texttt{a}(x))) & \Rightarrow & \texttt{g}(x) \\
\texttt{g}(\texttt{a}(x)) & \Rightarrow & \texttt{g}(\texttt{h}(\texttt{g}(x))) \\
\texttt{h}(\texttt{b}(x)) & \Rightarrow & \texttt{h}(\texttt{f}(\texttt{h}(x), x)) \\
\texttt{g}(x) & \Rightarrow & \texttt{a}(x) \\
\texttt{h}(x) & \Rightarrow & \texttt{b}(x)
\end{array}$$

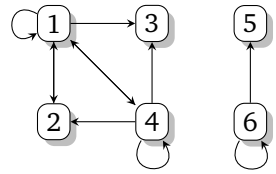
To start, we calculate the dependency pairs: the set $\mathcal{P}_1 := \mathsf{DP}(\mathcal{R})$ consists of:

$$\begin{array}{rcl}
\mathtt{g}^\sharp(\mathtt{h}(\mathtt{a}(x))) & \Rrightarrow & \mathtt{g}^\sharp(x) \\
\mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{g}^\sharp(\mathtt{h}(\mathtt{g}(x))) \\
\mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{h}^\sharp(\mathtt{g}(x)) \\
\mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{g}^\sharp(x) \\
\mathtt{h}^\sharp(\mathtt{b}(x)) & \Rrightarrow & \mathtt{h}^\sharp(\mathtt{f}(\mathtt{h}(x), x)) \\
\mathtt{h}^\sharp(\mathtt{b}(x)) & \Rrightarrow & \mathtt{h}^\sharp(x)
\end{array}$$

We thus know: *$\mathcal{R}$ is terminating if $(\mathcal{P}_1, \mathcal{R}, \mathtt{minimal})$ is a finite dependency pair problem.*

We use the dependency graph processor on this problem. Observing that a term of the form $\mathtt{g}(s)$ cannot reduce to a term of the form $\mathtt{b}(t)$, we make the following approximation:

$$\begin{array}{rrcl}
1 & \mathtt{g}^\sharp(\mathtt{h}(\mathtt{a}(x))) & \Rrightarrow & \mathtt{g}^\sharp(x) \\
2 & \mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{g}^\sharp(\mathtt{h}(\mathtt{g}(x))) \\
3 & \mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{h}^\sharp(\mathtt{g}(x)) \\
4 & \mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{g}^\sharp(x) \\
5 & \mathtt{h}^\sharp(\mathtt{b}(x)) & \Rrightarrow & \mathtt{h}^\sharp(\mathtt{f}(\mathtt{h}(x), x)) \\
6 & \mathtt{h}^\sharp(\mathtt{b}(x)) & \Rrightarrow & \mathtt{h}^\sharp(x)
\end{array}$$



The graph has two SCCs: $\{1, 2, 4\}$ and $\{6\}$.

Thus, let us define:

$$\mathcal{P}_2 := \left\{ \begin{array}{rcl}
\mathtt{g}^\sharp(\mathtt{h}(\mathtt{a}(x))) & \Rrightarrow & \mathtt{g}^\sharp(x) \\
\mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{g}^\sharp(\mathtt{h}(\mathtt{g}(x))) \\
\mathtt{g}^\sharp(\mathtt{a}(x)) & \Rrightarrow & \mathtt{g}^\sharp(x)
\end{array} \right\}$$

$$\mathcal{P}_3 := \{ \quad \mathtt{h}^\sharp(\mathtt{b}(x)) \quad \Rrightarrow \quad \mathtt{h}^\sharp(x) \quad \}$$

The dependency graph processor maps $(\mathcal{P}_1, \mathcal{R}, \mathtt{minimal})$ to the set consisting of $(\mathcal{P}_2, \mathcal{R}, \mathtt{minimal})$ and $(\mathcal{P}_3, \mathcal{R}, \mathtt{minimal})$.

Thus, *$\mathcal{R}$ is terminating if $(\mathcal{P}_2, \mathcal{R}, \mathtt{minimal})$ and $(\mathcal{P}_3, \mathcal{R}, \mathtt{minimal})$ are both finite dependency pair problems.*

We select the problem $(\mathcal{P}_1, \mathcal{R}, \mathtt{minimal})$ and use the reduction pair processor. For the reduction pair, let us use polynomial interpretations; we will use an interpretation with $\mathcal{J}(\mathtt{f}) = \boldsymbol{\lambda} xy.0$, $\mathcal{J}(\mathtt{g}) = \mathcal{J}(\mathtt{a}) = \boldsymbol{\lambda} x.x + 1$ and $\mathcal{J}(\mathtt{h}) = \mathcal{J}(\mathtt{g}^\sharp) = \boldsymbol{\lambda} x.x$. With this interpretation, $[\![l]\!]_\mathcal{J} \geq [\![r]\!]_\mathcal{J}$ for all rules $l \Rightarrow r$. For the dependency pairs we have:

$$\begin{array}{rcccccl}
[\![\mathtt{g}^\sharp(\mathtt{h}(\mathtt{a}(x)))]\!] & = & x+1 & > & x & = & [\![\mathtt{g}^\sharp(x)]\!] \\
[\![\mathtt{g}^\sharp(\mathtt{a}(x))]\!] & = & x+1 & \geq & x+1 & = & [\![\mathtt{g}^\sharp(\mathtt{h}(\mathtt{g}(x)))]\!] \\
[\![\mathtt{g}^\sharp(\mathtt{a}(x))]\!] & = & x+1 & > & x & = & [\![\mathtt{g}^\sharp(x)]\!]
\end{array}$$

Thus, the reduction pair processor maps $(\mathcal{P}_2, \mathcal{R}, \texttt{minimal})$ to $\{(\mathcal{P}_4, \mathcal{R}, \texttt{minimal})\}$, where $\mathcal{P}_4 = \{\mathtt{g}^\sharp(\mathtt{a}(x)) \Rrightarrow \mathtt{g}^\sharp(\mathtt{h}(\mathtt{g}(x)))\}$. This gives:

$\mathcal{R}$ is terminating if $(\mathcal{P}_3, \mathcal{R}, \texttt{minimal})$ and $(\mathcal{P}_4, \mathcal{R}, \texttt{minimal})$ are both finite dependency pair problems.

Now, let us consider the problem $(\mathcal{P}_3, \mathcal{R}, \texttt{minimal})$, with the subterm criterion processor. If $\nu(\mathtt{h}^\sharp) = 1$, we have:

$$\overline{\nu}(\mathtt{h}^\sharp(\mathtt{b}(x))) = \mathtt{b}(x) \rhd x = \overline{\nu}(\mathtt{h}^\sharp(x))$$

Thus, the subterm criterion processor maps $(\mathcal{P}_3, \mathcal{R}, \texttt{minimal})$ to $(\emptyset, \mathcal{R}, \texttt{minimal})$. We have: *$\mathcal{R}$ is terminating if $(\emptyset, \mathcal{R}, \texttt{minimal})$ and $(\mathcal{P}_4, \mathcal{R}, \texttt{minimal})$ are both finite dependency pair problems.*

Using the finite set processor on $(\emptyset, \mathcal{R}, \texttt{minimal})$, we obtain:

$\mathcal{R}$ is terminating if $(\mathcal{P}_4, \mathcal{R}, \texttt{minimal})$ is a finite dependency pair problem.

We select the remaining dependency pair problem $(\mathcal{P}_4, \mathcal{R}, \texttt{minimal})$, and consider the dependency graph processor. The dependency graph of $\mathcal{P}_4$ has no edges! Thus, the dependency graph processor maps $(\mathcal{P}_4, \mathcal{R}, \texttt{minimal})$ to $\emptyset$. We conclude:

$\mathcal{R}$ is terminating.

In this example, we did not use any $\mathcal{R}$-altering steps: we could have done all this with the original "chain-free" notion. This is not always the case. In the higher-order setting, we will for instance use a formative rules processor, which often allows us to throw away rules.

Conclusion. This completes the discussion of the dependency pair framework for first-order term rewriting. In the rest of this chapter, we will consider a higher-order extension of this framework. Moreover, several processors will be presented: extensions of rule removal, the dependency graph, subterm criterion and usable rules, and in addition several new ones specialised for the higher-order setting.

7.2 The Dependency Pair Framework

The dependency pair framework in the higher-order setting is defined much like the first-order framework presented in Section 7.1.1. There is one major difference: we will have to cater for formative dependency chains. In the last chapter, we have seen two examples where we gain strength if we can assume that the reductions $t_i \Rightarrow_{\mathcal{R}}^* s_{i+1}$ in a dependency chain are formative: we only need to consider formative rules, and may use tagging to weaken the subterm criterion. Unfortunately, the results from Chapter 6.4 do not allow us to transform a given (minimal) dependency chain into a formative chain. Instead, we will introduce an additional flag.

Definition 7.11. A *higher-order dependency pair problem* (or *DP problem*) is a tuple $(\mathcal{P}, \mathcal{R}, m, f)$ where:

- $\mathcal{P}$ is a set of dependency pairs;
- $\mathcal{R}$ is a set of rules;
- m is either `minimal` or `arbitrary`;
- f is either `formative` or `all`.

A higher-order dependency pair problem $(\mathcal{P}, \mathcal{R}, m, f)$ is *finite* if there is no infinite $(\mathcal{P}, \mathcal{R}, m, f)$-*chain*. That is, there is no chain $C = [(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that:

- for all i: $\rho_i \in \mathcal{P} \cup \{\texttt{beta}\}$;
- for all i: $t_i \Rightarrow_{\mathcal{R}}^{*} s_{i+1}$;
- if $m = \texttt{minimal}$ then C is a minimal dependency chain;
- if $f = \texttt{formative}$ then C is a formative dependency chain.

A higher-order dependency pair problem is *infinite* if it is not finite, or the set of rules $\mathcal{R}$ is non-terminating.

> *Comment:* This notion of an infinite dependency pair problem is copied directly from the first-order definition. However, there has been no non-termination research with a higher-order dependency pair approach at all. Thus, it is not clear whether this definition of "infinite" will suffice. It may well be revised in future versions.

The definition of a higher-order dependency pair problem is a fair bit more elaborate than needed to present the results we have seen so far, all of which concern minimal, formative dependency chains. However, once we start manipulating the set $\mathcal{R}$ (as is done with a number of processors in the first-order case), the various flags will become very relevant. In this chapter, however, we will primarily see processors which leave these flags intact.

We will typically refer to a higher-order dependency pair problem as just a "dependency pair problem".

Theorem 7.12. *An AFSM $\mathcal{R}$ is terminating if and only if the dependency pair problem* $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal}, \texttt{formative})$ *is finite. An AFSM is non-terminating if and only if the dependency pair problem* $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal}, \texttt{formative})$ *is infinite.*

Proof. The first part is a reformulation of Theorem 6.44, the second part a consequence of the first. □

Theorem 7.12 provides the basis for the higher-order version of the dependency pair framework: to prove termination of an AFSM, it suffices to prove finiteness of some dependency pair problem. To do the latter, we will use processors.

Definition 7.13. A *dependency pair processor* is a function which takes a dependency pair problem and returns a set of dependency pair problems or `NO`.

A dependency pair processor $Proc$ is *sound* if a dependency pair problem P is finite whenever $Proc(P)$ is not `NO` and all elements of $Proc(P)$ are finite.

A dependency pair processor $Proc$ is *complete* if a dependency pair problem P is infinite whenever $Proc(P)$ is `NO` or some element of $Proc(P)$ is infinite.

When considering termination, we are mostly interested in *sound* processors; when considering non-termination in *complete* ones. Of course, processors which are both sound *and* complete, have the advantage that using them does not sacrifice generality.

To prove termination of an AFSM $\mathcal{R}$, we can use the following algorithm:

1. let A be the set $\{(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal}, \texttt{formative})\}$;
2. while A is non-empty:
   - select a problem P from A;
   - choose a sound processor S and let $A := A \setminus \{P\} \cup S(P)$ (if $S(A) = \texttt{NO}$, then this processor cannot be applied).
3. if the second step completes (so A ends up empty), conclude termination (if it does not complete, we cannot conclude anything).

And to prove non-termination:

1. let A be the set $\{(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal}, \texttt{formative})\}$;
2. while A is non-empty:
   - select a problem P from A;
   - choose a complete processor S;
   - if $S(A) = \texttt{NO}$, then conclude non-termination;
   - otherwise let $A := A \setminus \{P\} \cup S(P)$
3. if the second step completes (so A ends up empty), we cannot conclude anything.

The correctness of both algorithms follows from Theorem 7.12 and the definitions of sound and complete. In the first algorithm, we constantly preserve the property that $\mathcal{R}$ is terminating if all elements of A are finite. In the second, we preserve the property that $\mathcal{R}$ is non-terminating if and only if A contains an infinite element.

Of course, these algorithms are highly non-deterministic: at any iteration of the while loop we must choose some processor and apply it to some problem. In practice we will typically use a strategy to select the right processor. For example, it makes sense to first try to use only those processors which cannot make the problem harder (so far we have only seen processors which make the problem easier, but for instance the usable rules processor we will see in Section 7.6 does come with downsides), and from those processors, first focus on the computationally easiest ones. Moreover, we may want to combine the algorithms for termination and non-termination to avoid double work.

In the rest of this chapter, we will consider many processors, most of which are both sound and complete. The following lemma makes it easy to quickly conclude completeness in most cases:

Lemma 7.14. *A processor which maps $(\mathcal{P}, \mathcal{R}, m, f)$ to a set consisting of problems $(\mathcal{P}', \mathcal{R}', m, f)$ with $\mathcal{P}' \subseteq \mathcal{P}$ and $\mathcal{R}' \subseteq \mathcal{R}$ is complete.*

Proof. The processor is complete if for any element $(\mathcal{P}', \mathcal{R}', m, f)$ of the result: if $C = (\mathcal{P}', \mathcal{R}', m, f)$ is infinite, then so is $(\mathcal{P}, \mathcal{R}, m, f)$. There are two reasons why C may be infinite: $\mathcal{R}'$ is non-terminating, or C is not finite. In the former case, $\mathcal{R}$ is non-terminating as well, since $\Rightarrow_{\mathcal{R}}$ contains $\mathcal{R}'$. In the latter, any C-chain is also a $(\mathcal{P}, \mathcal{R}, m, f)$-chain. □

To start, let us consider the processors we can get almost for free: those which follow immediately from the results of Chapter 6, or are easy adaptations thereof.

Note: In each of the processors presented in this chapter, a processor is phrased as "a function which maps a problem C with certain properties to S". This should be read as: "a function which maps a problem C to S if C satisfies the given properties, and to $\{P\}$ otherwise".

Theorem 7.15 (Empty Set Processor)**.** *The processor which maps a DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ to $\emptyset$ if $\mathcal{P}$ is empty, is both sound and complete.*

Proof. This holds because a dependency chain with all $\rho_i = \texttt{beta}$ cannot exist (see also Observation II below Definition 6.28). □

Theorem 7.16 (Reduction Pair Processor)**.** *The processor which maps a DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ to $\{(\mathcal{P}_2, \mathcal{R}, m, f)\}$ provided the following clauses hold, is both sound and complete:*

- *$\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$;*
- *one of the following holds:*
  - *there is a standard reduction pair for $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$ (Definition 6.69);*
  - *$\mathcal{P}$ and $\mathcal{R}$ are both abstraction-simple and $f = \texttt{formative}$ and there is a tagged reduction pair for $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R})$ (Definition 6.70).*

Proof. Completeness holds by Lemma 7.14. As for soundness: this is an adaptation of Theorems 6.36 and 6.67. In both theorems, we merely used the reduction pair to show that if a dependency chain using only pairs in $\mathcal{P}$ and rules in $\mathcal{R}$ exists, then it has a tail which uses only pairs in $\mathcal{P}_2$. Thus, minimality and use of formative reductions are not needed, but are preserved if present. □

Theorem 7.17 (Formative Rules Processor)**.** *The processor which maps a DP problem* $(\mathcal{P}, \mathcal{R}, m, \texttt{formative})$ *to* $\{(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}), m, \texttt{formative})\}$ *is both sound and complete.*

Proof. Completeness is obvious: since $FR(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{R}^+$, every formative dependency chain which uses only rules in $FR(\mathcal{P}, \mathcal{R})^+$ is also a formative dependency which uses only rules in $\mathcal{R}^+$. As for soundness, this we obtain from Lemma 6.42: a formative l-reduction $t_i \Rightarrow^*_{\mathcal{R},in} l\gamma$ with $\rho_{i+1} = l \Rrightarrow p$ uses only rules in $FR(\rho_{i+1}, \mathcal{R}) \subseteq FR(\mathcal{P}, \mathcal{R})$. □

Thus, we have summarised the main results of Chapter 6 in three processors: one which eliminates a dependency pair problem, one which alters the set $\mathcal{P}$ and one which alters the set $\mathcal{R}$.

Considering Theorem 6.45 we might also think of a processor which maps a dependency pair problem $(\mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{R}, m, \texttt{formative})$ to $\{(\mathcal{P}_2, \mathcal{R}, m, \texttt{formative})\}$ if a standard dependency pair exists for $(\mathcal{P}_1, \mathcal{P}_2, FR(\mathcal{P}_1 \cup \mathcal{P}_2, \mathcal{R}))$. However, we will not need that: this processor is obtained as a combination of the formative rules processor and the reduction pair processor.

Apart from the processors corresponding to the results we have seen before, let us consider a few new ones. In the first place, the following processor provides an easy way to alter the minimality flag.

Theorem 7.18 (Introducing Minimality Processor)**.** *The processor which maps a problem* $(\mathcal{P}, \mathcal{R}, \texttt{arbitrary}, f)$*, where* $\mathcal{R}$ *is a terminating set of rules, to* $\{(\mathcal{P}, \mathcal{R}, \texttt{minimal}, f)\}$ *is both sound and complete.*

Proof. This is a basic simplification: if $\mathcal{R}$ is terminating, then the minimality constraint in a $(\mathcal{P}, \mathcal{R}, m, f)$-dependency chain is automatically satisfied. □

This processor originates from the first-order case (it is for instance implemented in AProVE), and shows a way to restore minimality if we have lost it through other processors. We will see a processor that loses the minimality flag (as a price for dropping rules) in Section 7.6.

Finally, let us consider one more processor that changes $\mathcal{R}$; we saw the first-order counterpart of this processor in Section 7.1.2.

Theorem 7.19 (Rule Removal Processor)**.** *The processor which maps a DP problem* $(\mathcal{P}, \mathcal{R}, m, f)$ *to* $\{(\mathcal{P}_2, \mathcal{R}_2, m, f)\}$ *provided the follow clauses hold, is both sound and complete:*

- $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2,\ \mathcal{R} = \mathcal{R}_1 \uplus \mathcal{R}_2$;
- *there is a strong reduction pair* $(\succsim, \succ)$ *such that one of the following holds:*
  - $(\succsim, \succ)$ *is a standard reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R}_2)$ *and* $l \succ r$ *for all rules* $l \Rightarrow r \in \mathcal{R}_1$;
  - $\mathcal{P}$ *and* $\mathcal{R}$ *are both abstraction-simple and* $(\succsim, \succ)$ *is a tagged reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{R}_2)$ *and* $l \succ \mathtt{tag}(r)$ *for all rules* $l \Rightarrow r \in \mathcal{R}_1$.

Proof. Completeness holds by Lemma 7.14; we consider soundness.

From the proofs of Theorem 6.31 and Theorem 6.65, as well as Theorems 6.36 and 6.67, we know that, given an infinite dependency chain over $\mathcal{P}$ where always $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$, a reduction pair like this leads to an infinite $\succsim$ reduction. But every time a rule in $\mathcal{R}_1$ is used in the reduction $t_i \Rightarrow^*_{\mathcal{R},in} s_{i+1}$, then by monotonicity of $\succ$, we must have that $t_i \succ s_{i+1}$. The same holds in a formative reduction if any rule in $\mathcal{R}_1^+$ is used (if $l \succ r$, then $l' \succ r'$ for any rule $l' \Rightarrow r'$ generated from $l \Rightarrow r$, since by the definition of a strong reduction pair, $\succ$ is monotonic). Thus, if this happens infinitely often, we have a contradiction to well-foundedness of $\succ$. As before, we also obtain a contradiction if a dependency pair in $\mathcal{P}_1$ is used infinitely often.

We conclude that dependency pairs in $\mathcal{P}_1$ and rules in $\mathcal{R}_1$ can be used only so often; the dependency chain must have a tail which uses only pairs in $\mathcal{P}_2$ and rules in $\mathcal{R}_2$. Thus we obtain soundness. □

To demonstrate how the framework can be used, let us reconsider the termination of `twice`, now from the different perspective using dependency pair problems and processors!

Example 7.20. Recall the original definition of the `twice` system, with rules:

$$\begin{array}{rcl} \mathtt{I}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\ \mathtt{I}(\mathtt{s}(X)) & \Rightarrow & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\ \mathtt{twice}(F) & \Rightarrow & \lambda y.F \cdot (F \cdot y) \end{array}$$

The dependency pairs of (the β-saturated version of) $\mathcal{R}$ are:

$$\begin{array}{llcl} 1. & \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X \\ 2. & \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}^\sharp(\lambda x.\mathtt{I}(x)) \\ 3. & \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{I}^\sharp(x) \end{array} \qquad \begin{array}{llcl} 4. & \mathtt{twice}^\sharp(F) & \Rrightarrow & F \cdot (F \cdot y) \\ 5. & \mathtt{twice}^\sharp(F) & \Rrightarrow & F \cdot y \\ 6. & \mathtt{twice}(F) \cdot X & \Rrightarrow & F \cdot (F \cdot X) \\ 7. & \mathtt{twice}(F) \cdot X & \Rrightarrow & F \cdot X \end{array}$$

By Theorem 7.12: $\mathcal{R}$ *is terminating if the dependency pair problem* $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ *is finite.*

We first use the formative rules processor on the dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$. By Example 6.39 $\mathcal{R}_2 := FR(\mathcal{P}, \mathcal{R})$ consists of:

$$\begin{array}{rcl} \mathtt{I}(\mathtt{s}(X)) & \Rightarrow & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X) \\ \mathtt{twice}(F) \cdot X & \Rightarrow & F \cdot (F \cdot X) \end{array}$$

Thus, $\mathcal{R}$ *is terminating if the DP problem* $(\mathcal{P}, \mathcal{R}_1, \texttt{minimal}, \texttt{formative})$ *is finite.*

We now use the reduction pair processor, with the reduction pair from Example 6.37. This allows us to replace the dependency pair problem $(\mathcal{P}, \mathcal{R}_1, \texttt{minimal}, \texttt{formative})$ by $(\mathcal{P}_1, \mathcal{R}_1, \texttt{minimal}, \texttt{formative})$, where $\mathcal{P}_1$ consists of the dependency pairs $4, 5, 6, 7$. Thus:

$\mathcal{R}$ *is terminating if the DP problem* $(\mathcal{P}_1, \mathcal{R}_1, \texttt{minimal}, \texttt{formative})$ *is finite.*

Let us use the formative rules processor again. As we saw in Example 6.37, $FR(\mathcal{P}_1, \mathcal{R}_1) = \emptyset$. Thus we have:

$\mathcal{R}$ *is terminating if the DP problem* $(\mathcal{P}_1, \emptyset, \texttt{minimal}, \texttt{formative})$ *is finite.*

Finally, using the reduction pair processor with $(\succsim, \succ)$ from Example 6.47, we can map $(\mathcal{P}_1, \emptyset, \texttt{minimal}, \texttt{formative})$ to $\{(\emptyset, \emptyset, \texttt{minimal}, \texttt{formative})\}$. We find:

$\mathcal{R}$ *is terminating if the DP problem* $(\emptyset, \emptyset, \texttt{minimal}, \texttt{formative})$ *is finite.*

Using the empty set processor we obtain:

$\mathcal{R}$ *is terminating.*

In this example, we are always working with one single dependency pair problem, which is iteratively simplified either in the rules or in the dependency pair component. In the rest of the chapter we will see several more processors, which naturally allows for more interesting proofs.

7.3 Optimising Collapsing Dependency Pairs

If we consider the way collapsing dependency pairs are used in dependency chains, there is room for some optimisations. These results have no counterpart in first-order rewriting, because the dependency pairs in question do not occur in the first-order setting.

7.3.1 Extending Meta-variable Applications

The first optimisation is in particular useful in systems where meta-variables do not take arguments, so where collapsing dependency pairs have the form $l^\sharp \Rrightarrow F \cdot p_1 \cdots p_n$.

Consider a dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$. For a given i, suppose $\rho_i = l^\sharp \Rrightarrow p\ (A)$ and $p = F(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m$ with $m > n$, and $s_i = l^\sharp\gamma,\ t_i = p\gamma$. If $F(p_1, \ldots, p_n)\gamma$ is an abstraction, then by definition of $\Rightarrow_{\mathcal{R},in}$, the next step must necessarily be a $\texttt{beta}$-step.

So let us abuse notation a little, and write $F(p_1, \ldots, p_m)$ for the right-hand side of this dependency pair, to reflect the intuitive idea that F may "eat" all p_j, if it can. Formally, this is defined as follows:

Definition 7.21 (Extended meta-variable applications)**.** An extended meta-term is defined by the clauses from Definition 2.2, and additionally:

$$F(s_1, \ldots, s_m) : \tau \text{ if } F : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \sigma_{n+1} \to \ldots \to \sigma_m \to \tau$$
$$\text{and } s_1 : \sigma_1, \ldots, s_m : \sigma_m$$

The definition of substitution for extended meta-terms includes, in addition to the normal clauses:

$$F(s_1,\dots,s_m)\gamma = q[x_1 := s_1\gamma,\dots,x_m := s_m\gamma] \text{ if } \gamma(F) = \lambda x_1 \dots x_m.q$$
$$F(s_1,\dots,s_m)\gamma = q[x_1 := s_1\gamma,\dots,x_k := s_k\gamma] \cdot s_{k+1}\gamma \cdots s_m\gamma \text{ if}$$
$$\gamma(F) = \lambda x_1 \dots x_k.q \text{ with } k < m \text{ and } q \text{ not an abstraction}$$

Theorem 7.22 (Extended Meta-application Processor)**.** *The processor which maps a problem $(\mathcal{P},\mathcal{R},m,f)$ to $\{(\mathcal{P}',\mathcal{R},m,f)\}$ is sound, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ by replacing all dependency pairs of the form $l \Rrightarrow F(p_1,\dots,p_n)\cdot p_{n+1}\cdots p_m\ (A)$ by $l \Rrightarrow F(p_1,\dots,p_m)\ (A)$.*

Proof. Towards a contradiction, suppose that there is a dependency chain $C = [(\rho_i,s_i,t_i) \mid i \in \mathbb{N}]$ with all $\rho_i \in \mathcal{P} \cup \{\mathtt{beta}\}$. We construct a dependency chain (which is minimal and/or formative if the original is) as follows.

Let $t'_{-1} := t_0$. Given $j \in \mathbb{N}$, we assume that we have already chosen (ρ'_0,s'_0,t'_0), $\dots$, $(\rho'_{j-1},s'_{j-1},t'_{j-1})$, and that we have some i such that $t'_{j-1} = t_{i-1}$.

If ρ_i is `beta` or a non-collapsing dependency pair, then let $\rho'_j,s'_j,t'_j := \rho_i,s_i,t_i$. Do the same if $\rho_i = l \Rrightarrow p\ (A)$ and p has the form $F(u_1,\dots,u_n)$. The result still satisfies all requirements of a (minimal, formative) dependency chain, and uses dependency pairs in $\mathcal{P}'$.

If, however, $\rho_i = l \Rrightarrow p\ (A)$ and $p = F(u_1,\dots,u_n)\cdot u_{n+1}\cdots u_m$ with $m > n$, then choose $\rho'_j := l \Rrightarrow F(u_1,\dots,u_m)$ and $s'_j := s_i$. Consider the substitution γ given by case 2 of Definition 6.23.

If $\gamma(F) = \lambda x_1 \dots x_m.q$, then t_i is a β-redex. $\rho_{i+1},\dots,\rho_{i+m-n}$ must all be `beta`, and a subterm-step may only be done in step $i+m-n$ (because before then, the β-redex is not at the top). Then, $t_{i+m-n} = v^\sharp[x_m := u_m\gamma]$, where v is a subterm of $q[x_1 := u_1\gamma,\dots,x_{m-1} := u_{m-1}\gamma]$ which contains x_m. Since the $u_i\gamma$ do not contain x_m (we can make sure of this by α-conversion), v must have the form $w[x_1 := u_1\gamma,\dots,x_{m-1} := u_{m-1}\gamma]$, where w is a subterm of q containing x_m. But then $t_{i+m-n} = w^\sharp[\vec{x} := \vec{u}\gamma]$, and we can choose $t'_j := t_{i+m-n}$. This gives a valid step by case 2d of the definition of a dependency chain.

Alternatively, if $\gamma(F) = \lambda x_1 \dots x_k.q$ with $n \le k < m$, then $\rho_{i+1},\dots,\rho_{i+k-n}$ are all `beta`, but in none of these steps a subterm step is done, because the β-redex is not at the top. Thus, $t_{i+k-n} = F(p_1,\dots,p_m)\gamma$, so we can safely choose $t'_j := t_{i+k-n}$. □

Note that this processor is *sound*, but not *complete*: we cannot necessarily convert a chain with dependency pairs $l \Rrightarrow F(p_1,\dots,p_m)$ back to $l \Rrightarrow F(p_1,\dots,p_n)\cdot p_{n+1}\cdots p_m$. For example, consider the set $\mathcal{P}$ of dependency pairs consisting of:

$$\begin{array}{rcl} \mathtt{f}^\sharp(F,X) & \Rrightarrow & F\cdot\mathtt{a}\cdot X \\ \mathtt{g}^\sharp(\mathtt{a}) & \Rrightarrow & \mathtt{f}^\sharp(\lambda xy.\mathtt{g}(x),\mathtt{a}) \end{array}$$

And let $\mathcal{R} = \{\mathtt{f}(F,X) \Rightarrow F\cdot\mathtt{a}\cdot X,\ \mathtt{g}(\mathtt{a}) \Rightarrow \mathtt{f}(\lambda xy.\mathtt{g}(x),\mathtt{a})\}$.

There is no infinite $(\mathcal{P},\mathcal{R},m,f)$-chain. This is because any infinite dependency chain on these pairs would have to use the second pair at some point.

Say $\rho_i = \mathtt{g}^\sharp(\mathtt{a}) \Rrightarrow \mathtt{f}^\sharp(\lambda xy.\mathtt{g}(x), \mathtt{a})$. Then $s_{i+1} = \mathtt{f}^\sharp(\lambda xy.\mathtt{g}(x), \mathtt{a})$ and $t_{i+1} = (\lambda xy.\mathtt{g}(x)) \cdot \mathtt{a} \cdot \mathtt{a}$. Because of the form, ρ_{i+2} must be `beta`, and $t_{i+2} = (\lambda y.\mathtt{g}(\mathtt{a})) \cdot \mathtt{a}$. But now we get in trouble, because due to the form ρ_{i+3} can only be `beta`, but neither case 3 or case 3b of Definition 6.23 applies.

On the other hand, the corresponding set of dependency pairs $\mathcal{P}'$,

$$\begin{array}{rcl} \mathtt{f}^\sharp(F, X) & \Rrightarrow & F(\mathtt{a}, X) \\ \mathtt{g}^\sharp(\mathtt{a}) & \Rrightarrow & \mathtt{f}^\sharp(\lambda xy.\mathtt{g}(x), \mathtt{a}) \end{array}$$

does admit an infinite $(\mathcal{P}', \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$-chain, with $s_1 = \mathtt{f}^\sharp(\lambda xy.\ \mathtt{g}(x))$, $s_2 = \mathtt{g}^\sharp(\mathtt{a})$, $s_3 = s_1$ and so on.

However, we do know that the dependency pair $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, m, f)$ is (in-)finite if and only if $(\mathsf{DP}(\mathcal{R})', \mathcal{R}, m, f)$ is. This holds by the arguments of Lemma 6.27, combined with the observation that $F(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m\gamma$ always β-reduces to $F(s_1, \ldots, s_m)\gamma$: if $\mathsf{DP}(\mathcal{R})'$ admits a dependency chain, then there is an infinite reduction, so by Theorem 7.12 (and the observation that dropping minimality and formative flags is sound) the DP problem problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, m, f)$ is finite.

The processor of Theorem 7.22 may make it easier to prove finiteness of a DP problem using polynomial interpretations. All definitions of Chapter 4 go through whether we work with extended meta-terms or normal meta-terms, and $[\![F(p_1, \ldots, p_m)]\!]_{\mathcal{J},\alpha}$ is likely to be smaller than $[\![F(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m]\!]_{\mathcal{J},\alpha}$. HORPO cannot deal with extended meta-terms, but since always $(F(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m)\gamma \to_\beta^* F(p_1, \ldots, p_m)\gamma$, we could simply transform any extended meta-terms back to normal meta-terms before applying HORPO.

Example 7.23. Consider the following toy system:

$$\begin{array}{rcl} \mathtt{f}(F, X, \mathtt{0}) & \Rightarrow & \mathtt{0} \\ \mathtt{f}(F, X, \mathtt{s}(Y)) & \Rightarrow & \mathtt{g}(Y, \mathtt{either}(Y, F \cdot X)) \\ \mathtt{g}(X, Y) & \Rightarrow & \mathtt{f}(\lambda x.\mathtt{s}(\mathtt{0}), Y, X) \\ \mathtt{either}(X, Y) & \Rightarrow & X \\ \mathtt{either}(X, Y) & \Rightarrow & Y \end{array}$$

To prove termination of this system (which originates from an AFS, so only uses meta-variables without arguments), we must prove that the DP problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is finite, where $\mathcal{P}$ consists of:

$$\begin{array}{rcl} \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rrightarrow & \mathtt{g}^\sharp(Y, \mathtt{either}(Y, F \cdot X)) \\ \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rrightarrow & \mathtt{either}^\sharp(Y, F \cdot X) \\ \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rrightarrow & F \cdot X \\ \mathtt{g}^\sharp(X, Y) & \Rrightarrow & \mathtt{f}^\sharp(\lambda x.\mathtt{s}^-(\mathtt{0}), Y, X) \end{array}$$

Using the extended meta-application processor, we can replace the right-hand side of the collapsing dependency pair by $F(X)$, and by the formative rules processor we can throw away all rules except the two `either` rules. Thus, $\mathcal{R}$ is

terminating if $(\mathcal{P}_{ex}, \mathcal{R}_{form}, \texttt{minimal}, \texttt{formative})$ is finite, where:

$$\mathcal{P}_{ex} = \left\{ \begin{array}{rcl} \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rightarrow & \mathtt{g}^\sharp(Y, \mathtt{either}(Y, F \cdot X)) \\ \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rightarrow & \mathtt{either}^\sharp(Y, F \cdot X) \\ \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rightarrow & F \cdot X \\ \mathtt{g}^\sharp(X, Y) & \Rightarrow & \mathtt{f}^\sharp(\lambda x.\mathtt{s}(\mathtt{0}), Y, X) \end{array} \right\}$$

$$\mathcal{R}_{form} = \left\{ \begin{array}{rcl} \mathtt{either}(X, Y) & \Rightarrow & X \\ \mathtt{either}(X, Y) & \Rightarrow & Y \end{array} \right\}$$

We use the reduction pair processor (with a tagged reduction pair, since $\mathcal{P}$ and $\mathcal{R}$ are both abstraction-simple). Consider the polynomial interpretation with:

$$\begin{array}{rclcrcl} \mathcal{J}(\mathtt{f}^\sharp) & = & \boldsymbol{\lambda} fnm.f(n) + m + 1 & & \mathcal{J}(\mathtt{0}) & = & 0 \\ \mathcal{J}(\mathtt{g}^\sharp) & = & \boldsymbol{\lambda} nm.n + 1 & & \mathcal{J}(\mathtt{either}) & = & \boldsymbol{\lambda} nm.n + m \\ \mathcal{J}(\mathtt{either}^\sharp) & = & \boldsymbol{\lambda} nm.0 & & \mathcal{J}(\mathtt{s}) & = & \boldsymbol{\lambda} n.n \\ & & & & \mathcal{J}(\mathtt{s}^-) & = & \boldsymbol{\lambda} n.n \end{array}$$

With these interpretations, the rules are easily oriented with $\succsim$, and $\mathcal{J}(\mathtt{s}^-) \sqsupseteq \mathcal{J}(\mathtt{s})$. Moreover, we can orient the dependency pairs; writing $\alpha(F) = f,\ \alpha(X) = n,\ \alpha(Y) = m$, we have:

- $[\![\mathtt{f}^\sharp(F, X, \mathtt{s}(Y))]\!]_{\mathcal{J},\alpha} = f(n)+m+1 \geq m+1 = [\![\mathtt{g}^\sharp(Y, \mathtt{either}(Y, F \cdot X))]\!]_{\mathcal{J},\alpha}$;
- $[\![\mathtt{f}^\sharp(F, X, \mathtt{s}(Y))]\!]_{\mathcal{J},\alpha} = f(n) + m + 1 \geq 0 = [\![\mathtt{either}^\sharp(Y, F \cdot X)]\!]_{\mathcal{J},\alpha}$;
- $[\![\mathtt{f}^\sharp(F, X, \mathtt{s}(Y))]\!]_{\mathcal{J},\alpha} = f(n) + m + 1 > f(n) = [\![F(X)]\!]_{\mathcal{J},\alpha}$;
- $[\![\mathtt{g}^\sharp(X, Y)]\!]_{\mathcal{J},\alpha} = n + 1 \geq n + 1 = [\![\mathtt{f}^\sharp(\lambda x.\mathtt{s}^-(\mathtt{0}), Y, X)]\!]_{\mathcal{J},\alpha}$.

Thus, we can remove the collapsing dependency pair. We could not have used this interpretation without changing the dependency pair, because then the right-hand side of the collapsing pair would have been interpreted to $f(n) + n$.

Although this modification of dependency pairs adds relatively little power (testing with WANDA shows no difference on the termination problem database), it has the advantage of being exceedingly easy to implement: when finding polynomial interpretations for dependency pairs, we merely treat the right-hand side of a constraint $l \mathrel{(\succsim)} F \cdot \vec{p}$ as a meta-variable application. This often leads to easier polynomials, and avoids using the (inefficient) $\max$ function for application.

7.3.2 Adding Meta-variable Restrictions

The next modification for collapsing dependency pairs is designed with an eye on the *dependency graph*, which will be introduced for higher-order rewriting in Section 7.4. This transformation is based on the following observation: when

a dependency pair $l^\sharp \Rrightarrow F(p_1, \ldots, p_m)\ (A)$ is used somewhere in a minimal dependency chain, say at place ρ_i, the right-hand side is probably substituted by a term which does not "eat" all of the p_i. For if it was, then t_i would be the marked version of a strict subterm of s_i.

Theorem 7.24. *The processor which maps a DP problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, f)$ to $(\mathcal{P}', \mathcal{R}, \mathtt{minimal}, f)$ provided the conditions below are satisfied, is both sound and complete:*

- *if a term s is terminating under $\Rightarrow_\mathcal{R}$, then there is no $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, f)$-chain which starts in $s^\sharp$ or any of its subterms;*[3]
- *$\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$, where $\mathcal{P}_2$ contains only collapsing dependency pairs of the form $l \Rrightarrow F(p_1, \ldots, p_n)\ (A)$;*
- *$\mathcal{P}' = \mathcal{P}_1 \cup \{l \Rrightarrow F(p_1, \ldots, p_m)\ (A \cup \{F : i\}) \mid l \Rrightarrow F(p_1, \ldots, p_m) \in \mathcal{P}_2,\ 1 \leq i \leq m\}$;*

Proof. Whenever a dependency pair from $\mathcal{P}_2$ is used somewhere in a minimal formative (tagged) dependency chain, it cannot be the case that the meta-variable eats *all* its arguments: if so, $s_i \rhd t_i$, and by minimality of the chain, this means t_i is terminating under $\Rightarrow_\mathcal{R}$. This gives a contradiction with the first assumption of the lemma. Thus, the dependency pair must use at least one of its arguments, so at least one of the new dependency pairs is applicable.

For completeness, note that every one of the new dependency pairs could be replaced by its original in a dependency chain without altering the s_i, t_i. □

This transformation is in particular useful for collapsing dependency pairs where the meta-variable has only one argument: we can simply add the requirement that the argument actually occurs in the meta-variable, a requirement that might help remove some edges in the dependency graph.

Example 7.25. Consider the AFSM with two rules:

$$\begin{array}{rcl} \mathtt{f}(\mathtt{0}) & \Rightarrow & \mathtt{g}(\lambda x.\mathtt{0}, \mathtt{1}) \\ \mathtt{g}(F, X) & \Rightarrow & F \cdot \mathtt{f}(X) \end{array}$$

Initially, this system gives the following dependency pairs:

$$\begin{array}{rcl} \mathtt{f}^\sharp(\mathtt{0}) & \Rrightarrow & \mathtt{g}^\sharp(\lambda x.\mathtt{0}, \mathtt{1}) \\ \mathtt{g}^\sharp(F, X) & \Rrightarrow & F \cdot \mathtt{f}(X) \\ \mathtt{g}^\sharp(F, X) & \Rrightarrow & \mathtt{f}^\sharp(X) \end{array}$$

[3] This is the case if $\mathcal{P} \subseteq \mathsf{DP}(\mathcal{R})$. The property is also preserved by the extended meta-application processor.

The AFSM is terminating if $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \texttt{minimal}, \texttt{formative})$ is a finite DP problem. By Theorems 7.22 and 7.24 this is the case if $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{formative})$ is finite, where $\mathcal{P}$ consists of:

$$\begin{array}{rcl} \mathtt{f}^\sharp(\mathtt{0}) & \Rrightarrow & \mathtt{g}^\sharp(\lambda x.\mathtt{0}, \mathtt{1}) \\ \mathtt{g}^\sharp(F, X) & \Rrightarrow & F(\mathtt{f}(X))\ \{F : 1\} \\ \mathtt{g}^\sharp(F, X) & \Rrightarrow & \mathtt{f}^\sharp(X) \end{array}$$

In Example 7.33 we will see how this transformation of $\mathcal{P}$ helps us.

7.3.3 General Split Transformation

A last transformation on collapsing dependency pairs that might be useful is to split a dependency pair which uses a meta-variable with arguments in two cases: the argument occurs in the substitute, or it does not. This way, we could for instance replace one dependency pair

$$\mathtt{f}^\sharp(\lambda xy.F(x, y)) \Rrightarrow F(\mathtt{f}(\lambda xy.x), \mathtt{0})\ \{F : 2\}$$

by the two new dependency pairs:

$$\begin{array}{rcll} \mathtt{f}^\sharp(\lambda xy.F(x, y)) & \Rrightarrow & F(\mathtt{f}(\lambda xy.x), \mathtt{0}) & \{F : 1,\ F : 2\} \\ \mathtt{f}^\sharp(\lambda xy.F(y)) & \Rrightarrow & F(\mathtt{0}) & \{F : 1\} \end{array}$$

Since every application of the original dependency pair could be transformed into one of the new pairs, this transformation of a dependency pair is sound. Because every application of one of the new pairs could be transformed into an application of the original pair, it is complete.

Phrasing this in the most general way possible, we obtain the following result:

Theorem 7.26. *The processor which maps a DP problem $(\mathcal{P} \cup \{\rho\}, \mathcal{R}, m, f)$ to $(\mathcal{P} \cup \{\rho_1, \ldots, \rho_n\}, \mathcal{R}, m, f)$ provided the conditions below are satisfied, is both sound and complete:*

- *every occurrence of ρ in a $(\mathcal{P}, \mathcal{R}, m, f)$-chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ can be replaced by one of the ρ_i without changing s_i and t_i;*
- *every occurrence of some ρ_j can be replaced by ρ without affecting s_i and t_i.*

Proof. This obviously holds; the proof is in the conditions. □

Note that it is also obvious that the processor which replaces $\mathcal{P} \cup \{\rho\}$ by $\mathcal{P} \cup \{\rho_1, \ldots, \rho_n\}$ if only the first of these constraints is satisfied is still sound, even if it is not complete.

Theorems 7.24 and 7.26 will primarily be useful in combination with the dependency graph, which we will study now.

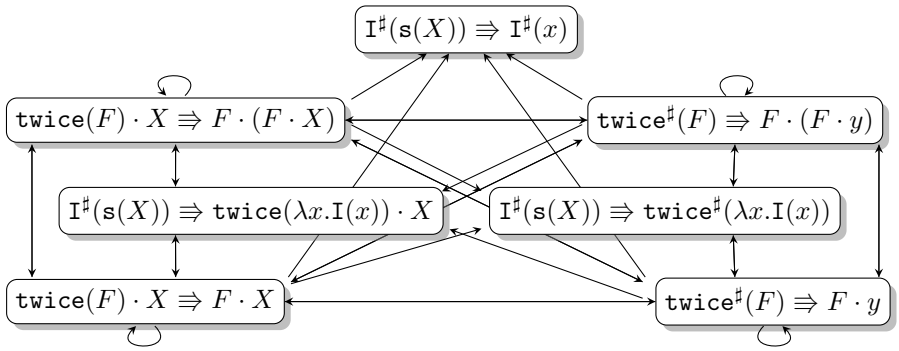
7.4 The Dependency Graph

The dependency graph, which provides a powerful dependency pair processor in the first-order dependency pair framework, can also be extended to our setting. The notions are very similar to the first-order case; however, in the presence of collapsing dependency pairs, the graph is likely to be quite dense.

Definition 7.27. The *dependency graph* of a pair $(\mathcal{P}, \mathcal{R})$ is a graph with the pairs in $\mathcal{P}$ as nodes, and an edge from node $l \Rrightarrow p\ (A)$ to node $l' \Rrightarrow p'\ (B)$ if either p is headed by a meta-variable application, $p = Z(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ with $m \geq n$ and $m > 0$, or there is a substitution γ which respects A and a substitution δ which respects B, such that $p\gamma \Rightarrow^*_{\mathcal{R},in} l'\delta$.

Here, we say γ respects *A if for all conditions $Z : i \in A$ we have $\gamma(Z) = \lambda x_1 \ldots x_n.s$ with $x_i \in FV(s)$.*

Example 7.28. The dependency graph of $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$ for the AFS `twice`:



A *cycle* is a set C of dependency pairs such that between every two pairs $\rho, \pi \in C$ there is a non-empty path in the graph using only nodes in C. A cycle that is not contained in any other cycle is called a *strongly connected component* (SCC).

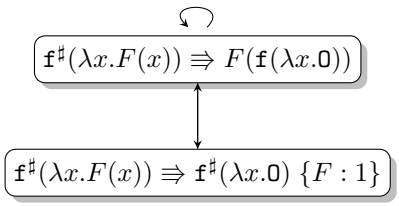
Comment: The requirement to add an edge from any collapsing node to all nodes in the graph is necessary by clauses 2d and 3b in Definition 6.23: a dependency chain could for instance have a dependency pair of the form $l \Rrightarrow F \cdot r$ followed by `beta` with a subterm step, and then any other dependency pair. Hence a collapsing rule may give rise to many cycles. If the system originates from a formalism without an application symbol, such as the IDTSs from Chapter 3.2, then this requirement may be weakened. For example, in the example used in Chapter 3.2, it would not be necessary to draw an edge from the dependency pair $\mathtt{f}^\sharp(\mathtt{a}, \mathtt{g}(\lambda xy.F(x,y))) \Rrightarrow F(\mathtt{a}, \mathtt{g}(\lambda xy.F(x,y)))$ to itself, since an application-free term of type `bool` in this system cannot have a subterm of type `nat`.

Because the dependency graph cannot be computed in general, it is common to use *approximations* of the dependency graph, which have the same nodes but possibly more edges. Approximations also make it possible to deal with infinite systems, without using an infinite graph.

Definition 7.29 (Graph Approximation)**.** A finite graph G *approximates* the dependency graph A if for every node ρ in A there is a node n_ρ in G, and whenever there is an edge from ρ to π in A, then there is an edge in G from n_ρ to n_π.

Consequently, all dependency graphs can be approximated with the trivial graph to the right. However, in finite systems, approximations typically have one node for every dependency pair.

A brute method to find an approximation of the dependency graph is to have an edge between $l \Rrightarrow p\ (A)$ and $l' \Rrightarrow p'\ (B)$ if $l \Rrightarrow p$ is collapsing, or if p and l' both have the form $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ for the same function symbol f and some $m \geq n \geq 0$.



However, there are more sophisticated methods, for example by making use of the meta-variable conditions which come with all dependency pairs, and which can be strengthened using the methods of Section 7.3. Consider for instance a system with a rule $\mathtt{f}(\lambda x.F(x)) \Rightarrow F(\mathtt{f}(\lambda x.\mathtt{0}))$. If we take into account that $\lambda x.\mathtt{0}$ cannot reduce to something of the form $\lambda x.s$ with $x \in FV(s)$, we get the dependency graph on the left, where the second dependency pair notably does not have a self-loop. Furthermore, first-order methods (see e.g. [119]) for finding good approximations may be extended. Some simple tricks will be mentioned in Chapter 8.4.

The dependency graph is useful because of the following observation:

Lemma 7.30. *Let $\mathcal{P}$ be a set of dependency pairs, $\mathcal{R}$ a set of rules, and G an approximation of the dependency graph of $(\mathcal{P}, \mathcal{R})$. If there is a dependency chain over $\mathcal{P}$ and $\mathcal{R}$, then it has a tail which uses only the dependency pairs in some strongly connected component of G.*

Proof. Given any dependency chain, note that, for any k, m: *(**) if $k < m$ then there is a path in G from n_{ρ_k} to n_{ρ_m}* (if $\rho_k, \rho_m \neq \mathtt{beta}$).

Since the graph approximation G is finite, but an infinite chain uses infinitely many dependency pairs, every infinite chain must visit at least one node d infinitely often. Let i, j be numbers such that $i < j$ and $n_{\rho_i} = n_{\rho_j} = d$. As there is a path in G from d to itself by (**), d is on a cycle; let C be the SCC containing d. Then for all ρ_k with $k > i$ and $\rho_k \neq \mathtt{beta}$ we must have that $n_{\rho_k} \in C$: there is a path in G from d to n_{ρ_k} by (**) because $i < k$, and since d occurs infinitely often, so also some $n_{\rho_m} = d$ for $m > k$, there must be a path from n_{ρ_k} to d as well. But then, the tail $\{\rho_j \mid j \geq i\}$ is a dependency chain in C. □

This leads to the following result:

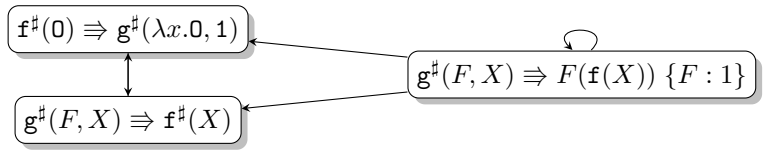
Theorem 7.31 (Dependency Graph Processor)**.** *The processor which maps a DP problem $(\mathcal{P}, \mathcal{R}, m, f)$ to $\{(C, \mathcal{R}, m, f) \mid C$ is an SCC of a dependency graph approximation of $(\mathcal{P}, \mathcal{R})\}$ is both sound and complete.*

Example 7.32. We consider the dependency pair problem $(\mathsf{DP}(\mathcal{R}_{\mathtt{twice}}), \mathcal{R}_{\mathtt{twice}}, \mathtt{minimal}, \mathtt{formative})$. The dependency graph (approximation) of this problem, which we saw in Example 7.28, has only one SCC C, consisting of:

$$\begin{array}{lcl@{\qquad}rcl} \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}(\lambda x.\mathtt{I}(x)) \cdot X & \mathtt{twice}^\sharp(F) & \Rrightarrow & F \cdot (F \cdot y) \\ \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}^\sharp(\lambda x.\mathtt{I}(x)) & \mathtt{twice}^\sharp(F) & \Rrightarrow & F \cdot y \\ & & & \mathtt{twice}(F) \cdot X & \Rrightarrow & F \cdot (F \cdot X) \\ & & & \mathtt{twice}(F) \cdot X & \Rrightarrow & F \cdot X \end{array}$$

Therefore `twice` is terminating if the DP problem $(C, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is finite.

Example 7.33. The system from Example 7.25 is terminating if the dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is terminating, where $\mathcal{R} = \{\mathtt{f}(\mathtt{0}) \Rightarrow \mathtt{g}(\lambda x.\mathtt{0}, \mathtt{1}),\ \mathtt{g}(F, X) \Rightarrow F \cdot \mathtt{f}(X)\}$, and $\mathcal{P}$ consists of the dependency pairs in the following graph:



The edges from the collapsing dependency pair to the other two and itself are necessary by definition of the dependency graph. The edges between the other two are necessary because the right-hand side of either can be instantiated to the left-hand side of the other. There is no edge from $\mathtt{g}^\sharp(F, X) \Rrightarrow \mathtt{f}^\sharp(X)$ to the collapsing pair because the root-symbols do not match.

Importantly, there is also no edge from $\mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{0}, \mathtt{1})$ to the collapsing dependency pair. This is because of the meta-variable restriction $F : 1$, since $\lambda x.\mathtt{0}$ ignores its first argument. Thus, because of the modifications of Theorems 7.22 and 7.24, the collapsing dependency pair has no incoming edges.

Consequently, there are two strongly connected components in this graph. The dependency graph processor maps $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ to the set $\{(\mathcal{P}_1, \mathcal{R}, \mathtt{minimal}, \mathtt{formative}),\ (\mathcal{P}_2, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})\}$, where $\mathcal{P}_1 = \{\mathtt{g}^\sharp(F, X) \Rrightarrow F(\mathtt{f}(X))\}$ and $\mathcal{P}_2 = \{\mathtt{f}^\sharp(\mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(\lambda x.\mathtt{0}, \mathtt{1}),\ \mathtt{g}^\sharp(F, X) \Rrightarrow \mathtt{f}^\sharp(X)\}$. Thus, the AFSM is finite if both these dependency pair problems are finite. Using the formative rules processor on the first of these, we obtain:

The AFSM from Example 7.25 is terminating if the dependency pair problems $(\mathcal{P}_1, \emptyset, \mathtt{minimal}, \mathtt{formative})$ and $(\mathcal{P}_2, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ are both finite.

Using the reduction pair processor, we quickly eliminate the first DP problem, by considering the polynomial interpretation with $\mathcal{J}(\mathtt{f}) = \boldsymbol{\lambda} n.n$ and $\mathcal{J}(\mathtt{g}^\sharp) = \boldsymbol{\lambda} fn.f(n) + 1$. As for the second, by the reduction pair processor we can replace this DP problem by the simpler $(\{\mathtt{g}^\sharp(F, X) \Rrightarrow \mathtt{f}^\sharp(X)\}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$, provided a reduction pair exists such that:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{f}^\sharp(\mathtt{0}) & \succ & \mathtt{g}^\sharp(\lambda x.\mathtt{0}, \mathtt{1}) & \mathtt{f}(\mathtt{0}) & \succsim & \mathtt{g}(\lambda x.\mathtt{0}, \mathtt{1}) \\ \mathtt{g}^\sharp(F, X) & \succsim & \mathtt{f}^\sharp(X) & \mathtt{g}(F, X) & \succsim & F \cdot \mathtt{f}(X) \end{array}$$

Since $\mathcal{P}$ is non-collapsing, this reduction pair does not need to respect $\rhd^{\mathcal{F}}$ or $\rhd^-$.

Let us use StarHorpo with an argument function:

$$\begin{array}{rcl} \pi(\mathtt{g}^\sharp) & = & \lambda xy.\mathtt{f}^\sharp(y) \\ \pi(\mathtt{g}) & = & \lambda xy.@^{\mathtt{o}\to\mathtt{o}}(x, \mathtt{f}(y)) \end{array}$$

The other symbols are left alone, so e.g. $\pi(\mathtt{f}) = \lambda x.\mathtt{f}(x)$. This gives constraints:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{f}^\sharp(\mathtt{0}) & \succ_\star & \mathtt{f}^\sharp(\mathtt{1}) & \mathtt{f}(\mathtt{0}) & \succeq_\star & @^{\mathtt{o}\to\mathtt{o}}(\lambda x.\mathtt{0}, \mathtt{f}(\mathtt{1})) \\ \mathtt{f}^\sharp(X) & \succeq_\star & \mathtt{f}^\sharp(X) & @^{\mathtt{o}\to\mathtt{o}}(F, \mathtt{f}(X)) & \succeq_\star & @^{\mathtt{o}\to\mathtt{o}}(F, \mathtt{f}(X)) \end{array}$$

Which is satisfied if we choose $\mathtt{f} \blacktriangleright @^{\mathtt{o}\to\mathtt{o}}$ and $\mathtt{0} \blacktriangleright \mathtt{1}$.

Finiteness of the remaining DP problem $(\{\mathtt{g}^\sharp(F, X) \Rrightarrow \mathtt{f}^\sharp(X)\}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is quickly obtained, since its dependency graph has no cycles.

Example 7.34. Consider the system with a single rule $\mathtt{f}(\lambda x.F(x)) \Rightarrow F(\mathtt{f}(\lambda x.\mathtt{0}))$ whose dependency graph was shown in the text. To prove termination, we must show finiteness of the dependency pair problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$. To this end, let us first use the formative rules processor to replace this problem by $(\mathsf{DP}(\mathcal{R}), \emptyset, \mathtt{minimal}, \mathtt{formative})$, and then use the reduction pair processor. Using HOIPO with $\mathtt{f}^\sharp \blacktriangleright \mathtt{f}$ and an argument function which maps $\mathtt{0}$ to $\bot_{\mathtt{nat}}$, we obtain:

$$\begin{array}{ll} & \mathtt{f}^{\sharp\star}(\lambda x.F(x)) \\ \Rightarrow_{\mathtt{select}} & F(\mathtt{f}^{\sharp\star}(\lambda x.F(x))) \\ \Rightarrow_{\mathtt{copy}} & F(\mathtt{f}(\mathtt{f}^{\sharp\star}(\lambda x.F(x)))) \\ \Rightarrow_{\mathtt{abs}} & F(\mathtt{f}(\lambda y.\mathtt{f}^{\sharp\star}(\lambda x.F(x), y))) \\ \Rightarrow_{\mathtt{bot}} & F(\mathtt{f}(\lambda y.\bot_{\mathtt{nat}})) \end{array}$$

$$\begin{array}{ll} & \mathtt{f}^\sharp(\lambda x.F(x)) \\ \Rightarrow_{\mathtt{bot}} & \mathtt{f}^\sharp(\lambda y.\bot_{\mathtt{nat}}) \end{array}$$

Thus, we obtain termination of the system if $(\mathcal{P}, \emptyset, \mathtt{minimal}, \mathtt{formative})$ is a finite dependency pair problem, where $\mathcal{P}$ consists of the single dependency pair $\mathtt{f}^\sharp(\lambda x.F(x)) \Rrightarrow \mathtt{f}^\sharp(\lambda x.\mathtt{0})\ \{F : 1\}$. But now the meta-variable conditions come in! The dependency graph of $(\mathcal{P}, \emptyset)$ has no edges, and thus the dependency graph processor maps $(\mathcal{P}, \emptyset, \mathtt{minimal}, \mathtt{formative})$ to $\emptyset$; termination is proved.

Here, we see a second use for the meta-variable conditions $\{F : i\}$. These conditions were originally introduced primarily for the sake of getting a completeness result (which makes it possible to use the dependency pair framework for non-termination as well as termination, even though that is not done in this work). But as we saw in Example 7.33, they also have a use in the construction of dependency graph approximations.

However, at this point their purpose has mostly been served. We shall not need these conditions for any further transformations of $\mathcal{P}$.

7.5 The Subterm Criterion

As in the first-order case, we may attempt to use the subterm criterion. But here, we run into some trouble. For how should we deal with collapsing dependency pairs? And what about dependency pairs of the form $f(X_1) \cdot X_2 \Rightarrow r\ (A)$, whose root symbol is not marked?

A subterm criterion was defined for the static dependency approach on the HRS formalism in [87]. In this setting, it is very natural, as there are only dependency pairs of the form $f^\sharp(\vec{l}) \Rrightarrow g^\sharp(\vec{r})$. But we will have to adapt the definitions to work with the approach in this thesis.

Unfortunately, the issue of collapsing dependency pairs is very hard, if not impossible, to overcome. However, many higher-order systems still have subsystems which do not use collapsing dependency pairs. We saw this happen in Example 7.33. Also, it is sometimes possible to delete the collapsing dependency pairs with a reduction pair, as was done in Example 7.23. For such non-collapsing sets, the subterm criterion may still be used.

To start, let a set $\mathcal{P}$ of non-collapsing dependency pairs be given, and let $\mathcal{H}$ be the set of function symbols f such that any left- or right-hand side of a dependency pair has the form $f(\vec{s}) \cdot \vec{t}$. A *projection function* for $\mathcal{P}$ is a function ν which assigns to each $f \in \mathcal{H}$ a number i such that for all dependency pairs $l \Rrightarrow p\ (A) \in \mathcal{P}$, the following function $\overline{\nu}$ is well-defined for both l and p:

$$\overline{\nu}(f(s_1, \ldots, s_m) \cdot s_{m+1} \cdots s_n) = s_{\nu(f)}$$

Theorem 7.35 (Subterm Criterion Processor)**.** *The processor which maps a dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, f)$ to $(\mathcal{P}_2, \mathcal{R}, \mathtt{minimal}, f)$ provided the following conditions hold, is both sound and complete.*

- *$\mathcal{P}$ is non-collapsing;*
- *$\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$;*
- *a projection function ν exists such that:*
  - *$\overline{\nu}(l) \rhd \overline{\nu}(p)$ for all $l \Rrightarrow p \in \mathcal{P}_1$, and*
  - *$\overline{\nu}(l) = \overline{\nu}(p)$ for $l \Rrightarrow p \in \mathcal{P}_2$.*

Proof. Completeness holds by Lemma 7.14.

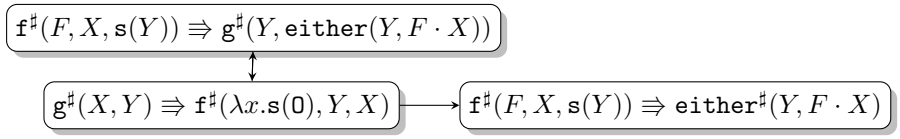
If there is a minimal dependency chain over $\mathcal{P}$, then by Observation III from Section 6.3 we can assume that it does not use `beta` (as $\mathcal{P}$ is non-collapsing). Since the left-hand side of a dependency pair is always a pattern (and therefore its subterms are, too), $\overline{\nu}(l) \rhd \overline{\nu}(p)$ implies $\overline{\nu}(l)\gamma \rhd \overline{\nu}(p)\gamma$. Thus, from such a dependency chain we obtain an infinite reduction: $\overline{\nu}(s_1) \trianglerighteq \overline{\nu}(t_1) \Rightarrow_{\mathcal{R}}^{*} \overline{\nu}(s_2) \trianglerighteq \overline{\nu}(t_2) \Rightarrow_{\mathcal{R}}^{*} \ldots$ Either infinitely many of these steps are dependency pairs in $\mathcal{P}_1$, so take a $\rhd$ step, or a tail of the dependency chain uses only pairs in $\mathcal{P}_2$, so $(\mathcal{P}_2, \mathcal{R}, \mathtt{minimal}, f)$ is not finite. In the latter case we are done, so, towards a contradiction, assume the former.

The relation $\Rightarrow_{\mathcal{R}} \cup \rhd$ is well-founded on the class of terms which are terminating under $\Rightarrow_{\mathcal{R}}$: $\Rightarrow_{\mathcal{R}}$ on its own is well-founded by the class restriction, $\rhd$ on its own is well-founded because it decreases the size of a term, and a reduction using both infinitely often can be rewritten to an infinite $\Rightarrow_{\mathcal{R}}$ reduction by observing that $s \rhd t \Rightarrow_{\mathcal{R}} q$ implies $s \Rightarrow_{\mathcal{R}} C[q] \rhd q$ for some context C. Since we considered a minimal chain, all $\overline{\nu}(s_i)$ and $\overline{\nu}(t_i)$ are minimal, and thus we obtain the required contradiction! □

Example 7.36. Recall the system from Example 7.23. We have seen that this system is terminating if $(\mathcal{P}_{ex}, \mathcal{R}_{form}, \mathtt{minimal}, \mathtt{formative})$ is finite, where $\mathcal{P}_{ex}$ is the set consisting of the non-collapsing dependency pairs:

$$\begin{array}{rcl} \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rrightarrow & \mathtt{g}^\sharp(Y, \mathtt{either}(Y, F \cdot X)) \\ \mathtt{f}^\sharp(F, X, \mathtt{s}(Y)) & \Rrightarrow & \mathtt{either}^\sharp(Y, F \cdot X) \\ \mathtt{g}^\sharp(X, Y) & \Rightarrow & \mathtt{f}^\sharp(\lambda x.\mathtt{s}(\mathtt{0}), Y, X) \end{array}$$

Considering its dependency graph,



we can drop the dependency pair using $\mathtt{either}^\sharp$ with the dependency graph processor. We use the subterm criterion on the remaining two dependency pairs. Choosing $\nu(\mathtt{f}^\sharp) = 3$ and $\nu(\mathtt{g}^\sharp) = 1$ we have:

$$\begin{array}{rcccccl} \overline{\nu}(\mathtt{f}^\sharp(F, X, \mathtt{s}(Y))) & = & \mathtt{s}(Y) & \rhd & Y & = & \overline{\nu}(\mathtt{g}^\sharp(Y, \mathtt{either}(Y, F \cdot X))) \\ \overline{\nu}(\mathtt{f}^\sharp(F, X, \mathtt{s}(Y))) & = & X & = & X & = & \overline{\nu}(\mathtt{f}^\sharp(\lambda x.\mathtt{s}(\mathtt{0}), Y, X)) \end{array}$$

Thus, this system is terminating if the dependency pair problem $(\{\mathtt{g}^\sharp(X, Y) \Rrightarrow \mathtt{f}^\sharp(\lambda x.\mathtt{s}(\mathtt{0}), Y, X)\}, \mathcal{R}_{form}, \mathtt{minimal}, \mathtt{formative})$ is finite. Since the dependency graph of this set has no cycles, this is clearly the case!

7.6 Usable Rules

Although formative rules provide a nice higher-order counterpart of usable rules, it would be even nicer if we had both! True, the proof for the usable rules approach essentially breaks in the presence of collapsing rules and dependency pairs – but, like the subterm criterion, the technique may still have some merit when we are dealing with a set of non-collapsing dependency pairs.

A usable rules approach was defined for static dependency in [114]. But even in this setting, where dependency pairs are always non-collapsing, the usable rules approach doesn't work quite as well as in the first-order case. In particular, rules where the right-hand side has any subterms $x \cdot s$ with x a free variable (in AFSMs, this would be a meta-variable application) cause trouble. The definitions in this section use roughly the same restriction as in [114]. In particular the use of patterns: either all rules are usable, or the *right-hand side* of all usable rules is a pattern. An important difference is that here we use *typed symbols*, as in the definition of formative rules.

Apart from restricting to non-collapsing sets of dependency pairs, the results in this section are only applicable if $\mathcal{R}$ is *finitely branching*. That is, for every term s we assume that there are only finitely many different t such that $s \Rightarrow_{\mathcal{R}} t$. In practice, this is rarely a restriction because for instance finite systems are always finitely branching.

We consider $\mathcal{R}^{++}$, which is the set $\mathcal{R} \cup \{l \cdot Z_1 \cdots Z_n \Rightarrow r \cdot Z_1 \cdots Z_n \mid l \Rightarrow r \in \mathcal{R}$, all Z_i fresh meta-variables and $l \cdot \vec{Z}$ well-typed$\}$. Unlike the previously used $\mathcal{R}^{+}$, this set does include rules $l \cdot \vec{Z} \Rightarrow (\lambda x.r) \cdot \vec{Z}$ if $l \Rightarrow \lambda x.r$ is a rule in $\mathcal{R}$. However, $\mathcal{R}$ does not need to be β-saturated. We will reuse the function $Symb$ from Definition 6.38.

Definition 7.37 (Usable Rules)**.** Let $\langle f, \sigma \rangle \sqsupseteq_{us} A$ denote that there is a rewrite rule $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \Rightarrow r$ in $\mathcal{R}^{++}$ with $r : \sigma$ where $A \in Symb(r)$ or r is not a pattern. The reflexive-transitive closure of $\sqsupseteq_{us}$ is denoted by $\sqsupseteq^*_{us}$. Overloading notation, for a meta-term s let $s \sqsupseteq^*_{us} B$ denote that $Symb(s)$ contains a typed symbol A with $A \sqsupseteq^*_{us} B$, or s is not a pattern.

The set of usable rules of a meta-term s, notation $UR(s)$, consists of those rules $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \Rightarrow r \in \mathcal{R}^{++}$ such that $s \sqsupseteq^*_{us} \langle f, \sigma \rangle$, where $r : \sigma$.

The set of usable rules of a non-collapsing dependency pair $l \Rrightarrow f(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m$ is the union $UR(p_1) \cup \ldots \cup UR(p_m)$. The set of usable rules of a set of dependency pairs, $UR(\mathcal{P})$, is the union $\bigcup_{\rho \in \mathcal{P}} UR(\rho)$ if $\mathcal{P}$ is non-collapsing and just $\mathcal{R}$ otherwise.

This definition should be considered parametrised with the set $\mathcal{R}$. When the set of rules under consideration is not clear from context, we might write $UR(s, \mathcal{R})$ or $UR(\rho, \mathcal{R})$ or $UR(\mathcal{P}, \mathcal{R})$ rather than $UR(s)$, $UR(\rho)$ or $UR(\mathcal{P})$ respectively.

Example 7.38. Consider the following toy AFSM, where all function symbols have output type nat, and $F : \mathtt{nat} \to \mathtt{nat}$:

$$\begin{array}{lcllcl} \mathtt{f(a)} & \Rightarrow & \mathtt{f(b)} & \mathtt{g}(F) & \Rightarrow & F \cdot \mathtt{0} \\ \mathtt{h}(F) & \Rightarrow & \mathtt{f(a)} & \mathtt{i}(F) & \Rightarrow & \mathtt{g}(F) \end{array}$$

With this AFSM, $\langle \mathtt{h}, \mathtt{nat}\rangle \sqsupseteq^*_{us} \langle \mathtt{h}, \mathtt{nat}\rangle, \langle \mathtt{f}, \mathtt{nat}\rangle, \langle \mathtt{a}, \mathtt{nat}\rangle, \langle \mathtt{b}, \mathtt{nat}\rangle$ only. However, $\langle \mathtt{i}, \mathtt{nat}\rangle \sqsupseteq_{us} \langle \mathtt{g}, \mathtt{nat}\rangle$, and because $F \cdot \mathtt{0}$ is not a pattern, $\langle \mathtt{g}, \mathtt{nat}\rangle \sqsupseteq_{us}$ everything. Thus, also $\langle \mathtt{i}, \mathtt{nat}\rangle \sqsupseteq^*_{us} A$ for all typed symbols A.

Example 7.39. Consider an AFSM for list manipulation which has four rules:

$$\begin{array}{rcl} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y)) \\ \mathtt{append}(\mathtt{nil}, X) & \Rightarrow & X \\ \mathtt{append}(\mathtt{cons}(X, Y), Z) & \Rightarrow & \mathtt{cons}(X, \mathtt{append}(Y, Z)) \end{array}$$

To prove termination of this system, we must prove finiteness of (DP, $\mathcal{R}$, minimal, formative), which by the dependency graph processor reduces to the question whether both ($\mathcal{P}_1, \mathcal{R}$, minimal, formative) and ($\mathcal{P}_2, \mathcal{R}$, minimal, formative) are finite. Here,

$$\mathcal{P}_1 = \left\{ \begin{array}{lcl} \mathtt{map}^\sharp(\lambda x.F(x), \mathtt{cons}(X, Y)) & \Rrightarrow & F(X) \\ \mathtt{map}^\sharp(\lambda x.F(x), \mathtt{cons}(X, Y)) & \Rrightarrow & \mathtt{map}^\sharp(\lambda x.F(x), Y)) \end{array} \right\}$$

$$\mathcal{P}_2 = \{\mathtt{append}^\sharp(\mathtt{cons}(X, Y), Z) \Rrightarrow \mathtt{append}^\sharp(Y, Z)\}$$

$\mathcal{P}_1$ contains a dependency pair with $F(X)$ in the right-hand side, so $UR(\mathcal{P}_1, \mathcal{R}) = \mathcal{R}$. For the second DP problem usable rules have a larger effect: $UR(\mathcal{P}_2, \mathcal{R}) = \emptyset$.

This definition of usable rules is very similar to the one for formative rules in Section 6.4. Where formative rules are intuitively the rules which can contribute to the creation of the pattern of a dependency pair, usable rules can be thought of as the rules which some part of the right-hand side of a dependency pair can reduce to.

In Lemma 7.42 we will see that when proving finiteness of a dependency pair problem ($\mathcal{P}, \mathcal{R}$, minimal, f), we can mostly restrict attention to the rules in $UR(\mathcal{P}, \mathcal{R})$. The proof of this, however, is wildly different from the similar result for formative rules, but rather similar to the first-order setting [48, 53] and the static setting for HRSs [114].

To start, we introduce a new symbol $\mathtt{p}_\sigma : [\sigma \times \sigma] \longrightarrow \sigma$ for all types σ, as well as a fresh symbol $\bot_\sigma$ of type σ. The set $\mathcal{C}_\epsilon$ consists of the (infinitely many) rules:

$$\begin{array}{lcl} \mathtt{p}_\sigma(Z_1, Z_2) & \Rightarrow & Z_1 \\ \mathtt{p}_\sigma(Z_1, Z_2) & \Rightarrow & Z_2 \end{array}$$

Let a fixed set of dependency pairs $\mathcal{P}$ be given, in whose usable rules we are interested. We also fix some set $\mathcal{R}$, and assume that $\mathcal{R} \not\subseteq UR(\mathcal{P}, \mathcal{R})$. A *usable*

symbol is any typed symbol A such that for some $l \Rrightarrow g(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m \in \mathcal{P}$ and some i: $p_i \sqsupseteq^*_{us} A$. Obviously, if $\langle f, \sigma \rangle$ is a usable symbol of $\mathcal{P}$, then any rule $f(\vec{s}) \cdot \vec{t} \Rightarrow r : \sigma \in \mathcal{R}^{++}$ is a usable rule of $\mathcal{P}$.

For an idea sketch, a finitely branching term which does not have the form $f(\vec{u}) \cdot \vec{v} : \sigma$ with $\langle f, \sigma \rangle$ a usable symbol, will be encoded by the list of its reducts. For example if $s \Rightarrow_{\mathcal{R}} t_1, \ldots, t_n$, we could replace s by $\mathtt{p}(t_1, \mathtt{p}(t_2, \ldots, \mathtt{p}(t_n, \bot) \ldots))$. Using the $\mathcal{C}_\epsilon$-rules, this term still reduces to everything s reduces to. Doing this replacement everywhere in a term does not affect applicability of usable rules, yet will, in a terminating term, eventually lead to a term which only contains usable symbols. In such a term, only usable rules and $\mathcal{C}_\epsilon$-rules can be applied.

To work! For any terminating term $s : \sigma$, define $\varphi(s)$ as follows:

- $\varphi(s) = \lambda x.\varphi(s')$ if $s = \lambda x.s'$;
- $\varphi(s) = x \cdot \varphi(s_1) \cdots \varphi(s_n)$ if $s = x \cdot s_1 \cdots s_n$;
- $\varphi(s) = f(\varphi(s_1), \ldots, \varphi(s_n)) \cdot \varphi(s_{n+1}) \cdots \varphi(s_m)$ if $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma$ and $\langle f, \sigma \rangle$ is a usable symbol;
- $\varphi(s) = \mathtt{p}_\sigma(f(\varphi(s_1), \ldots, \varphi(s_n)) \cdot \varphi(s_{n+1}) \cdots \varphi(s_n), D_\sigma(\{t \mid s \Rightarrow_{\mathcal{R}} t\}))$ if $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma$ and and $\langle f, \sigma \rangle$ is not a usable symbol;
- $\varphi(s) = D_\sigma(\{t \mid s \Rightarrow_{\mathcal{R}} t\})$ if s does not have one of the forms above;
- in these definitions, $D_\sigma(\emptyset) = \bot_\sigma$, and $D_\sigma(X) = \mathtt{p}_\sigma(\varphi(t), D_\sigma(X \setminus \{t\}))$ if X is non-empty and t lexicographically its smallest element

Note that $\{t \mid s \Rightarrow_{\mathcal{R}} t\}$ is finite by the assumption that $\Rightarrow_{\mathcal{R}}$ is finitely branching, and that this definition is therefore well-defined by induction on s (which was assumed to be a terminating term), ordered with $\Rightarrow_{\mathcal{R}} \cup \rhd$.

Lemma 7.40. *If s is a pattern and γ a substitution whose domain contains only meta-variables (and contains all meta-variables in $FMV(s)$), then $\varphi(s\gamma) \Rightarrow^*_{\mathcal{C}_\epsilon} s\gamma^\varphi$, where $\gamma^\varphi = [Z := \varphi(\gamma(Z)) \mid Z \in dom(\gamma)]$. If all elements of $Symb(s)$ are usable symbols of $\mathcal{P}$, then even $\varphi(s\gamma) = s\gamma^\varphi$.*

Proof. By induction on the form of s, which is obvious when s is an abstraction or variable-headed application. If $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma$, then we are also done with the induction hypothesis if $\langle f, \sigma \rangle$ is usable, and if not (in which case we do not need to consider the "all elements of $Symb(s)$ are usable" case), $\varphi(s\gamma) \Rightarrow_{\mathcal{C}_\epsilon} f(\varphi(s_1), \ldots, \varphi(s_n)) \cdot \varphi(s_{n+1}) \cdots \varphi(s_m)$, which by the induction hypothesis $\Rightarrow^*_{\mathcal{C}_\epsilon} s\gamma^\varphi$. Since s is a pattern, the only remaining case is when $s = Z(x_1, \ldots, x_n)$ with all x_i distinct bound variables. In this case, we can write $\gamma(Z) = \lambda x_1 \ldots x_n.t$, and therefore $\varphi(s\gamma) = \varphi(t) = s\gamma^\varphi$. □

Lemma 7.41. *Suppose that $\mathcal{R} \not\subseteq UR(\mathcal{P})$. If $s \Rightarrow_{\mathcal{R}} t$ with s terminating, then $\varphi(s) \Rightarrow^*_{UR(\mathcal{P})\cup\mathcal{C}_\epsilon} \varphi(t)$.*

Proof. First note that (**) $D_\sigma(X) \Rightarrow^*_{\mathcal{C}_\epsilon} \varphi(u)$ for all $u \in X$. This is obvious with induction on the size of X. The lemma holds by induction on the form of s. If s is an abstraction, or headed by a variable (so the reduction happens in a subterm), this is obvious with the induction hypothesis. If $\varphi(s) = D_\sigma(X)$ or $\mathrm{p}_\sigma(\ldots, D_\sigma(X))$ where $X = \{q \mid s \Rightarrow_{\mathcal{R}} q\}$, we are done by (**). Otherwise, $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma$ and $\varphi(s) = f(\varphi(s_1), \ldots, \varphi(s_n)) \cdot \varphi(s_{n+1}) \cdots \varphi(s_m)$.

If the reduction is done in one of the s_i, we can use the induction hypothesis to obtain the lemma. Otherwise a headmost step is done, so $s = l\gamma \cdot s_{k+1} \cdots s_m$ and $t = r\gamma \cdot s_{k+1} \cdots s_m$. Let $l' := l \cdot Z_{k+1} \cdots Z_m$ and $r' := r \cdot Z_{k+1} \cdots Z_m$ for fresh meta-variables $Z_{k+1}, \ldots, Z_m$. Let $\delta := \gamma \cup [Z_{k+1} := s_{k+1}, \ldots, Z_m := s_m]$. Now, $l' \Rightarrow r' \in \mathcal{R}^{++}$, and because $\langle f, \sigma\rangle$ is a usable symbol of $\mathcal{P}$, even $l' \Rightarrow r' \in UR(\mathcal{P})$.

Using Lemma 7.40, $\varphi(s) = \varphi(l'\delta) \Rightarrow^*_{\mathcal{C}_\epsilon} l'\delta^\varphi \Rightarrow_{UR(\mathcal{P})} r'\delta^\varphi$.

By the assumption that $\mathcal{R} \not\subseteq UR(\mathcal{P})$ we know that r' must be a pattern, and since $\langle f, \sigma\rangle \sqsupseteq_{us} A$ for all $A \in Symb(r')$, Lemma 7.40 provides that $r'\delta^\varphi = \varphi(r'\delta) = \varphi(t)$ as required. □

From this we easily obtain the result which allows us to restrict attention to usable rules when trying to prove absence of a minimal dependency chain:

Lemma 7.42. *Suppose $\mathcal{P}$ is non-collapsing and $\mathcal{R} \not\subseteq UR(\mathcal{P}, \mathcal{R})$ finitely branching. If there is a minimal dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ with all $\rho_i \in \mathcal{P}$, then there is also a dependency chain $[(\rho_i, s'_i, t'_i) \mid i \in \mathbb{N}]$ where for all i: $t'_i \Rightarrow^*_{UR(\mathcal{P},\mathcal{R})\cup\mathcal{C}_\epsilon} s'_{i+1}$.*

Proof. Since $\mathcal{R} \not\subseteq UR(\mathcal{P}, \mathcal{R})$, we know that $\mathcal{P}$ is non-collapsing. Let a minimal dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ be given with all $\rho_i \in \mathcal{P}$. Since $\mathcal{P}$ is non-collapsing, no subterm steps are ever taken.

For all i, let $\rho_i = l_i \Rrightarrow p_i\ (A_i)$ and $s_i = l_i\gamma_i$, $t_i = p_i\gamma_i$. Define $s'_i, t'_i := l_i\gamma_i^\varphi, p_i\gamma_i^\varphi$. Write $p_i = f(u_1, \ldots, u_m) \cdot u_{m+1} \cdots u_n$ and $l_{i+1} = f(v_1, \ldots, v_m) \cdot v_{m+1} \cdots v_n$. By Lemmas 7.40 and 7.41, we then have:

$$\begin{aligned} t'_i &= p_i\gamma_i^\varphi \\ &= f(\varphi(u_1\gamma_i), \ldots, \varphi(u_m\gamma_i)) \cdot \varphi(u_{m+1}\gamma_i) \cdots \varphi(u_n\gamma_i) \\ &\Rightarrow^*_{UR(\rho_{i+1},\mathcal{R})\cup\mathcal{C}_\epsilon} f(\varphi(v_1\gamma_i), \ldots, \varphi(v_m\gamma_i)) \cdot \varphi(v_{m+1}\gamma_i) \cdots \varphi(v_n\gamma_i) \\ &\Rightarrow^*_{\mathcal{C}_\epsilon} l_{i+1}\gamma_i^\varphi \end{aligned}$$

Since $UR(\rho_i, \mathcal{R}) \subseteq UR(\mathcal{P}, \mathcal{R})$, the resulting chain has the required property. □

From this lemma (and Observation III below Definition 6.28) we can easily obtain the following processor:

Theorem 7.43 (Usable Rules Processor)**.** *The processor which maps a dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, f)$ to $\{(\mathcal{P}, UR(\mathcal{P}, \mathcal{R}), \mathtt{arbitrary}, \mathtt{all}) \cup \mathcal{C}_\epsilon\}$ provided $\mathcal{R}$ is finitely branching is sound.*

Note that this processor is not complete, and loses both minimality, and a possible `formative` tag. Although it is not avoidable to lose the `minimal` tag as demonstrated in [119], it is an open question whether it may be possible to preserve formative reductions.

Even supposing it is possible to preserve the formative rules flag, though, the usable rules processor is very lossy. Most importantly, we lose the minimality flag which allows us to use the usable rules processor again. It is likely that there are situations where the ability to permanently remove non-usable rules is worth the loss of the flags. However, in most situations, it will probably be preferable to use the following, alternative processor.

Theorem 7.44 (Reduction Pair Processor with Usable Rules)**.** *The processor which maps a problem* $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, f)$ *to* $(\mathcal{P}_2, \mathcal{R}, \texttt{minimal}, f)$ *provided the conditions below are satisfied, is both sound and complete:*

- $\mathcal{P} = \mathcal{P}_1 \uplus \mathcal{P}_2$*;*
- $\mathcal{R}$ *is finitely branching;*
- *there is a standard reduction pair for* $(\mathcal{P}_1, \mathcal{P}_2, UR(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_\epsilon)$.

Proof. In Lemma 7.42, a $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, f)$-chain is transformed into a $(\mathcal{P}, UR(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_\epsilon, \texttt{minimal}, \texttt{all})$-chain *which uses the same dependency pairs infinite often*. If the latter can use the elements of $\mathcal{P}_1$ only finitely many times (as is demonstrated with this reduction pair), then the same holds for the former. □

The $\mathcal{C}_\epsilon$ rules, although there are infinitely many of them, are trivial for most reduction pairs, especially because the $\mathtt{p}_\sigma$ symbols don't occur in any other rules.

Unfortunately, the restrictions to apply usable rules are fairly strong. We might reasonably wonder whether it is really necessary for the right-hand sides of all usable rules and dependency pairs to be patterns. The following example demonstrates that the requirement cannot be dropped.

Example 7.45. Consider a system with the following rules:

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{f}(F,X,Y) & \Rightarrow & \mathtt{g}(X,Y,F,X) & \mathtt{h}(\mathtt{0}) & \Rightarrow & \mathtt{a}(\mathtt{0}) \\
\mathtt{g}(\mathtt{s}(X),\mathtt{s}(Y),F,Z) & \Rightarrow & \mathtt{g}(X,Y,F,Z) & \mathtt{h}(\mathtt{s}(X)) & \Rightarrow & \mathtt{a}(\mathtt{h}(X)) \\
\mathtt{g}(\mathtt{0},\mathtt{0},F,Z) & \Rightarrow & \mathtt{f}(F,\mathtt{s}(Z),F\cdot\mathtt{h}(Z)) & \mathtt{i}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\
 & & & \mathtt{i}(\mathtt{a}(X)) & \Rightarrow & \mathtt{s}(\mathtt{i}(X))
\end{array}$$

This system admits a minimal dependency chain, constantly iterating the dependency pairs in the set $\mathcal{P}$:

$$\begin{array}{rcl}
\mathtt{f}^\sharp(F,X,Y) & \Rrightarrow & \mathtt{g}^\sharp(X,Y,F,X) \\
\mathtt{g}^\sharp(\mathtt{s}(X),\mathtt{s}(Y),F,Z) & \Rrightarrow & \mathtt{g}^\sharp(X,Y,F,Z) \\
\mathtt{g}^\sharp(\mathtt{0},\mathtt{0},F,Z) & \Rrightarrow & \mathtt{f}^\sharp(F,\mathtt{s}(Z),F\cdot\mathtt{h}(Z))
\end{array}$$

This chain has the form:

$$
\begin{array}{ll}
& \mathtt{f}^\sharp(\lambda x.\mathtt{i}(x), \mathtt{s}^n(\mathtt{0}), \mathtt{s}^n(\mathtt{0})) \\
\Rrightarrow & \mathtt{g}^\sharp(\mathtt{s}^n(\mathtt{0}), \mathtt{s}^n(\mathtt{0}), \lambda x.\mathtt{i}(x), \mathtt{s}^n(\mathtt{0})) \\
\Rrightarrow^* & \mathtt{g}^\sharp(\mathtt{0}, \mathtt{0}, \lambda x.\mathtt{i}(x), \mathtt{s}^n(\mathtt{0})) \\
\Rrightarrow & \mathtt{f}^\sharp(\lambda x.\mathtt{i}(x), \mathtt{s}^{n+1}(\mathtt{0}), (\lambda x.\mathtt{i}(x)) \cdot \mathtt{h}(\mathtt{s}^n(\mathtt{0}))) \\
\Rrightarrow^*_{\mathcal{R}} & \mathtt{f}^\sharp(\lambda x.\mathtt{i}(x), \mathtt{s}^{n+1}(\mathtt{0}), \mathtt{a}^{n+1}(\mathtt{0})) \\
\Rrightarrow^*_{\mathcal{R}} & \mathtt{f}^\sharp(\lambda x.\mathtt{i}(x), \mathtt{s}^{n+1}(\mathtt{0}), \mathtt{s}^{n+1}(\mathtt{0})) \\
\ldots &
\end{array}
$$

Note that we need the i-rules to obtain this dependency chain. Indeed, if these two rules are omitted there is no chain at all, even if we do include the $\mathcal{C}_\epsilon$-rules (this takes some effort to see as none of the reduction pairs we have used so far can handle this system; roughly, the instantiation for $\gamma(F)$ which keeps being passed on unmodified must have the form $\lambda x.s$ where s reduces to $\mathtt{s}^n(\mathtt{0})$ for any n, which is not possible with the given rules). Thus, to obtain Lemma 7.42 it is essential that all rules are usable when the right-hand side of a dependency pair or usable rule is not a pattern.

It is an open question whether we could drop or weaken the right-hand side pattern requirement to obtain some variation of Theorem 7.44. For example, it may be possible to ignore the condition if a reduction pair of a certain form is used. This is left for future research.

7.7 Splitting First-order Rules

In the given form, Theorem 7.44 is mostly relevant to those parts of a dependency pair problem where the higher-order aspect, application of meta-variables, does not play a role. One reason why this might happen is a higher-order system which has a first-order subsystem. Consider for example the following system, which is a slight adaptation of the problem Applicative_AG01_innermost__#4.26 which appears in the termination problem database:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathtt{true} & : & \mathtt{bool} & \mathtt{prev} & : & [\mathtt{nat}] \longrightarrow \mathtt{nat} \\
\mathtt{false} & : & \mathtt{bool} & \mathtt{s} & : & [\mathtt{nat}] \longrightarrow \mathtt{nat} \\
\mathtt{nil} & : & \mathtt{list} & \mathtt{cons} & : & [\mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list} \\
\mathtt{0} & : & \mathtt{nat} & \mathtt{up} & : & [\mathtt{list}] \longrightarrow \mathtt{list} \\
\multicolumn{3}{r}{\mathtt{filter} \;:\; [(\mathtt{nat} \to \mathtt{bool}) \times \mathtt{list}] \longrightarrow \mathtt{list}} & & & \\
\multicolumn{3}{r}{\mathtt{filter2} \;:\; [\mathtt{bool} \times (\mathtt{nat} \to \mathtt{bool}) \times \mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list}} & & & \\
\multicolumn{3}{r}{\mathtt{map} \;:\; [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{list}] \longrightarrow \mathtt{list}} & & & \\
\multicolumn{3}{r}{\mathtt{le} \;:\; [\mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{bool}} & & & \\
\multicolumn{3}{r}{\mathtt{minus} \;:\; [\mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{nat}} & & & \\
\multicolumn{3}{r}{\mathtt{minus2} \;:\; [\mathtt{bool} \times \mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{nat}} & & &
\end{array}
$$

$$\begin{array}{rcl}
\mathtt{prev}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\
\mathtt{prev}(\mathtt{s}(X)) & \Rightarrow & X \\
\mathtt{le}(\mathtt{0}, X) & \Rightarrow & \mathtt{true} \\
\mathtt{le}(\mathtt{s}(X), \mathtt{0}) & \Rightarrow & \mathtt{false} \\
\mathtt{le}(\mathtt{s}(X), \mathtt{s}(Y)) & \Rightarrow & \mathtt{le}(X, Y) \\
\mathtt{minus}(X, Y) & \Rightarrow & \mathtt{minus2}(\mathtt{le}(X, Y), X, Y) \\
\mathtt{minus2}(\mathtt{true}, X, Y) & \Rightarrow & \mathtt{0} \\
\mathtt{minus2}(\mathtt{false}, X, Y) & \Rightarrow & \mathtt{s}(\mathtt{minus}(\mathtt{prev}(X), Y)) \\
\mathtt{map}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{map}(F, \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F \cdot X, \mathtt{map}(F, Y)) \\
\mathtt{filter}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{filter}(F, \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{filter2}(F \cdot X, F, X, Y) \\
\mathtt{filter2}(\mathtt{true}, F, X, Y) & \Rightarrow & \mathtt{cons}(X, \mathtt{filter}(F, Y)) \\
\mathtt{filter2}(\mathtt{false}, F, X, Y) & \Rightarrow & \mathtt{filter}(F, Y) \\
\mathtt{up}(X) & \Rightarrow & \mathtt{map}(\lambda x.\mathtt{s}(x), X)
\end{array}$$

Intuitively, only `map`, `filter`, `filter2` and `up` use the higher-order aspect; the other rules are really first-order. As it happens, this first-order subsystem is quite hard to handle. All techniques we have seen so far fail to prove finiteness of $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$, where $\mathcal{P}$ is the set:

$$\left\{ \begin{array}{rcl}
\mathtt{minus}^\sharp(X, Y) & \Rightarrow\!\!\!\!\Rightarrow & \mathtt{minus2}^\sharp(\mathtt{le}(X, Y), X, Y) \\
\mathtt{minus2}^\sharp(\mathtt{false}, X, Y) & \Rightarrow\!\!\!\!\Rightarrow & \mathtt{minus}^\sharp(\mathtt{prev}(X), Y)
\end{array} \right\}$$

Considering that the usable rules of this set do not include any of the higher-order rules, it would be very nice if we could view this part as a first-order dependency pair problem and use the many existing methods which have not yet been generalised to higher-order rewriting. However, the definition of a higher-order dependency chain allows λ-abstractions and applications in the formation of the terms in the chain, and reductions with a β-rule. It will take some work to see that we can, in fact, ignore the higher-order aspect.

First let us define the intuitive notion of "first-order" rules and dependency pairs. We say a meta-term s is *essentially first-order* if s is closed and has no subterms of functional type. A rule $l \Rightarrow r$ is essentially first-order if both l and r are essentially first-order meta-terms. A dependency pair $l \Rrightarrow p\ (A)$ is essentially first-order if both l and p are essentially first-order meta-terms and A is empty.

Example 7.46. In the example system above, all the `prev`, `le`, `minus` and `minus2` rules are essentially first-order. The elements of $\mathcal{P}$ are essentially first-order dependency pairs.

If both a set of dependency pairs $\mathcal{P}$ and its usable rules are essentially first-order, then any dependency chain over $\mathcal{P}$ can be used to find a first-order dependency chain over $\mathcal{P}$, as we will see below. Write $\mathcal{C}_{\epsilon base}$ for the essentially first-order subset of $\mathcal{C}_\epsilon$: $\{\mathtt{p}_\iota(X, Y) \Rightarrow X, \mathtt{p}_\iota(X, Y) \Rightarrow Y \mid \iota \in \mathcal{B}\}$.

Theorem 7.47. *Let $\mathcal{P}$ be a set of essentially first-order dependency pairs, and $\mathcal{R}$ a finitely branching set of rules. Suppose $A := UR(\mathcal{P}, \mathcal{R})$ consists only of essentially first-order rules. Then $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, f)$ is finite if there is no dependency chain $C = [(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that:*

- *all $\rho_i \in \mathcal{P}$;*
- *for all i: $t_i \Rightarrow^*_{A \cup \mathcal{C}_{\epsilon\, base}} s_{i+1}$ without β-steps, and all terms in this reduction are essentially first-order terms.*

Proof. We define a new function ψ for terminating base-type terms as follows:

- $\psi(x) = x$ if x is a variable;
- $\psi(f(s_1, \ldots, s_n)) = f(\psi(s_1), \ldots, \psi(s_n))$ if f is a usable symbol of $\mathcal{P}$;
- $\psi(f(s_1, \ldots, s_n) : \iota) = \mathtt{p}_\iota(f(\psi(s_1), \ldots, \psi(s_n)), D_\iota(\{t \mid f(\vec{s}) \Rightarrow_{\mathcal{R}} t\}))$ if f is not a usable symbol of $\mathcal{P}$, but all s_i have base types;
- $\psi(s : \iota) = D_\iota(\{t \mid s \Rightarrow_{\mathcal{R}} t\})$ if s has any other form;
- in this definition, $D_\iota(\emptyset) = \bot_\iota$ (a fixed variable), and $D_\iota(X) = \mathtt{p}_\sigma(\psi(t), D_\iota(X \setminus \{t\}))$ if X is non-empty and t lexicographically its smallest element.

Note that $\psi(s)$ is always an essentially first-order term. We also see that any $\Rightarrow_{\mathcal{C}_{\epsilon\, base}}$-reduct of an essentially first-order term is an essentially first-order term.

To start, we can obtain a result very like Lemma 7.40: if s is an essentially first-order meta-term and γ a substitution whose domain contains only meta-variables (and contains all meta-variables in $FMV(s)$), then $\psi(s\gamma) \Rightarrow^*_{\mathcal{C}_{\epsilon\, base}} s\gamma^\psi$; if all elements of $Symb(s)$ are usable symbols of $\mathcal{P}$, then even $\psi(s\gamma) = s\gamma^\psi$. Like Lemma 7.40, the proof is a trivial induction on the form of s.

Using this result, we obtain a counterpart to Lemma 7.41: if $s \Rightarrow_{\mathcal{R}} t$ with s a terminating base-type term, then $\psi(s) \Rightarrow^*_{A \cup \mathcal{C}_{\epsilon\, base}} \psi(t)$; this reduction involves only essentially first-order terms. Like in Lemma 7.41, we prove this by induction on the form of s. The result is obvious if $\psi(s)$ has the form $D_\iota(X)$ or $\mathtt{p}_\iota(\ldots, D_\iota(X))$ with $t \in X$; the only other case is when $s = f(s_1, \ldots, s_n)$ with f a usable symbol. If the reduction happens in one of the s_i then we use the induction hypothesis, otherwise $s = l\gamma$ and $t = r\gamma$ for some substitution γ and usable rule $l \Rightarrow r$. All symbols of r are usable symbols, so $\psi(s) \Rightarrow^*_{\mathcal{C}_{\epsilon\, base}} l\gamma^\psi \Rightarrow_A r\gamma^\psi = \psi(t)$.

Theorem 7.47 follows from this last result exactly as Lemma 7.42 was derived from the preceding lemmas. □

Thus, we can use first-order techniques to prove that a dependency chain over $\mathcal{P}$ does not exist.

However, this situation is not entirely ideal. It is typically harder to prove absence of *arbitrary* dependency chains of a given form than of *minimal* dependency chains. And we cannot easily use a first-order termination tool to determine whether such a chain can exist: most termination tools for first-order

rewriting take as input a term rewriting system, not a dependency pair problem (and moreover, they ignore types).

To make life easier, we might observe that if $\mathcal{P} \subseteq \mathsf{DP}(A)$, then a dependency chain over $\mathcal{P}, A$ can exist only if A is non-terminating. Thus we obtain:

Corollary 7.48 (First-order Rules Processor)**.** *The processor which maps a DP problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, f)$ to $\emptyset$ provided some set A exists such that the following conditions are satisfied, is both sound and complete:*

- *A is a set of essentially first-order rules which contains $UR(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_{\epsilon\, base}$;*
- *$\mathcal{R}$ is a finitely branching set of rules;*
- *$\mathcal{P} \subseteq \mathsf{DP}(A)$;*
- *A is terminating when seen as a first-order (many-sorted) TRS.*

(The *complete* part of this holds because the resulting set does not contain any infinite DP problems.)

Of course, a many-sorted TRS is terminating if the corresponding unsorted TRS is terminating, and there are many highly advanced first-order tools dedicated to finding such termination proofs. Corollary 7.48 allows us to make use of them directly. If $UR(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_{\epsilon\, base}$ is essentially first-order, then it is easy to find a set A containing $UR(\mathcal{P}, \mathcal{R}) \cup \mathcal{C}_{\epsilon\, base}$ and whose dependency pairs include $\mathcal{P}$.

Example 7.49. We attempt to prove finiteness of the dependency pair problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$, where $\mathcal{R}$ consists of the rules at the start of this section, and $\mathcal{P}$ is the set we saw before:

$$\begin{aligned} \mathtt{minus}^\sharp(X, Y) &\Rrightarrow \mathtt{minus2}^\sharp(\mathtt{le}(X, Y), X, Y) \\ \mathtt{minus2}^\sharp(\mathtt{false}, X, Y) &\Rrightarrow \mathtt{minus}^\sharp(\mathtt{prev}(X), Y) \end{aligned}$$

We choose for A the following set:

$$\begin{aligned} \mathtt{prev}(\mathtt{0}) &\Rightarrow \mathtt{0} \\ \mathtt{prev}(\mathtt{s}(X)) &\Rightarrow X \\ \mathtt{le}(\mathtt{0}, X) &\Rightarrow \mathtt{true} \\ \mathtt{le}(\mathtt{s}(X), \mathtt{0}) &\Rightarrow \mathtt{false} \\ \mathtt{le}(\mathtt{s}(X), \mathtt{s}(Y)) &\Rightarrow \mathtt{le}(X, Y) \\ \mathtt{minus}(X, Y) &\Rightarrow \mathtt{minus2}(\mathtt{le}(X, Y), X, Y) \\ \mathtt{minus2}(\mathtt{false}, X, Y) &\Rightarrow \mathtt{minus}(\mathtt{prev}(X), Y) \\ \mathtt{p_{nat}}(X, Y) &\Rightarrow X \\ \mathtt{p_{nat}}(X, Y) &\Rightarrow Y \end{aligned}$$

Then A satisfies the requirements from Corollary 7.48. A is terminating when seen as a first-order, untyped TRS, as is demonstrated for example by AProVE.

A downside of swapping to the first-order setting with Corollary 7.48 or Theorem 7.47 is given by the rules $\mathtt{p}_\iota(X,Y) \Rightarrow X, \mathtt{p}_\iota(X,Y) \Rightarrow Y$. Although these rules seem quite harmless – the $\mathtt{p}_\iota$ symbols do not occur in any other rules, and both StarHorpo and polynomial interpretations can trivially orient them both – they introduce non-determinism. This is inconvenient because it blocks some very nice results, for instance the fact that a locally confluent overlay TRS is terminating if it is innermost terminating [51].

For cases where this in particular may be relevant, let us consider a result which avoids these extra rules: this result will be limited to AFSMs where the rules have *unique normal forms*. Moreover, the dependency pairs under consideration should be *non-overlapping* with the corresponding rules. It is not in general decidable whether an AFSM has unique normal forms, but as we saw in Chapter 3.7.3, we can extend existing results for CRSs to AFSMs. Thus we have for instance that an *orthogonal* AFSM has unique normal forms. Moreover, orthogonality implies non-overlappingness.

Roughly, the idea of the split is as follows: by unicity of normal forms and non-overlappingness, the higher-order subterms of all s_i in a dependency chain with only first-order dependency pairs, can be assumed to be normalised. As topmost first-order steps cannot create higher-order redexes, higher-order subterms anywhere in this chain are normalised, and can be replaced by variables.

To work! We will need to consider a few more rules than $UR(\mathcal{P},\mathcal{R})$. For any set $\mathcal{P}$ of dependency pairs, we define $S_\mathcal{P}$ as the set of all symbols f such that $g \sqsupseteq^*_{us} f$ for any symbol g occurring in the left- or right-hand side of some dependency pair in $\mathcal{P}$. We let $A_\mathcal{P}$ consist of all rules $f(\vec{s}) \cdot \vec{t} \Rightarrow r$ in $\mathcal{R}$ where $f \in S_\mathcal{P}$. We are in particular interested in sets $\mathcal{P}$ of essentially first-order dependency pairs where all elements of $A_\mathcal{P}$ are essentially first-order rules.

We say that a set of essentially first-order dependency pairs $\mathcal{P}$ *overlaps* with $A_\mathcal{P}$ if there exist a pair $l \Rrightarrow p \in \mathcal{P}$ and a rule $u \Rightarrow v \in A_\mathcal{P}$ such that $l'\gamma = u\delta$ for some sub-metaterm l' of l which is not a meta-variable and substitutions γ, δ.

Lemma 7.50. *If for all subterms q of s we have that $q = f(q_1,\ldots,q_n)$ with $f \in A_\mathcal{P}$, or q is in $\mathcal{R}$-normal form, then the same holds for all reducts of s.*

Proof. Suppose s has this property, and that $s \Rightarrow_\mathcal{R} t$; we use induction on the size of s. Since s is not in normal form, $s = f(s_1,\ldots,s_n)$ with $f \in S_\mathcal{P}$. If s reduces topmost to t, therefore, $s = l\gamma$, $t = r\gamma$ with $l \Rightarrow r \in A_\mathcal{P}$. Since r is essentially first-order and contains only symbols in $A_\mathcal{P}$, and the property holds for all subterms of any $\gamma(X)$, the property also holds for $r\gamma$. Otherwise $t = f(s_1,\ldots,s_i',\ldots,s_n)$ with $s_i \Rightarrow_\mathcal{R} s_i'$. By the induction hypothesis the property holds for s_i', and by assumption it holds for the other s_j. □

In Lemma 7.51 we will assume that all terminating terms s have a unique normal form. In this case we can define $\nu(s)$ as $s\downarrow_\mathcal{R}$. If $s = f(s_1,\ldots,s_n)$ with all s_i terminating, then let $\nu'(s) = f(\nu(s_1),\ldots,\nu(s_n))$.

Lemma 7.51 (Normalising Chains)**.** *Let $\mathcal{P}$ be a set of essentially first-order dependency pairs, and suppose $A_\mathcal{P}$ contains only essentially first-order rules. Suppose moreover that all terminating terms s have a unique normal form with respect to $\mathcal{R}$, and that $\mathcal{P}$ does not overlap with $A_\mathcal{P}$. If there exists a minimal dependency chain $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ with all $\rho_i \in \mathcal{P}$, then there also exists a minimal dependency chain $[(\rho_i, \nu'(s_i), q_i) \mid i \in \mathbb{N}]$ such that each $q_i \Rightarrow^*_{A_\mathcal{P},in} \nu'(s_{i+1})$.*

Proof. For given i, let $\rho_i = l^\sharp \Rrightarrow p^\sharp$ and let γ be such that $s_i = l^\sharp\gamma$ and $t_i = p^\sharp\gamma$. Write $\gamma^\downarrow$ for the substitution which maps X to $\gamma(X)\downarrow_\mathcal{R}$ for $X \in dom(\gamma)$. Write $l^\sharp = f^\sharp(l_1, \ldots, l_n)$ and $p^\sharp = g^\sharp(p_1, \ldots, p_m)$.

By the non-overlappingness property, $l'\gamma^\downarrow$ cannot be an instance of the left-hand side of a rule for any strict subterm l' of l which is not a meta-variable, so each $l_j\gamma\downarrow_\mathcal{R}$ is exactly $l_j\gamma^\downarrow$. Thus, $\nu'(s_i) = l^\sharp\gamma^\downarrow$. Let $q_i := p^\sharp\gamma^\downarrow$.

Since $t \Rightarrow^*_{\mathcal{R},in} s_{i+1}$ we can write $s_{i+1} = g^\sharp(u_1, \ldots, u_m)$, where each $p_j\gamma \Rightarrow^*_\mathcal{R} u_j$. By the unique normal forms property, $p_j\gamma^\downarrow\downarrow_\mathcal{R} = p_j\gamma\downarrow_\mathcal{R} = u_j\downarrow_\mathcal{R}$. Noting that all subterms of q_i are either $\mathcal{R}$-normalised or are functional terms with a root symbol in $S_\mathcal{P}$, Lemma 7.50 gives us that $p_j\gamma^\downarrow \downarrow_{A_\mathcal{P}} = p_j\gamma^\downarrow \downarrow_\mathcal{R}$. Thus, $q_i \Rightarrow^*_{A_\mathcal{P}} g^\sharp(\nu(u_1), \ldots, \nu(u_m)) = \nu'(s_{i+1})$ as required.

The resulting chain is minimal because s_i (whose immediate subterms are terminating) reduces with non-topmost-steps to $\nu'(s_i)$, and $p^\sharp\gamma$ reduces with non-topmost-steps to q_i. □

Finally, to get rid of (normalised!) higher-order subterms, we introduce a variable x_ι for all base types ι. For a base-type term $s : \iota$, we define $\xi(s)$ as $f(\psi(s_1), \ldots, \psi(s_n))$ if s has the form $f(s_1, \ldots, s_n)$ with $f \in S_\mathcal{P}$, and $\xi(s) = x_\iota$ otherwise. It follows easily that:

Lemma 7.52 (Replacing higher-order terms)**.** *If all subterms of s are $\mathcal{R}$-normalised or have the form $f(\vec{q})$ with $f \in S_\mathcal{P}$, and if $s \Rightarrow_{A_\mathcal{P}} t$, then $\psi(s) \Rightarrow_{A_\mathcal{P}} \psi(t)$.*

Proof. With induction on q it is evident that, for essentially first-order meta-terms q containing only symbols in $S_\mathcal{P}$, always $\xi(q\gamma) = q\gamma^\xi$. Using induction on the position of the redex in s, this provides the base case ($s \Rightarrow_{A_\mathcal{P}} t$). The induction case, $s = f(s_1, \ldots, s_i, \ldots, s_n) \Rightarrow_{A_\mathcal{P},in} f(s_1, \ldots, s'_i, \ldots, s_n) = t$, holds by the induction hypothesis. □

We now have all the preparations to obtain a counterpart of Theorem 7.47:

Theorem 7.53. *Let $\mathcal{P}$ be a set of essentially first-order dependency pairs, and suppose $\mathcal{R}$ has unique normal forms. Suppose moreover that $A_\mathcal{P}$ consists of essentially first-order rules, and $\mathcal{P}$ does not overlap with $A_\mathcal{P}$. Then there is a minimal dependency chain over $\mathcal{P}$ where always $t_i \Rightarrow^*_{\mathcal{R},in} s_{i+1}$ if and only if there is a minimal dependency chain over $\mathcal{P}$ which involves only essentially first-order terms, and where for all i: $t_i \Rightarrow^*_{A_\mathcal{P}} s_{i+1}$.*

Proof. One side is obvious: if $\mathcal{P}$ is chain-free, then no such sequence exists, since $A_{\mathcal{P}} \subseteq \mathcal{R}$. For the other direction, if there is a minimal dependency chain over $\mathcal{P}, \mathcal{R}$, then by Lemma 7.51 there is a minimal dependency chain over $\mathcal{P}$ which uses only rules in $A_{\mathcal{P}}$. By Lemma 7.52 this chain can be transformed into one which uses only essentially first-order terms. □

The requirements for Theorem 7.53 are essential. Consider for example the following system, where the higher-order part lacks the "unique normal forms" property:

$$\begin{array}{lcllcl} \mathtt{f}(X,\mathtt{b}) & \Rightarrow & \mathtt{g}(X,X) & \mathtt{h}(\lambda x.F(x)) & \Rightarrow & F(\mathtt{a}) \\ \mathtt{g}(X,\mathtt{a}) & \Rightarrow & \mathtt{f}(X,X) & \mathtt{h}(\lambda x.F(x)) & \Rightarrow & F(\mathtt{b}) \end{array}$$

Although $\mathcal{R}_{\mathtt{TFO}}$ (which consists of the two rules on the left) is terminating and orthogonal, there is an infinite dependency chain using only truly first-order dependency pairs:

$$\begin{array}{rcl} \mathtt{f}^\sharp(\mathtt{h}(\lambda x.x),\mathtt{b}) & \Rightarrow & \mathtt{g}^\sharp(\mathtt{h}(\lambda x.x),\mathtt{h}(\lambda x.x)) \Rightarrow \mathtt{g}^\sharp(\mathtt{h}(\lambda x.x),\mathtt{a}) \\ & \Rightarrow & \mathtt{f}^\sharp(\mathtt{h}(\lambda x.x),\mathtt{h}(\lambda x.x)) \Rightarrow \mathtt{f}^\sharp(\mathtt{h}(\lambda x.x),\mathtt{b}) \end{array}$$

This happens because the first-order part is duplicating, and $\mathtt{h}(\lambda x.x,\mathtt{a})$ reduces both to $\mathtt{a}$ and to $\mathtt{b}$. Note that the role of the higher-order part could be taken over by the $\mathtt{p}_\iota$-rules: $\mathcal{R}_{\mathtt{TFO}} \cup \mathcal{C}_{\epsilon\, base}$ is not terminating.

As before, we also obtain:

Corollary 7.54 (Non-overlapping First-order Rules Processor)**.** *The processor which maps a DP problem $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, f)$ to $\emptyset$ provided some set A exists such that the following conditions are satisfied, is both sound and complete:*

- *A is a set of essentially first-order rules which contains $A_{\mathcal{P}}$;*
- *$\mathcal{P}$ is a set of essentially first-order dependency pairs, included in $\mathsf{DP}(A)$;*
- *$\mathcal{R}$ has unique normal forms;*
- *$\mathcal{P}$ and $A_{\mathcal{P}}$ are non-overlapping;*
- *A is terminating when seen as a first-order (many-sorted) TRS.*

As a final alternative to Corollaries 7.48 and 7.54, we might be able to handle *all* first-order dependency pairs of a termination problem at once.

To this end, let B be a set consisting of all function symbols $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau$ such that some σ_i or τ is not a base type, or a rule $f(l_1, \ldots, l_n) \Rightarrow r$ exists which is not essentially first-order. We define $\mathtt{PHO}$, the set of *potentially higher-order symbols*, recursively: $\mathtt{PHO}$ contains all symbols in B and, if there is a rule $f(l_1, \ldots, l_n) \Rightarrow r$ where either r or one of the l_i contains a symbol in $\mathtt{PHO}$, then also $f \in \mathtt{PHO}$. Let $\mathtt{TFO}$, the set of *truly first-order symbols*, be defined as $\mathcal{F} \setminus \mathtt{PHO}$. A (meta-)term is called truly first-order if it is built only from symbols

in TFO and base-type (meta-)variables. A rule is truly first-order if it has the form $f(l_1, \ldots, l_n) \Rightarrow r$ with $f \in \mathtt{TFO}$, and potentially higher-order otherwise; let $\mathcal{R}_{\mathtt{TFO}}$ denote the set of truly first-order rules. We say a set of truly first-order rules $\mathcal{R}$ is *overlay* if for all $l \Rightarrow r,\ u \Rightarrow v \in \mathcal{R}$, substitutions γ, δ and non-empty contexts C: if $l = C[l']$ with $l'\gamma = u\delta$, then l' is a meta-variable.

Theorem 7.55 (First-Order Splitting)**.** *Suppose that one of the following holds:*

- *$\mathcal{R}$ is finitely branching and $\mathcal{R}_{\mathtt{TFO}} \cup \mathcal{C}_{\epsilon\, base}$ is terminating when seen as a many-sorted first-order TRS,* or
- *$\mathcal{R}$ has unique normal forms, $\mathcal{R}_{\mathtt{TFO}}$ is overlay, and $\mathcal{R}_{\mathtt{TFO}}$ is terminating when seen as a many-sorted first-order TRS.*

Then $\mathcal{R}$ is terminating if and only if the dependency pair problem $(\mathsf{DP}(\mathcal{R}_{\mathtt{PHO}}), \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ is finite.

Proof. We first observe that all truly first-order rules are also essentially first-order rules, and if $l \Rightarrow r$ is a truly first-order rule, then any symbol f occurring in l or r is in TFO; moreover, if $f \sqsupseteq_{us} g$ then also $g \in \mathtt{TFO}$. Thus, all rules of the form $f(\vec{s}) \Rightarrow t$ are also truly first-order rules, and $UR(r, \mathcal{R}) \subseteq \mathcal{R}_{\mathtt{TFO}}$. We then note that for $\mathcal{P} := \mathsf{DP}(\mathcal{R}_{\mathtt{TFO}})$, both $UR(\rho, \mathcal{R}) \subseteq \mathcal{R}_{\mathtt{TFO}}$ and $A_{\mathcal{P}} \subseteq \mathcal{R}_{\mathtt{TFO}}$, as all symbols in left- and right-hand sides of $\mathcal{P}$ are in TFO and TFO is closed under $\sqsupseteq_{us}$.

In any dependency chain either all dependency pairs are in $\mathsf{DP}(\mathcal{R}_{\mathtt{PHO}})$, or a tail of the chain uses only dependency pairs in $\mathsf{DP}(\mathcal{R}_{\mathtt{TFO}})$. This is because a truly first-order dependency pair $f^\sharp(\vec{l}) \Rrightarrow g^\sharp(\vec{p})$ can only be followed by another truly first-order dependency pair: g occurs in the right-hand side of some truly first-order rule $l \Rightarrow r$, so all pairs of the form $g^\sharp(\vec{s}) \Rightarrow r'$ are also in $\mathsf{DP}(\mathcal{R}_{\mathtt{TFO}})$.

We know that $\mathcal{R}$ is terminating if and only if there is no minimal formative dependency chain, so both the DP problems $(\mathsf{DP}(\mathcal{R}_{\mathtt{TFO}}), \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ and $(\mathsf{DP}(\mathcal{R}_{\mathtt{PHO}}), \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ are finite. Theorems 7.48 and 7.54 guarantee finiteness of the former. Thus, it suffices to prove finiteness of the latter. □

Theorem 7.55 does not cover all the cases where Theorems 7.48 or 7.54 may be helpful, but nevertheless provides a convenient quick check: if $\mathcal{R}_{\mathtt{TFO}}$ (possibly with the extra rules of $\mathcal{C}_{\epsilon\, base}$ is terminating as a many-sorted TRS, then we can immediately throw away all truly first-order dependency pairs.

In a system with unique normal forms, where $\mathcal{R}_{\mathtt{TFO}}$ is overlay, we can truly split the termination proof into two parts: first, prove termination of $\mathcal{R}_{\mathtt{TFO}}$ as a many-sorted TRS; second, use the dependency pair framework, but omit the dependency pairs for the first-order rules. Doing so does not lose generality, for if $\mathcal{R}_{\mathtt{TFO}}$ is not terminating as a many-sorted first-order TRS, then $\mathcal{R}$ itself cannot be terminating!

And that is not all. Since $\mathcal{R}_{\mathtt{TFO}}$ is an overlay TRS with unique normal forms, it is terminating if it is innermost terminating: by [51] this holds for a locally confluent overlay TRS, and by e.g. [118] an innermost terminating (so weakly

normalising) TRS with unique normal forms is confluent. Since [39] shows that innermost termination is persistent (a many-sorted TRS is innermost terminating if and only if it is innermost terminating without regarding types), an automatic approach can send the resulting TRS to any first-order termination prover without losing generality, whether or not this prover is type-conscious.

7.8 Static Dependency Pairs

To close off this chapter, let us look at an alternative approach to higher-order dependency pairs. The dynamic style presented in Chapter 6 and improved in this chapter is not the only approach to higher-order dependency pairs: as discussed in Section 6.1, there are strong results in *static* dependency pairs [87, 114]. Most importantly, in a static dependency pair approach we do not have to consider collapsing dependency pairs. As we have seen, having only non-collapsing dependency pairs makes life a lot easier for the termination prover: the dependency graph is more sparse, we can use the subterm criterion, usable rules, and there is a fairly good chance that other techniques from the first-order world, too, extend more easily than in the presence of collapsing dependency pairs.

Since the dynamic dependency pair approach is applicable to a larger class of systems, but the static dependency pair approach gives easier constraints, it seems sensible for a termination prover to use *both* techniques. Ideally, the techniques should be merged in the same framework.

Static dependency pairs having been defined for HRSs, we could use Transformation 3.6 to transpose the technique to AFSMs. But this would not give us the ability to use formative rules with static dependency pairs, nor would we have a parallel for the meta-variable conditions used in dynamic setting. In order to use both formalisms in the same framework, and to give the static approach the additional power of formative rules and meta-variable conditions, we will instead natively derive the static dependency pair result for AFSMs here.

To start, consider the restriction we shall use:

Definition 7.56. An AFSM $(\mathcal{F}, \mathcal{R})$ is *plain function passing* if:

- all symbols in $\mathcal{F}$ have a type declaration $[\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \iota$, with ι a base type;
- for all rules $f(l_1, \ldots, l_n) \Rightarrow r$ in $\mathcal{R}$ and all functional meta-variables F which occur in r, some $l_i = \lambda x_1 \ldots x_m.F(x_{i_1}, \ldots, x_{i_k})$.

Here, a "functional meta-variable" is a meta-variable which either has a functional output type, or which takes arguments.

The first of these requirements can be satisfied by η-expanding and uncurrying the system, as was done in Chapter 2.3 (although we may lose termination in the process). The second, which essentially expresses that functional meta-variables

should occur as direct arguments of the left-hand side of a rule (with an exception if they are not actually used in a relevant way) is a more fundamental property.

Consider for instance the AFSM `eval` introduced in Example 6.14. This AFSM does have the first property, but in the rule

$$\mathtt{eval}(\mathtt{fun}(\lambda x.F(x), X, Y), Z) \Rightarrow F(\mathtt{dom}(X, Y, Z))$$

the $\lambda x.F(x)$ does not occur as a direct argument of the left-hand side, and Z occurs in the right-hand side, so this system is not plain function passing. The same holds for the similar system $\mathtt{eval}'$, which has a rule

$$\mathtt{eval}(\mathtt{fun}(F, X, Y), Z) \Rightarrow F \cdot \mathtt{dom}(X, Y, Z)$$

The system `twice` satisfies the second requirement, but not the first (`twice` has a functional type as output type). However, with the theory of Chapter 2.3 this can be amended. The symbol `twice` is assigned a type denotation $\mathtt{twice} : [(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{nat}] \longrightarrow \mathtt{nat}$, and the rewrite rules become:

$$\begin{array}{rcl} \mathtt{I}(\mathtt{0}) & \Rightarrow & \mathtt{0} \\ \mathtt{I}(\mathtt{s}(X)) & \Rightarrow & \mathtt{s}(\mathtt{twice}(\lambda x.\mathtt{I}(x), X)) \\ \mathtt{twice}(F, X) & \Rightarrow & (\lambda y.F \cdot (F \cdot y)) \cdot X \end{array}$$

If the resulting system is terminating, then so is the original.

Definition 7.57 (Static Candidate Terms)**.** Recall the definition of β*-reduced sub-meta-terms* of a meta-term from Definition 6.18. The *static candidate terms* of a closed meta-term s are those β-reduced sub-meta-terms which have the form $f(s_1, \ldots, s_n)\ (A)$ for some $f \in \mathcal{D}$.

As before, we consider candidate terms equal if they are equal modulo renaming of variables, so a meta-term only has finitely many different candidate terms.

By this definition, only functional meta-terms are static candidate terms (and they have base type by the restriction on $\mathcal{F}$!). However, the price for this is paid in the definition of a dependency pair:

Definition 7.58 (Static Dependency Pairs)**.** The *static dependency pairs* of a rule $l \Rightarrow r$ are all pairs $l^\sharp \Rrightarrow p'^\sharp$ where $p \in Cand(r)$, p' is obtained from p by replacing all free variables by fresh meta-variables, and p' is not a strict subterm of l.

The set of static dependency pairs of a set of rules $\mathcal{R}$, notation $\mathsf{DP}^{static}(\mathcal{R})$, consists of the static dependency pairs of all rules in $\mathcal{R}$.

Unlike the dynamic definition, free variables in the right-hand sides are replaced by meta-variables, which may cause trouble later. Note, however, that static dependency pairs are still dependency pairs as defined in Definition 6.20. In fact, this is why Definition 6.20 does not require that $FMV(p) \subseteq FMV(l)$ holds in a dependency pair $l \Rrightarrow p\ (A)$.

Example 7.59. The static dependency pairs of the altered `twice` system are:

$$\begin{array}{lcl} \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{twice}^\sharp(\lambda x.\mathtt{I}(x), X) \\ \mathtt{I}^\sharp(\mathtt{s}(X)) & \Rrightarrow & \mathtt{I}^\sharp(Y) \end{array}$$

Note that both sides of a static dependency pair have base type, by the restriction on the type declaration of the f.

A static dependency chain, now, is defined exactly like our original definition (Definition 6.23). However, since there are no collapsing dependency pairs, we do not have to consider steps with $\rho_i = \mathtt{beta}$, nor subterms steps, so can omit constraints 3 and 2d, and since static dependency pairs do not contain free variables we can omit constraint 2e. What remains is a definition very close to the first-order case, but which does have the meta-variable conditions:

Definition 7.60. A *static dependency chain* is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i:

1. $\rho_i = l_i \Rrightarrow p_i\ (A_i) \in \mathsf{DP}^{static}$;
2. there exists a substitution γ such that $s_i = l_i\gamma$ and $t_i = p_i\gamma$, and γ respects A_i (**);
3. $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ (***);

A static dependency chain is *minimal* if the strict subterms of all t_j are terminating in $\Rightarrow_{\mathcal{R}}$, and *formative* if always $t_i \Rightarrow^*_{\mathcal{R},in} s_{i+1}$ by a formative ρ_{i+1}-reduction.

(**) As before, the phrase "γ respects A" means that for all $F : j \in A$ the substitute $\gamma(F) = \lambda x_1 \ldots x_n.q$ has the property that $x_j \in FV(q)$.

(***) Since all s_i, t_i have the form $f^\sharp(\vec{q})$, they cannot be reduced at the top. Thus, $t_i \Rightarrow^*_{\mathcal{R}} s_{i+1}$ corresponds exactly with $t_i \Rightarrow^*_{\mathcal{R},in} s_{i+1}$.

An example of a minimal, formative static dependency chain is the chain where $\rho_i = \mathtt{I}^\sharp(\mathtt{s}(X)) \Rrightarrow \mathtt{I}^\sharp(Y)$, $s_i = \mathtt{I}^\sharp(\mathtt{s}(\mathtt{0}))$ and $t_i = \mathtt{I}^\sharp(\mathtt{s}(\mathtt{0}))$ for all i; this uses the substitution $[X := \mathtt{0}, Y := \mathtt{s}(\mathtt{0})]$. Since we have seen that `twice` is, in fact, terminating, this demonstrates that using static dependency pairs is not complete. However, it is sound, as demonstrated by the following result:

Theorem 7.61. *If a plain function passing AFSM $(\mathcal{F}, \mathcal{R})$ is non-terminating, it admits a minimal, formative, static dependency chain.*

Proof. Recall the notion of *computability* with respect to $\Rightarrow_{\mathcal{R}}$ from Definition 5.14:

- a base-type term is computable if it is terminating under $\Rightarrow_{\mathcal{R}}$;
- a term $s : \sigma \to \tau$ is computable if for all computable terms $t : \sigma$ the term $s \cdot t$ is computable

It is easy to see that a term $s : \sigma_1 \to \ldots \to \sigma_n \to \iota$ with ι a base type is computable

if and only if $s \cdot t_1 \cdots t_n$ is terminating for all computable $t_1 : \sigma_1, \ldots, t_n : \sigma_n$, that the reduct of a computable term is computable and, as in Lemma 5.15, that all variables are computable and that all computable terms are terminating.

Recall, too, the notion of β-reduced sub-meta-terms. To make the induction hypothesis easier, it will be practical to use an extension of this notion; $\psi_A(s) =$

$$\{s\ (A)\} \cup \begin{cases} \psi_A(t) & \text{if} \quad s = \lambda x.t \\ \psi_A(s_1) \cup \ldots \cup \psi_A(s_n) & \text{if} \quad s = f(s_1, \ldots, s_n) \\ \psi_A(s_1) \cup \ldots \cup \psi_A(s_n) & \text{if} \quad s = x \cdot s_1 \cdots s_n \\ \psi_{A_1}(s_1) \cup \ldots \cup \psi_{A_n}(s_n) & \text{if} \quad s = Z(s_1, \ldots, s_m) \cdot s_{m+1} \cdots s_n \\ & \quad \text{where } A_i = A \cup \{Z : i\} \\ \psi_A(t[x := q] \cdot \vec{u}) \cup \psi_A(q) & \text{if} \quad s = (\lambda x.t) \cdot q \cdot \vec{u} \end{cases}$$

A pair $t\ (A)$ in this set is *minimal* with respect to some property P if $s\ (A)$ has property P, but all other meta-terms in $\psi_A(s)$ do not.

We assume a plain function passing, non-terminating AFSM $(\mathcal{F}, \mathcal{R})$, and construct a minimal, formative, static dependency chain. Since a non-terminating term is not computable, we are always able to find a non-computable term t. Choose an element $s\ (A)$ of $\psi_\emptyset(t)$ that is minimal with respect to the property "$\lambda x_1 \ldots x_n.s$ is not computable", where $FV(s) \setminus FV(t) = \{x_1, \ldots, x_n\}$. Since $t\ (\emptyset)$ itself has this property, such an element can always be found.

We should find out what form s might have. To this end, consider the claim:

(**) $\lambda x_1 \ldots x_n.s$ is computable if and only if $s[x_1 := q_1, \ldots, x_n := q_n]$ is computable for computable terms $q_1, \ldots, q_n$.

One direction is obvious: if $\lambda x_1 \ldots x_n.s$ is computable, then so is $(\lambda x_1 \ldots x_n.s) \cdot \vec{q}$, which β-reduces to $s[x_1 := q_1, \ldots, x_n := q_n]$, so the latter is computable too. For the other direction, the statement holds because $\lambda \vec{x}.s$ is computable if and only if for all computable $q_1, \ldots, q_n, \ldots, q_k$ the term $(\lambda \vec{x}.s) \cdot \vec{q}$ is terminating. If $s = s[\vec{x} := \vec{x}]$ is computable, then both s and all q_i are terminating, so an infinite reduction starting in $(\lambda \vec{x}.s) \cdot \vec{q}$ must eventually take some headmost β-steps. As in the proof of for instance Theorem 6.24, we can safely assume these β-steps are done immediately, so $s[x_1 := t_1, \ldots, x_n := t_n] \cdot t_{n+1} \cdots t_k$ is non-terminating. But this contradicts computability of $s[x_1 := t_1, \ldots, x_n := t_n]$!

Now, what form could s have?

- Suppose $s = \lambda y.q$. If $\lambda x_1 \ldots x_n.s$ is not computable, then neither is $\lambda x_1 \ldots x_n y.q$, as this is the same term. Thus, s is not minimal.

  We conclude: s cannot be an abstraction.

- Suppose $s = x \cdot s_1 \cdots s_n$ with x a variable. By (**), $s\gamma$ is non-computable for some computable substitution γ, but all $s_i\gamma$ are computable. However, whether $x \in dom(\gamma)$ or not, $x\gamma$ is computable, so $s\gamma$ is an application of computable terms, and must be computable itself!

  We conclude: s could not be a variable, or an application headed by a variable.

- Suppose $s = f(s_1, \ldots, s_n)$ with f a constructor symbol. Using (**), $s\gamma$ is non-computable for some substitution γ, but all $s_i\gamma$ are computable, and therefore terminating. Since topmost reductions are impossible for a term whose root symbol is a constructor symbol, there is no infinite reduction starting in $s\gamma$. As s has base type, this means $s\gamma$ is computable, contradiction.

  We conclude: s cannot be a functional term with a constructor symbol as root symbol.

- Suppose $s = (\lambda x.q) \cdot u \cdot \vec{v}$. Using (**), $s\gamma$ is non-computable for some substitution γ, but both $u\gamma$ and $q[x := u] \cdot \vec{v}\gamma$ are computable. By non-computability of $s\gamma$, there are computable terms $w_1, \ldots, w_n$ such that $s\gamma \cdot \vec{w}$ is not terminating. Since each of $u\gamma, q\gamma$, all $v_i\gamma$ and all w_i are terminating (by assumption for u and the w_i, and for the others termination is implied by termination of $q[x := u] \cdot \vec{v}\gamma$), a headmost β-step must eventually be taken. As seen before, we can safely assume it happens immediately, so $(q[x := u] \cdot \vec{v})\gamma \cdot \vec{w}$ is non-terminating, which implies non-computability of $(q[x := u] \cdot \vec{v})\gamma$, contradiction.

  We conclude: s could not be an application headed by an abstraction.

Since functional terms have a base type as output type, s cannot be an application at all, nor a variable or abstraction. The only remaining form is $f(s_1, \ldots, s_n)$ with $f \in \mathcal{D}$. We see that $s\gamma$ is *minimal non-terminating*, because all $s_i\gamma$ are computable, so terminating, and $s\gamma$ itself has base type. Let $q_{-1} := s\gamma$.

For all $i \in \mathbb{N}$, suppose we have a functional, base-type term q_{i-1} which is minimal non-terminating, and all whose immediate subterms are even computable. Let $t_{i-1} = q_{i-1}^\sharp$. Consider an infinite reduction starting in q_{i-1}.

Eventually, a topmost step must be taken. We have: $q_{i-1} \Rightarrow^*_{\mathcal{R},in} l\gamma \Rightarrow_{\mathcal{R}} r\gamma \Rightarrow_{\mathcal{R}} \ldots$, with $r\gamma$ still being non-terminating. By Lemma 6.41 we can assume that the reduction $q_{i-1} \Rightarrow^*_{\mathcal{R},in} l\gamma$ is a formative l-reduction.

Let $l = f(l_1, \ldots, l_n)$. Since the direct subterms of q_{i-1} are computable, and the reducts of a computable term are also computable, we know that all $l_i\gamma$ are computable. Let $p\ (A)$ be a minimal pair in $\psi_\emptyset(r)$ with the property that (1) γ respects A, and (2) $\lambda x_1 \ldots x_n.(p\gamma)$ is not computable, if $FV(p) = \{x_1, \ldots, x_n\}$ (since $r\gamma$ itself is not computable, and γ respects $\emptyset$, such p exists).

Let $p' := p[x_1 := Z_1, \ldots, x_n := Z_n]$ for fresh meta-variables $Z_1, \ldots, Z_n$. Suppose we can see that $p\ (A)$ is a candidate term of r. Then p' is a functional term. Moreover, since $\lambda\vec{x}.p\gamma$ is not computable we can find an extension δ of γ such that all $\delta(Z_i)$ are computable, $p'\delta$ is not terminating, yet $p''\delta$ is computable for all strict subterms p'' of p', by minimality of p'. Certainly p'' is not a strict subterm of l, as the direct subterms of $l\gamma = l\delta$ are computable, so all strict subterms are terminating.

Thus, we can choose $\rho_i := l^\sharp \Rrightarrow p'^\sharp\ (A),\ s_i := l^\sharp\gamma,\ q_i := p\delta$ and $t_i := p'^\sharp\delta$, and with these values the inductive reasoning continues.

It remains to be seen that if p (A) is minimal in $\psi_\emptyset(r)$ with the property that γ respects A and $\lambda x_1 \dots x_n.(p\gamma)$ is not computable, then p is a candidate term of r. That is, we must see that p has the form $f(p_1, \dots, p_n)$ with $f \in \mathcal{D}$. This is enough, because all elements of $\psi_\emptyset(r)$ which have such a form are also β-reduced sub-meta-terms of r.

Consider what forms p could have. It cannot be an abstraction $\lambda x.p'$, as p' would also do the job (contradicting minimality of p), and for any other form we use (**): we must see that $p\delta$ is computable when $\delta = \gamma \cup [x_1 := q_1, \dots, x_n := q_n]$ for computable terms $q_1, \dots, q_n$, and may assume that $p''\delta$ is computable for the immediate sub-meta-terms p'' of p, provided this does not clash with meta-variable conditions.

- For the case where p is a variable, application headed by a variable, application headed by an abstraction, or a functional term with root symbol in $\mathcal{C}$, see the reasoning used above. We obtain a contradiction in the same way here.
- If $p = Z$, a base-type meta-variable, then $Z\delta = \gamma(Z)$, a strict subterm of $l\gamma$, which is terminating and therefore computable.
- If $p = Z(p_1, \dots, p_m) \cdot p_{m+1} \cdots p_k$ for some functional meta-variable Z, then, because $\mathcal{R}$ is plain function passing, one of the l_i has the form $\lambda y_1 \dots y_m.$ $Z(y_1, \dots, y_m)$. Thus, $\gamma(Z) = \lambda y_1 \dots y_m.q$, is computable.

  We may assume that all $p_i\delta$ with $i > m$ are computable. For smaller i, we have that $p_i\delta$ is computable if γ respects $A \cup \{Z : i\}$, so if y_i occurs in q. Let $u_i := p_i\delta$ if either $i > m$ or γ respects $A \cup \{Z : i\}$, and u_i is a fresh variable otherwise. Then all u_i are computable, and therefore $\gamma(Z) \cdot u_1 \cdots u_k$ is, too. This term β-reduces to $q[y_1 := u_1, \dots, y_m := u_m] \cdot u_{m+1} \cdots u_k$, which is therefore computable. Since this term equals $q[y_1 := p_1\delta, \dots, y_m := p_m\gamma] \cdot (p_{m+1}\delta) \cdots (p_k\delta) = p\delta$ (this is the case because y_i does not occur in q when $u_i \neq p_i\delta$), we have the required result.

The only remaining form for p is a functional term whose root symbol is defined. The conclusion is that p must, indeed, be a candidate term of r. □

Thus we see: if a system is plain function passing, then it is terminating if it does not admit a minimal, formative dependency chain on the static dependency pairs. The beauty of this conclusion is the following: every static dependency chain is a dynamic dependency chain – it just uses different dependency pairs. Since neither reduction pairs, nor any of the results of this chapter use the fact that in dynamic dependency pairs all meta-variables in the right-hand side also appear in the left,[4] all results immediately apply. We could use the same dependency

[4] An exception is the proof of Theorem 7.24, where we use that if the right-hand side of a dependency pair has the form $F(\vec{s})$, then F occurs in the left-hand side of the dependency pair. This is still the case with static dependency pairs, as the fresh meta-variables are not the right-hand side of a dependency pair.

pair module with static or dynamic dependency pairs. When using an automatic tool, we could simply start with dynamic dependency pairs, call the dependency pair algorithm, and if that fails, yet the system is plain function passing (or can be made plain function passing by η-expanding it), calculate the static dependency pairs and call the dependency pair framework with that instead.

But in some cases, we can do better! The examples `twice` and `eval` used in this chapter are particularly suited for the dynamic dependency pair approach, but let us now consider the example from Section 7.7. This system has the following dynamic dependency pairs:

$$\begin{aligned}
\mathtt{le}^\sharp(\mathtt{s}(X),\mathtt{s}(Y)) &\Rrightarrow \mathtt{le}^\sharp(X,Y)\\
\mathtt{minus}^\sharp(X,Y) &\Rrightarrow \mathtt{minus2}^\sharp(\mathtt{le}(X,Y),X,Y)\\
\mathtt{minus}^\sharp(X,Y) &\Rrightarrow \mathtt{le}^\sharp(X,Y)\\
\mathtt{minus2}^\sharp(\mathtt{false},X,Y) &\Rrightarrow \mathtt{minus}^\sharp(\mathtt{prev}(X),Y)\\
\mathtt{minus2}^\sharp(\mathtt{false},X,Y) &\Rrightarrow \mathtt{prev}^\sharp(X)\\
\mathtt{map}^\sharp(F,\mathtt{cons}(X,Y)) &\Rrightarrow F\cdot X\\
\mathtt{map}^\sharp(F,\mathtt{cons}(X,Y)) &\Rrightarrow \mathtt{map}(F,Y)\\
\mathtt{filter}^\sharp(F,\mathtt{cons}(X,Y)) &\Rrightarrow F\cdot X\\
\mathtt{filter}^\sharp(F,\mathtt{cons}(X,Y)) &\Rrightarrow \mathtt{filter2}^\sharp(F\cdot X,F,X,Y)\\
\mathtt{filter2}^\sharp(\mathtt{true},F,X,Y) &\Rrightarrow \mathtt{filter}^\sharp(F,Y)\\
\mathtt{filter2}^\sharp(\mathtt{false},F,X,Y) &\Rrightarrow \mathtt{filter}^\sharp(F,Y)\\
\mathtt{up}^\sharp(X) &\Rrightarrow \mathtt{map}^\sharp(\lambda x.\mathtt{s}(x),X)
\end{aligned}$$

And its static dependency pairs are:

$$\begin{aligned}
\mathtt{le}^\sharp(\mathtt{s}(X),\mathtt{s}(Y)) &\Rrightarrow \mathtt{le}^\sharp(X,Y)\\
\mathtt{minus}^\sharp(X,Y) &\Rrightarrow \mathtt{minus2}^\sharp(\mathtt{le}(X,Y),X,Y)\\
\mathtt{minus}^\sharp(X,Y) &\Rrightarrow \mathtt{le}^\sharp(X,Y)\\
\mathtt{minus2}^\sharp(\mathtt{false},X,Y) &\Rrightarrow \mathtt{minus}^\sharp(\mathtt{prev}(X),Y)\\
\mathtt{minus2}^\sharp(\mathtt{false},X,Y) &\Rrightarrow \mathtt{prev}^\sharp(X)\\
\mathtt{map}^\sharp(F,\mathtt{cons}(X,Y)) &\Rrightarrow \mathtt{map}(F,Y)\\
\mathtt{filter}^\sharp(F,\mathtt{cons}(X,Y)) &\Rrightarrow \mathtt{filter2}^\sharp(F\cdot X,F,X,Y)\\
\mathtt{filter2}^\sharp(\mathtt{true},F,X,Y) &\Rrightarrow \mathtt{filter}^\sharp(F,Y)\\
\mathtt{filter2}^\sharp(\mathtt{false},F,X,Y) &\Rrightarrow \mathtt{filter}^\sharp(F,Y)\\
\mathtt{up}^\sharp(X) &\Rrightarrow \mathtt{map}^\sharp(\lambda x.\mathtt{s}(x),X)
\end{aligned}$$

That is, the static dependency pairs are exactly the dynamic dependency pairs, except for the collapsing ones. Thus, if there is a (minimal formative) dependency chain on the static dependency pairs, then there is also one on the dynamic ones. In this case we might as well use only the static approach immediately.

Definition 7.62. An AFSM $(\mathcal{F},\mathcal{R})$ is *strongly plain function passing* (SPFP) if it is plain function passing, and the right-hand sides of rules in $\mathcal{R}$ do not have subterms of the form $\lambda x.C[f(s_1,\ldots,s_n)]$ where f is a defined symbol and $x \in FV(f(\vec{s}))$.

Theorem 7.63. *A SPFP AFSM admits a (minimal, formative) static dependency chain if and only if it is non-terminating.*

Proof. One direction is Theorem 7.61; for the other, note that any static dependency pair in an SPFP AFSM is also a dynamic dependency pair. Thus, any static dependency chain in an SPFP AFSM *is* a dynamic dependency chain, so its existence implies non-termination of the AFSM by Theorem 6.24. □

Apart from strengthening the dynamic approach (by allowing us to remove collapsing dependency pairs in an AFSM if it is strong plain function passing), this theorem shows that the static approach is complete for the class of SPFP systems.

In summary, we can combine the static and dynamic approaches as follows:

```
if R is strong plain function passing:
   return DPframework(STATIC)
if DPframework(DYNAMIC) = TERMINATING:
   return TERMINATING
η-expand R if necessary
if R is plain function passing:
   return DPframework(STATIC)
```

A valid question would be: can we do more? Rather than using the two approaches in sequence, or perhaps in parallel, could we *combine* them, and for instance use static dependency pairs as a processor in the dynamic framework, where the collapsing dependency pairs are dropped under certain circumstances? Theorem 7.63 seems to come close, but can only be used when starting the dependency pair framework, not as a processor.

In general, this seems like a difficult task. Consider for instance the non-terminating AFSM with $\mathcal{F} = \{\mathsf{0} : \mathtt{nat},\ \mathsf{f} : [\mathtt{nat}] \longrightarrow \mathtt{nat},\ \mathsf{g} : [\mathtt{nat} \to \mathtt{nat}] \longrightarrow \mathtt{nat}\}$ and the following rules:

$$\begin{array}{rcl} \mathsf{f}(\mathsf{0}) & \Rightarrow & \mathsf{g}(\lambda x.\mathsf{f}(x)) \\ \mathsf{g}(\lambda x.F(x)) & \Rightarrow & F(\mathsf{0}) \end{array}$$

The dynamic dependency pairs of this system are:

$$\begin{array}{rcl} \mathsf{f}^\sharp(\mathsf{0}) & \Rrightarrow & \mathsf{g}^\sharp(\lambda x.\mathsf{f}(x)) \\ \mathsf{f}^\sharp(\mathsf{0}) & \Rrightarrow & \mathsf{f}^\sharp(x) \\ \mathsf{g}^\sharp(\lambda x.F(x)) & \Rrightarrow & F(\mathsf{0}) \end{array}$$

The second of these dependency pairs has no outgoing edges in the dependency graph (recall that variables may only be instantiated with other variables). Thus, in the dynamic dependency pair framework, this pair would be immediately eliminated. In the resulting dependency pair problem, we should certainly not eliminate the collapsing dependency pair, for it is essential to get a dependency chain!

This example shows that we have to watch out: the naive way of using static dependency pairs as a processor will not be valid. So far, no better method of combining the approaches has been proposed.

7.9 Overview

In this chapter we have seen a definition of the dependency pair framework for higher-order rewriting. In addition, we have seen a variety of processors to transform dependency pair problems. Some of these are extensions of existing first-order techniques: the dependency graph, subterm criterion and usable rules. Others, such as the transformations of collapsing dependency pairs, are specific to the higher-order case. A new feature, too, is the *formative rules* processor.

Moreover, we have seen that the *static* style of dependency pairs can be used with the same framework, and derived that formative rules and meta-variable conditions, two features thus far limited to the dynamic style, can also be used in this setting. As a consequence, we have obtained a completeness result for the static approach, in the class of *strong plain function passing* systems.

The dependency pair framework for higher-order rewriting has a long distance to go yet. For a start, there are many first-order processors which we might consider for extension. To name some examples (all of which appear in [119]): a reduction pair with usable rules with respect to an argument filtering, switching to innermost termination,[5] or narrowing. In the higher-order setting, too, we might use the dependency pair framework for innermost termination and non-termination as well as full termination.

For a different direction of research, the `formative` flag deserves some study. This flag is not present in the first-order case (in fact, the technique was first defined by Femke van Raamsdonk and me, for the higher-order setting), so little parallel exists in the first-order world to draw from. We might consider dedicated techniques, like formative rules in combination with an argument filtering. Or we could investigate whether the method can be strengthened, and used as a processor, rather than a flag. Or we may avoid including some of the collapsing rules. There are many open questions!

[5] In the higher-order setting, full termination is not equivalent to innermost termination even in orthogonal systems, as is demonstrated by the orthogonal system $\mathtt{f}(\lambda x.F(x)) \Rightarrow F(\mathtt{a})$, $\mathtt{g}(X) \Rightarrow X$, $\mathtt{h}(\mathtt{a}) \Rightarrow \mathtt{f}(\lambda x.\mathtt{g}(\mathtt{h}(x)))$. However, we may in some cases be able to do a switch to innermost *weak* termination, where the innermost strategy does not affect terms beneath an abstraction.

Wanda 8

Or, How can we make a computer do the work?

The techniques discussed in this paper can all be used for pen-and-paper termination proofs. To prove termination of a system like `map`, we could manually find a polynomial interpretation, or a symbol precedence for a path ordering, and obtain an elegant proof. However, in practical applications of rewriting, we might have systems with thousands of rules. This is for example the case with the compilers specified in CRSX (see Chapter 3.5). And even for small systems, there are situations imaginable where an application would like to check termination without asking a human to provide a proof first.

This is why in recent years, techniques for proving termination of first-order rewriting are only considered as valuable as their potential for efficient automation. Several tools have been written, which compete against each other in the annual *termination competition* [125]. In higher-order rewriting, too, there is a move towards automation. Since 2010, the termination competition has a higher-order category, on the AFS formalism from Chapter 3.4. The *termination problem database* [126], a database of termination problems, contains 156 benchmarks of this formalism in the current version (8.0.1).

Most of the techniques in this work were designed with automation in mind, and have been implemented in the tool WANDA. WANDA is one of the two tools which participated in the higher-order category of the termination competitions 2010 and 2011. In 2010 WANDA ended in second place, in 2011 in first place.

She – for let us imagine WANDA as a girl who, pen and paper in hand, tries to apply the results in this thesis to find termination proofs – follows roughly the following algorithm:

Input read the input from a text file in one of the supported formalisms, and transform it into the AFSM formalism used in this thesis;

Non-termination try some basic tricks to find a non-terminating term;

Rule Removal iteratively remove rules as long as we can, using polynomial interpretations and the recursive definition of StarHorpo;

Dependency Pairs if no rules remain, return YES, otherwise pass control to the dependency pair module, which also uses the modules for polynomial interpretations and recursive path orderings (as well as various other features from the dependency pair approach).

The algorithm returns YES if WANDA can prove termination, NO if she can prove non-termination, and MAYBE if either possibility fails.

WANDA has some minor extra functionality: given the right input flags, she will translate between formalisms, normalise an input term or check for properties like η-expandedness or left-linearity of the given rules. In addition, there are flags to disable parts of the functionality. This is in particular useful to empirically test the relative power of the various techniques.

Source. WANDA is open-source and can be downloaded from:

http://wandahot.sourceforge.net/

Chapter Setup. The setup of this chapter follows the rough line of computation in WANDA: Section 8.1 discusses the underlying formalism and input transformations, Section 8.2 deals with the non-termination module, Section 8.3 considers rule removal and in Section 8.4 we will study the implementation of dependency pairs. The next two chapters discuss the two ways of obtaining reduction pairs: polynomial interpretations in Section 8.5 and StarHorpo with argument functions in Section 8.6. Finally, in Section 8.7 we will see some experimental data.

Most of this chapter is original, but Section 8.4 is based on the algorithm in [80] and most of Section 8.5 has been published in [42].

8.1 Format and Transformations

Types. With an eye on future extensions, WANDA uses polymorphic types. These types could essentially be seen as first-order terms: they are generated from variables (identified by a unique index) and *sort constructors* of a given arity (each sort constructor identified by a string) using the signature:

$$\sigma = \alpha \mid c(\sigma_1, \ldots, \sigma_n) \qquad \alpha \text{ a type variable and } c \text{ a sort constructor of arity } n$$

Every occurrence of a function symbol, variable or meta-variable is equipped with a type declaration, as described in Chapter 9.3. This makes it easy to quickly derive the type of a (meta-)term, but otherwise makes no difference to saving the types of all variables and meta-variables in a separate set.

> *Comment:* Although WANDA supports polymorphic input, and takes care not to violate soundness when presented with a polymorphic system, neither the current implementation of polynomial interpretations nor the implementation of StarHorpo supports polymorphism. Thus, the most likely output for such a system is MAYBE.

Terms and Rewriting. Other than the more advanced typing, WANDA uses the AFSM formalism as described in Chapter 2. Following Theorem 2.10, all terms are presented in *applicative* syntax. Various modules in the program use an arity (effectively using the applicative terms as though they are functional), so where it is needed, a maximal arity for the function symbols is dynamically calculated. This choice was made to allow for easy swapping between arities. For instance, when a rule is removed, some symbols might be assigned a higher arity.

Rewrite rules are pairs of meta-terms. There is some limited functionality for rewriting terms, but this functionality is not optimised, since for the most part, rewriting is not necessary when trying to prove termination.

Transformations. As we have seen in Chapter 2, the common formalisms can generally be translated to AFSMs without losing non-termination. At present, a transformation from AFSs to AFSMs has been implemented. The algorithm used has been described in Section 3.4.4: we first simplify AFSs to not have leading free variables and left-hand abstractions, and then turn variables into meta-variables without arguments. No other input formalisms than AFSs and AFSMs are accepted at the moment.

The choice to focus on monomorphic AFSs was made for practical reasons: the higher-order category in the annual termination competition uses this formalism, and consequently there is a whole database available with benchmarks in this category. Other formalisms will likely be added as the need for support arises; the transformations to support for instance PRSs are relatively little work. With the eye on future expansion, there are two flags, "attempt to prove non-termination" (set to true for AFSs, since the relevant transformations preserve termination as well as non-termination), and "use a β-first reduction strategy" (set to false for AFSs). Currently, both flags are only used for the non-termination module, although in the future it may also be worthwhile to consider how the strategy affects the dependency graph in the dependency pair framework.

8.2 Proving Non-termination

Although this work contains no dedicated methods to prove non-termination, we can use a number of easy observations:

Obvious Loop. If $s \Rightarrow_{\mathcal{R}}^{+} C[s\gamma]$, for some term s, substitution γ and context C, then the system is evidently non-terminating: $s \Rightarrow_{\mathcal{R}}^{+} C[s\gamma] \Rightarrow_{\mathcal{R}}^{+} C[C\gamma[s\gamma^2]] \Rightarrow_{\mathcal{R}}^{+} \ldots$ This is called *looping* non-termination. Various techniques for loop detection are discussed in e.g. [47].

At present, WANDA just does a very shallow check to detect looping non-termination: we simply take all the left-hand sides of rules, reduce them a few times, and see whether we get back to an instance of the same rule!

More precisely, for all rules $l \Rightarrow r$ we do the following:

1. consider the substitution
   $\gamma = [Z := \lambda x_1 \ldots x_n.y \cdot \vec{x} \mid Z : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in FMV(l)]$
   and let $l' := l\gamma$, so l' is a term;
2. perform a breadth-first search on reducts of l', but do not go beyond 1000 reducts;
3. if any of these reducts has a subterm which is an instance of l', we have a counterexample.

This is a very simple strategy of finding counterexamples for termination, which is not going to catch any sophisticated loops. However, it is quick and easy, quite possibly useful to spot mistakes in a recursive call, and proves non-termination for seven benchmarks in the termination problem database.

When using a β-first reduction strategy, the approach should be adapted in two ways. First, when reducing l' (step 2 of the algorithm), we must β-normalise after every step. Second, we have to be sure that $C\gamma^n[] \downarrow_\beta$ does, in fact, contain $\Box_\sigma$ for every n. To this end, step 3 in the algorithm is changed to only consider *reachable* subterms. When not using a strategy, all subterms are considered reachable. Reachability with a β-first reduction strategy is defined as follows:

- each term is reachable in itself;
- t is reachable in $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ if t is reachable in some s_i;
- t is reachable in $\lambda x.s$ if t is reachable in s;
- t is reachable in $x \cdot s_1 \cdots s_n$ if $x \notin dom(\gamma)$ and t is reachable in some s_i.

Again, this is a very blunt way of checking that $C\gamma^n \updownarrow_\beta^\eta$ always contains $\Box_\sigma$. It is possible that future versions of WANDA will use more permissive checks.

Admits Ω Counterexample. A commonly used example to demonstrate non-termination of the untyped λ-calculus is the following: let $\omega := \lambda x.x \cdot x$ and $\Omega := \omega \cdot \omega$. Then Ω self-reduces in a single step. If we encode the untyped λ-calculus as an AFSM, for instance

$$\mathcal{F} = \left\{ \begin{array}{lcl} \mathtt{App} & : & [\mathtt{term} \times \mathtt{term}] \longrightarrow \mathtt{term} \\ \mathtt{Lam} & : & [\mathtt{term} \to \mathtt{term}] \longrightarrow \mathtt{term} \end{array} \right\}$$

$$\mathcal{R} = \{\mathtt{App}(\mathtt{Lam}(F), Z) \Rightarrow F \cdot Z\}$$

the same counterexample corresponds to the term $\Omega := \mathtt{App}(\omega, \omega)$, where $\omega = \mathtt{Lam}(\lambda x.\mathtt{App}(x, x))$. Studying this higher-order version of the example more closely, the problem is that a functional term which appears inside a base-type term is taken out of its context and applied to something else.

Generalising this example, suppose we have a rule of the form $C[D[F], X_1, ..., X_n] \Rightarrow_\mathcal{R} E[F \cdot X_1 \cdots X_n]$, where $F : \sigma_1 \to \ldots \to \sigma_n \to \tau$ and $C[] : \tau$. Moreover, suppose that for some $j \in \{1, \ldots, n\}$ the types of $D[]$ and X_j correspond. Define:

$$\begin{array}{rcl}\omega & = & D[\lambda x_1 \ldots x_n.C[x_j, x_1, \ldots, x_n]] \\ \Omega & = & C[\omega, y_1, \ldots, y_{j-1}, \omega, y_{j+1}, \ldots, y_n]\end{array}$$

Then indeed $\Omega \Rightarrow_{\mathcal{R}} (\lambda x_1 \ldots x_n.C[x_j, x_1, \ldots, x_n]) \cdot y_1 \cdots y_{j-1} \cdot \omega \cdot y_{j+1} \cdots y_n \Rightarrow_\beta^* C[\omega, y_1, \ldots, y_{j-1}, \omega, y_{j+1}, \ldots, y_n] = \Omega$.

WANDA checks whether any rules satisfy the conditions described above. To increase the chance of this happening, simple meta-variables are first "flattened" as done in Transformation 2.18, and rules with a functional type are appended with meta-variables. This way, we also catch examples like the rule $\mathtt{f}(F) \Rightarrow F$ with $\mathtt{f} : [\mathtt{o} \to \mathtt{o}] \longrightarrow \mathtt{o} \to \mathtt{o}$.

There are many more checks that we might consider, and first-order methods that can be extended to the higher-order setting; see for instance [33, 47].

Rather than extending the existing methods, it may also be interesting to determine whether we can translate a (part of) a higher-order system to a first-order TRS with preservation of termination. Having that, we could ask an external first-order termination tool whether the resulting system is non-terminating, and if so, conclude that the original AFSM is non-terminating as well.

A partial result already exists: if the first-order part of a TRS is locally confluent and overlay, then types do not matter for termination, so non-termination of this first-order part implies non-termination of the higher-order part. We used this in Theorem 7.54, and this result is used as part of the dependency pair framework (where the analysis of the first-order part of $\mathcal{R}$ takes place). Further use of first-order tools, or direct implementations of higher-order non-termination techniques, are left for future work.

8.3 Rule Removal

WANDA's rule removal module provides a completely straightforward implementation of Theorem 2.23, using the following algorithm:

```
do
   calculate maximal arity for the function symbols
         and consider the rules in a functional form,
         respecting this arity
   call polynomial interpretations module with parameters
         STRONGLY MONOTONIC and l (≿)_(∼) r for all rules
   if any rules could be oriented with ≻:
      remove these rules from the problem
   if not:
      call StarHorpo module with parameters PRESERVING
            and l (≿)_(∼) r for all rules
      if any rules could be oriented with ≻:
         remove these rules from the problem
while any rules were removed by either method
```

So first, a maximal arity is calculated for the function symbols, since both the polynomial interpretation module and the path ordering module work best with functional terms (applicative terms which respect a given arity for the function symbols can be seen as functional terms). Then WANDA attempts to find a polynomial interpretation which satisfies the requirements. By Theorem 4.16 and the definition of an extended monotonic algebra, the polynomials which are used must be strongly monotonic.

If the module for polynomial interpretation manages to orient all rules weakly, and orient some of them strictly, the latter ones can be removed, and the loop continues. If the polynomial module fails, the path ordering module is called. This module combines StarHorpo with argument functions, creates a weak reduction pair following Theorem 5.39. If instructed to use argument preserving functions (as is done by passing the parameter `PRESERVING`), the result is even strongly monotonic, which is necessary for rule removal.

If any rules are removed, the loop continues, trying to remove more rules. Note that the next iteration might use a different arity for the function symbols – this is valid by Theorem 2.10. If no rules can be removed anymore, the remaining system is returned, to be passed to the dependency pair framework.

8.4 The Dependency Pair Framework

In Chapter 7 we considered the following algorithm to prove termination using dependency pairs:

```
let PROBS := {(DP(R), R, minimal, formative)}
while PROBS is non-empty do:
   let A be an element of PROBS
   choose a sound processor Proc such that Proc(A) ≠ NO
   let PROBS := PROBS \ {A}  ∪  Proc(A)
end while
return YES
```

To this basic algorithm we should add the use of static dependency pairs if the dynamic approach fails (or immediately in the case of a strongly plain function passing system), but that is about it!

Of course, the trick is to choose the right processor. We will consider an algorithm that uses only processors which leave the set $\mathcal{R}$ alone: formative rules are combined in the reduction pair processor. This is mostly for historic reasons: when the dependency pair module in WANDA was developed, it was designed around the *chain-free* notion. There has been little reason to rewrite it, since most processors just map a DP problem $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{formative})$ to one or more problems $(\mathcal{P}', \mathcal{R}, \texttt{minimal}, \texttt{formative})$ with $\mathcal{P}' \subseteq \mathcal{P}$.

A Dependency Pair Algorithm. In this algorithm we iterative over a dependency graph, removing nodes as the corresponding dependency pairs are removed by processors. This algorithm roughly corresponds with the sketch above, if you think of PROBS as the set of all problems $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{formative})$ where $\mathcal{P}$ is an SCC in the graph.

The outermost part of the algorithm we copy from Chapter 7.8:

```
if R is strong plain function passing:
   return DPframework(STATIC)
answer := DPframework(DYNAMIC)
if answer = YES or answer = NO:
   return answer
if R is plain function passing:
   if function symbols occur in partial applications in R:
      η-expand R following Transformation 2.14
   answer := DPframework(STATIC)
   if answer = YES: return YES
return MAYBE
```

Here, DPframework is the algorithm which consists of the following steps:

1. **β-saturate the system, calculate dependency pairs and the graph and do initial checks.**
   - β-saturate the system (Definition 6.15);
   - determine its dependency pairs according to the DYNAMIC or STATIC parameter (Definition 6.20 in the former case, Definition 7.58 in the latter);
   - determine whether $\mathcal{R}$ is abstraction-simple (Definition 6.48);
   - calculate an approximation G for the dependency graph (Section 7.4), altering collapsing dependency pairs wherever this helps (Section 7.3); more details about this are given later on in this section.
2. **Remove first-order dependency pairs, if possible.**

   determine $\mathcal{R}_{\texttt{TFO}}$ (Section 7.7);

   if $\mathcal{R}$ viewed as a CRS is orthogonal (Section 3.7.3):
   - use an external first-order termination tool to determine whether $\mathcal{R}_{\texttt{TFO}}$ is terminating as a many-sorted TRS;
   - if *yes*, remove all first-order dependency pairs from the graph;
   - if *no*, return NO (which indicates non-termination);
   - if we cannot find an answer, do nothing;

   alternatively, if $\mathcal{R}$ is finitely branching:

- use an external first-order termination tool to determine whether $\mathcal{R}_{\texttt{TFO}} \cup \{\mathtt{p}_\iota(X,Y) \Rightarrow X,\ \mathtt{p}_\iota(X,Y) \Rightarrow Y \mid \iota \text{ a base type }\}$ is terminating as a many-sorted TRS;
- if *yes*, remove all first-order dependency pairs from the graph;
- otherwise, do nothing.

3. **Select a suitable strongly connected component, if any exist.**

   Remove all nodes and edges from G which are not part of a cycle; if G is empty, return YES, otherwise choose an SCC $\mathcal{P}$ (Section 7.4).

4. **Apply the subterm criterion, if possible.**

   If $\mathcal{P}$ is non-collapsing, try finding a projection function ν (Section 7.5) such that $\overline{\nu}(l) \trianglerighteq \overline{\nu}(p)$ for all $l \Rrightarrow p\ (B) \in \mathcal{P}$, and this inequality is strict for at least one dependency pair; if this succeeds, remove those dependency pairs such that $\overline{\nu}(l) \triangleright \overline{\nu}(p)$ from the graph, and continue with (3).

5. **Determine the constraints for a reduction pair.**

   Let $FR := FR(\mathcal{P},\mathcal{R})$ (Definition 6.38). Recall that $\mathcal{F}_c^\sharp = \mathcal{F} \cup \{f^\sharp : \sigma \mid f : \sigma \in \mathcal{D}\} \cup \mathsf{C}$, as defined at the end of Section 6.3.2.

   a. If $\mathcal{P}$ is non-collapsing:

   - let $S := \emptyset$ and $\Sigma := \mathcal{F}_c^\sharp$;
   - let $\psi(s)$ be defined as just s;
   - let $A := UR(\mathcal{P}, FR)$ (Definition 7.37);

   b. If $\mathcal{P}$ is collapsing and $\mathcal{R}$ is not abstraction-simple:

   - let $S := \mathcal{F}$ and $\Sigma := \mathcal{F}_c^\sharp$;
   - let $\psi(s)$ be defined as just s;
   - let $A := FR \cup \{f(\vec{x}) \Rightarrow f^\sharp(\vec{x}) \mid f \in \mathcal{D}\}$;

   c. If $\mathcal{P}$ is collapsing and $\mathcal{R}$ is abstraction-simple:

   - let S be the set of function symbols $f^- : \sigma$ where $f : \sigma \in \mathcal{F}$ and f occurs below an abstraction in a right-hand side of $FR(\mathcal{P},\mathcal{R}) \cup \mathcal{P}$, and let $\Sigma := \mathcal{F}_c^\sharp \cup S$;
   - let $\psi(s)$ be defined as $\mathsf{tag}(s)$ (Definition 6.50);
   - let $A := \{l \Rightarrow \mathsf{tag}(r) \mid l \Rightarrow r \in FR(\mathcal{P},\mathcal{R})\} \cup \{f^-(\vec{x}) \Rightarrow f(\vec{x}) \mid f^- \in S\} \cup \{f^-(\vec{x}) \Rightarrow f^\sharp(\vec{x}) \mid f^- \in S,\ f \in \mathcal{D}\}$.

6. **Find a suitable reduction pair**

   For all dependency pairs $l \Rrightarrow p\ (A) \in \mathcal{P}$, let $\tilde{l} := l \cdot Z_1 \cdots Z_n$ for fresh meta-variables $Z_1, \ldots, Z_n$ (a base-type meta-term), and let $\overline{p} := \psi(p[\vec{x} := \vec{\mathsf{c}}]) \cdot \mathsf{c}_1^{\sigma_1} \cdots \mathsf{c}_m^{\sigma_m}$ (a base-type meta-term), where $\{\vec{x}\} = FV(p)$ (see Section 6.3.2 for the introduction of the c_i^{σ}). Use a type changing function to collapse all base types in $\tilde{l}$ and $\overline{p}$ to the single base type $\mathtt{o}$.

   Call the polynomial interpretations module with the parameter `WEAKLY MONOTONIC` if $\mathcal{P}$ is non-collapsing, and `SEMI-WEAKLY MONOTONIC` if $\mathcal{P}$ is collapsing, and constraints $\tilde{l} \succsim_{(\sim)} \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}$ and $l \succsim r$ for $l \Rightarrow r \in \mathcal{R}$. Also instruct the module that the elements of S should be interpreted with a function F which has the property that $F(\vec{n}) \sqsupseteq n(\vec{0})$ for all its arguments.

   If this succeeds, let $\mathcal{P}_1$ consist of those dependency pairs such that $\tilde{l} \succ \overline{p}$ and $\mathcal{P}_2$ of the remainder.

   If it does not succeed, call the path orderings module with the parameter `NON-PRESERVING` if $\mathcal{P}$ is non-collapsing, and `PARTLY PRESERVING` if $\mathcal{P}$ is collapsing, and constraints $\tilde{l} \succsim_{(\sim)} \overline{p}$ for $l \Rrightarrow p\ (A) \in \mathcal{P}$ and $l \succsim r$ for $l \Rightarrow r \in \mathcal{R}$. Also instruct the module that for the elements f of S we must have $f(\vec{s}) \cdot \vec{t} \succsim s_i \cdot \vec{\mathsf{c}}$ if both sides have base type.

   If this succeeds, let $\mathcal{P}_1$ consist of those pairs such that $\tilde{l} \succ \overline{p}$ and $\mathcal{P}_2$ of the remainder.

   if this, too, fails, return `MAYBE`.

7. **Continue with the remaining dependency pairs.**

   Remove all pairs in $\mathcal{P}_1$ from the graph, and continue with (3).

This elaborate algorithm summarises most of the results of Chapters 6 and 7. As previously remarked, we might think of the set `PROBS` as the set of all DP problems $(\mathcal{P}, \mathcal{R}, \mathtt{minimal}, \mathtt{formative})$ where $\mathcal{P}$ is an SCCs in the graph approximation that we iterate over. This, together with Theorem 7.31, justifies the way the graph is used in the algorithm (if we use the observation that having a dependency graph G, the subgraph of G which contains only the nodes of $\mathcal{P}$ is exactly the dependency graph of $\mathcal{P}$). An upside of this method is that we do not constantly need to regenerate the graph, and also the methods to get a nicer graph (altering collapsing dependency pairs) are done only once.

Step (2) uses the result of Theorem 7.55. In some cases we can return `NO` because Theorem 7.53 gives an equivalence, and in the orthogonal first-order setting types may be ignored without affecting terminating. Note that if first-order methods fail, higher-order methods (such as the formative rules, which fundamentally use the types of the system) may still succeed, so if we cannot decide that the first-order part is terminating, we continue with the full problem.[1]

[1] At the moment, the processors of Theorem 7.48 and 7.54 are not used separately. However, this may be a useful extension for a later time.

In Steps 3–7 the algorithm boils down to: "while there are sets which must be proved formative chain-free (these are the SCCs of G), try to remove dependency pairs from these sets".

Step (3), removing those nodes and edges which are not on an SCC, represents the dependency graph processor, Theorem 7.31. Step (4) implements the subterm criterion (Theorem 7.35). In Steps (5) and (6), the problem "is $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{formative})$ finite" is reduced to the same problem for a strict subset $\mathcal{P}_2$ of $\mathcal{P}$, by using a reduction pair with formative rules: under the hood, $(\mathcal{P}, \mathcal{R}, \texttt{minimal}, \texttt{formative})$ is mapped to $(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}), \texttt{minimal}, \texttt{formative})$, and then the reduction pair modules essentially split $\mathcal{P}$ into two groups: $\mathcal{P}_1$, those pairs for which $\tilde{l} \succ \overline{p}$, and $\mathcal{P}_2$, those pairs for which $\tilde{l} \succsim \overline{p}$. The resulting dependency pair problem, $(\mathcal{P}_2, FR(\mathcal{P}, \mathcal{R}), \texttt{minimal}, \texttt{formative})$ is of course finite if $(\mathcal{P}_2, \mathcal{R}, \texttt{minimal}, \texttt{formative})$ is.

To see that all restrictions are satisfied, consider each of the cases:

- $\mathcal{P}$ is non-collapsing: By Theorem 7.44 it suffices to see that a standard reduction pair exists for $(\mathcal{P}_1, \mathcal{P}_2, UR(\mathcal{P}, FR(\mathcal{P}, \mathcal{R})))$ (since $\mathcal{R}$ is obviously finitely branching, as we are dealing with a finite system). Step (6) provides such a standard reduction pair. Since S is empty, no additional constraints are given to the modules for polynomial interpretations or path orderings (the parameters `WEAKLY MONOTONIC` and `NON-PRESERVING` respectively give maximum freedom, as we will discuss in Sections 8.5 and 8.6).

- $\mathcal{P}$ is collapsing and $\mathcal{R}$ is not abstraction-simple; by Theorem 6.71 we should find a standard reduction pair for $(\mathcal{P}_1, \mathcal{P}_2, FR(\mathcal{P}, \mathcal{R}))$, so a reduction pair such that:
  - $l \cdot \vec{Z} \succ (p\gamma) \cdot \vec{s}$ for $l \Rrightarrow p \in \mathcal{P}_1$, for some substitution γ on $FV(p)$ and terms $\vec{s}$ for $l \Rrightarrow p \in \mathcal{P}_1$; this is satisfied by choice for $\tilde{l}, \overline{p}$ in (6);
  - $l \cdot \vec{Z} \succsim (p\gamma) \cdot \vec{s}$ for $l \Rrightarrow p \in \mathcal{P}_2$, similar;
  - $l \succsim r$ for $l \Rightarrow r \in FR(\mathcal{P}, \mathcal{R})$: holds because $FR(\mathcal{P}, \mathcal{R}) \subseteq A$;
  - $f(\vec{Z}) \succsim f^\sharp(\vec{Z})$ because a corresponding "rule" appears in A (note that, since these "rules" do not necessarily respect types, they should more be seen as a pair of terms than a rule);
  - $\rhd^{\mathcal{F}}$ is included in $\succsim$ because the modules for polynomial interpretations and path orderings are instructed that the elements of $\mathcal{F}$ should be interpreted with strongly monotonic functions in the first case, or the argument function should have a harmless form in the second case (more about this in the relevant sections). Also, this holds because of the parameter `SEMI-WEAKLY MONOTONIC` or `PARTLY PRESERVING`; these parameters enforce the default requirements from Section 6.6.1 and the standard properties from Section 6.6.2.

- $\mathcal{P}$ is collapsing and $\mathcal{R}$ is abstraction-simple; since $\mathcal{P} \subseteq \mathsf{DP}(\mathcal{R})$ also $\mathcal{P}$ is abstraction-simple. By Theorem 6.71 we must find a tagged reduction pair for $(\mathcal{P}_1, \mathcal{P}_2, FR(\mathcal{P}, \mathcal{R}))$, so a reduction pair such that:
  - $l \cdot \vec{Z} \succ (\mathsf{tag}(p)\gamma) \cdot \vec{s}$ for $l \Rrightarrow p \in \mathcal{P}_1$, for some substitution γ on $FV(p)$ and terms $\vec{s}$ for $l \Rrightarrow p \in \mathcal{P}_1$; this is satisfied by choice for $\tilde{l}, \overline{p}$ in (6);
  - $l \cdot \vec{Z} \succsim (\mathsf{tag}(p)\gamma) \cdot \vec{s}$ for $l \Rrightarrow p \in \mathcal{P}_2$, similar;
  - $l \succsim r$ for $l \Rightarrow r \in FR(\mathcal{P}, \mathcal{R})$: holds because $FR(\mathcal{P}, \mathcal{R}) \subseteq A$;
  - $f^-(\vec{Z}) \succsim f(\vec{Z}), f^\sharp(\vec{Z})$ is guaranteed for those f^- which actually occur in rules or dependency pairs because corresponding constraints are included in A, and for the other symbols f^- (which do not occur), we can assume a standard interpretation or argument function as discussed in Section 6.6;
  - $\rhd^-$ is included $\succsim$ because the modules for polynomial interpretations and path orderings are instructed that the elements of S should be interpreted with strongly monotonic functions or have a harmless translation with the argument function, and by the parameter `SEMI-WEAKLY MONOTONIC` or `PARTLY PRESERVING`.

Thus, each step of the algorithm is justified.

It is worth stating that the generation of dependency pairs assigns all function symbols a maximum arity, which is necessary because technically the system is applicative. Then, the framework treats the rules and dependency pairs as though the function symbols used this arity, reading a subterm $f \cdot s \cdot t$ with $ar(f) = 1$ as $f(s) \cdot t$. This arity is preserved throughout the framework, but not passed on to the modules for polynomial interpretations and path orderings. These modules, in keeping with Theorem 2.26, assign their own arity.

Furthermore, this algorithm is somewhat simplified from WANDA's actual workings. She does a few more things: for example, in some cases (in particular when a constraint $f(Z_1, \ldots, Z_n) \succsim r$ occurs) argument functions are used even in combination with polynomial interpretations (this is justified by Theorem 6.75), or a type changing function is applied that does not just collapse types.

Example 8.1. Consider the system `from`, which appears in the termination problem database under the same name. It has the following function symbols:

`0`	:	`nat`	`if`	:	$[\mathtt{bool} \times \mathtt{list} \times \mathtt{list}] \longrightarrow \mathtt{list}$
`s`	:	$[\mathtt{nat}] \longrightarrow \mathtt{nat}$	`lteq`	:	$[\mathtt{nat} \times \mathtt{nat}] \longrightarrow \mathtt{bool}$
`true`	:	`bool`	`from`	:	$[\mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list}$
`false`	:	`bool`	`chain`	:	$[(\mathtt{nat} \to \mathtt{nat}) \times \mathtt{list}] \longrightarrow \mathtt{list}$
`nil`	:	`list`	`incch`	:	$[\mathtt{list}] \longrightarrow \mathtt{list}$
`cons`	:	$[\mathtt{nat} \times \mathtt{list}] \longrightarrow \mathtt{list}$			

And rules:

$$\begin{array}{rcl@{\qquad}rcl}
\mathtt{lteq}(\mathtt{s}(X),\mathtt{0}) & \Rightarrow & \mathtt{false} & \mathtt{if}(\mathtt{true},X,Y) & \Rightarrow & X \\
\mathtt{lteq}(\mathtt{0},X) & \Rightarrow & \mathtt{true} & \mathtt{if}(\mathtt{false},X,Y) & \Rightarrow & Y \\
\mathtt{lteq}(\mathtt{s}(X),\mathtt{s}(Y)) & \Rightarrow & \mathtt{lteq}(X,Y) & \mathtt{from}(X,\mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{incch}(X) & \Rightarrow & \mathtt{chain}(\lambda y.\mathtt{s}(y),X) & \mathtt{chain}(F,\mathtt{nil}) & \Rightarrow & \mathtt{nil} \\
\mathtt{from}(X,\mathtt{cons}(Y,Z)) & \Rightarrow & \mathtt{if}(\mathtt{lteq}(X,Y),\mathtt{cons}(Y,Z),\mathtt{from}(X,Z)) & & & \\
\mathtt{chain}(F,\mathtt{cons}(Y,Z)) & \Rightarrow & \mathtt{cons}(F\cdot Y,\mathtt{chain}(F,\mathtt{from}(F\cdot Y,Z))) & & &
\end{array}$$

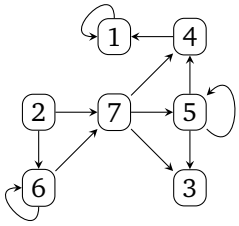
Let us follow the algorithm and see whether we can prove this system terminating. This system is strongly plain function passing, so we use the algorithm with a parameter `STATIC`.

(1) *β-saturate the system, calculate the dependency pairs and graph and do initial checks.*

The system does not need to be β-saturated, is abstraction-simple (although this will not matter, as we are using static dependency pairs), and has the following static dependency pairs:

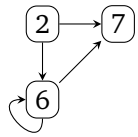
1. $\mathtt{lteq}^\sharp(\mathtt{s}(X),\mathtt{s}(Y)) \Rrightarrow \mathtt{lteq}^\sharp(X,Y)$
2. $\mathtt{incch}^\sharp(X) \Rrightarrow \mathtt{chain}^\sharp(\lambda y.\mathtt{s}(y),X)$
3. $\mathtt{from}^\sharp(X,\mathtt{cons}(Y,Z)) \Rrightarrow \mathtt{if}^\sharp(\mathtt{lteq}(X,Y),\mathtt{cons}(Y,Z),\mathtt{from}(X,Z))$
4. $\mathtt{from}^\sharp(X,\mathtt{cons}(Y,Z)) \Rrightarrow \mathtt{lteq}^\sharp(X,Y)$
5. $\mathtt{from}^\sharp(X,\mathtt{cons}(Y,Z)) \Rrightarrow \mathtt{from}^\sharp(X,Z)$
6. $\mathtt{chain}^\sharp(F,\mathtt{cons}(Y,Z)) \Rrightarrow \mathtt{chain}^\sharp(F,\mathtt{from}(F\cdot Y,Z))$
7. $\mathtt{chain}^\sharp(F,\mathtt{cons}(Y,Z)) \Rrightarrow \mathtt{from}^\sharp(F\cdot Y,Z)$

They can be presented in the following dependency graph (which is also the approximation which WANDA calculates):



(2) *Remove first-order dependency pairs, if possible.*

The `lteq`, `from` and `if` rules are all first-order; the `incch` and `chain` rules are not. The system is orthogonal, so we can safely send $\mathcal{R}_{\mathtt{TF0}}$ to a first-order prover unmodified. This results in a quick *yes*. Thus, we can remove the dependency pairs which originate from these rules from the graph, which leaves:



(3) *Select a suitable strongly connected component, if any exist.*

Removing all nodes and edges which are not part of a cycle, we are left with the following rather boring graph:

We (obviously) select $\mathcal{P} = \{\mathtt{chain}^\sharp(F, \mathtt{cons}(Y, Z)) \Rrightarrow \mathtt{chain}^\sharp(F, \mathtt{from}(F \cdot Y, Z))\}$.

(4) *Apply the subterm criterion, if possible.*

This is not possible: although $\mathcal{P}$ is non-collapsing, neither $\nu(\mathtt{chain}^\sharp) = 1$ nor $\nu(\mathtt{chain}^\sharp) = 2$ gives a strict subterm inclusion.

(5) *Determine the constraints for a reduction pair.*

Since $\mathcal{P}$ is non-collapsing, we have $S = \emptyset,\ \Sigma = \mathcal{F}_c^\sharp,\ \psi$ the identity function, and $A := UR(\mathcal{P}, FR(\mathcal{P}, \mathcal{R}))$. Since the right-hand side of our one dependency pair is not a pattern, all elements of $FR(\mathcal{P}, \mathcal{R})$ are usable. $FR(\mathcal{P}, \mathcal{R})$ consists of all rules except the two which reduce to `nil`.

(6) *Find a suitable reduction pair.*

We must find a reduction pair such that $\mathtt{chain}^\sharp(F, \mathtt{cons}(Y, Z)) \succ \mathtt{chain}^\sharp(F, \mathtt{from}(F \cdot Y, Z))$, and $l \succsim r$ for all rules except the two which reduce to `nil`. This is quickly satisfied with StarHorpo, using an argument function:

$$\begin{array}{rclrcl} \pi(\mathtt{s}) & = & \lambda x.x & \pi(\mathtt{incch}) & = & \lambda x.\mathtt{chain}(\lambda y.y, x) \\ \pi(\mathtt{lteq}) & = & \lambda xy.\mathtt{lteq}' & \pi(\mathtt{from}) & = & \lambda xy.\mathtt{from}'(y) \end{array}$$

For other symbols, π is the identity. Using a precedence $\mathtt{chain}^\sharp \blacktriangleright \mathtt{cons} \blacktriangleright \mathtt{from}' \blacktriangleright \mathtt{if} \approx \mathtt{lteq}' \blacktriangleright \mathtt{false} \approx \mathtt{true}$ both the dependency pair is oriented as required, and all the rules are as well:

$$\begin{array}{rcl} \mathtt{chain}^\sharp(F, \mathtt{cons}(Y, Z)) & \succ & \mathtt{chain}^\sharp(F, \mathtt{from}'(Z)) \\ \mathtt{lteq}'\ \mathtt{true}\ \mathtt{true} & \succsim & \mathtt{false} \\ \mathtt{lteq}'\ X\ X & \succsim & \mathtt{true} \\ \mathtt{lteq}'\ \mathtt{s}(X)\ \mathtt{s}(Y) & \succsim & \mathtt{lteq}'\ X\ Y \\ \mathtt{if}(\mathtt{true}, X, Y) & \succsim & X \\ \mathtt{if}(\mathtt{false}, X, Y) & \succsim & Y \\ \mathtt{chain}(\lambda y.y, X) & \succsim & \mathtt{chain}(\lambda y.y, X) \\ \mathtt{from}'(\mathtt{cons}(Y, Z)) & \succsim & \mathtt{if}(\mathtt{lteq}'\ Y\ Y, \mathtt{cons}(Y, Z), \mathtt{from}'(Z)) \\ \mathtt{chain}(F, \mathtt{cons}(Y, Z)) & \succsim & \mathtt{cons}(F \cdot Y, \mathtt{chain}(F, \mathtt{from}'(Z))) \end{array}$$

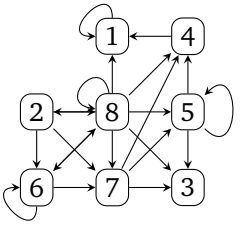
(7) *Continue with the remaining dependency pairs.*

Removing the strictly oriented dependency pair from the graph, G is empty.

(3) *Select a suitable strongly connected component, if any exist.*

The graph contains no further cycles; the algorithm returns `YES`.

In Example 8.1 we could also have got away with the `DYNAMIC` tag instead of the `STATIC` one. This would have led to a graph with one collapsing dependency pair:



Here, (8) is the pair $\texttt{chain}^\sharp(F, \texttt{cons}(Y, Z)) \Rrightarrow F \cdot Y$. With this choice, the algorithm would proceed almost the same, only in some places `s` is tagged, $\texttt{s}^-$.

Generating the Dependency Graph. In Step 1 of the algorithm, we generate an approximation of the dependency graph of $\mathsf{DP}(\mathcal{R})$, possibly changing dependency pairs along the way to give the graph fewer edges. Rather than extending the usual first-order methods, the dependency graph in WANDA is generated in a way that leaves more room for optimisations specific to the higher-order setting.

At the present time, the optimisations of Chapter 7.3 have not been implemented. This is planned for future upgrades. Otherwise, the graph is generated with the following steps:

- take all dependency pairs in $\mathsf{DP}(\mathcal{R})$ as nodes: since any system that can be input to WANDA is finite, this always gives a finite approximation;[2]
- add an edge from all collapsing dependency pairs to all nodes;
- add an edge from a pair $l \Rrightarrow f(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m \ (A)$ to any pair of the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \Rrightarrow p \ (B)$ if for all i it seems like a term of the form $p_i\gamma$ may reduce to a term of the form $l_i\delta$, provided δ respects B.

The last step is somewhat dubious: what does it mean that "a term of the form $p_i\gamma$ may reduce to a term of the form $l_i\delta$ provided δ respects B"?

To properly define this, let us first introduce a relation on typed symbols $\langle f, \sigma\rangle$ with $f \in \mathcal{F} \cup \{ABS, VAR\}$ and σ a type. This relation is somewhat similar to both $\sqsubseteq_{fo}$ and $\sqsupseteq_{us}$, but only considers the top shape of a meta-term.

$\langle f, \sigma\rangle \sqsupseteq_{top} \langle g, \sigma\rangle$ if there is a rule $f(\vec{s}) \cdot \vec{t} \Rightarrow g(\vec{q}) \cdot \vec{u}$ with both sides of type σ
$\langle f, \sigma\rangle \sqsupseteq_{top} \langle ABS, \sigma\rangle$ if there is a rule $f(\vec{s}) \cdot \vec{t} \Rightarrow \lambda x.q$ with both sides of type σ
$\langle f, \sigma\rangle \sqsupseteq_{top} \langle a, \sigma\rangle$ for any $a \in \mathcal{F} \cup \{ABS, VAR\}$ if there is a rule $f(\vec{s}) \cdot \vec{t} \Rightarrow Z(\vec{q}) \cdot \vec{u}$ with both sides of type σ

[2]The dependency pair framework in WANDA can also deal with polymorphic input to some extent, in which case a finite approximation of an in principle infinite system is calculated. Since polymorphism is beyond the scope of this thesis, we shall ignore this for now.

Moreover, let $\sqsupset_{top}^*$ be the reflexive-transitive closure of $\sqsupset_{top}$. Then it is clear that a term $(f(\vec{s})\cdot\vec{t})\gamma$ of type σ can only reduce to a term $(g(\vec{q})\cdot\vec{u})\delta$ if $\langle f,\sigma\rangle \sqsupset_{top}^* \langle g,\sigma\rangle$. Similarly, $(f(\vec{s})\cdot\vec{t})\gamma$ can only reduce to an abstraction if $\langle f,\sigma\rangle \sqsupset_{top}^* \langle ABS,\sigma\rangle$ and to a variable if $\langle f,\sigma\rangle \sqsupset_{top}^* \langle VAR,\sigma\rangle$.

Next, let us consider a definition which will particularly help us to identify cases where variables are "eaten" that might need to occur due to variable conditions. Given a set A of pairs $f:i$ of function symbols and an argument position, we say that a meta-variable Z is *safe* in a meta-term s, notation $s \trianglerighteq_A^{safe} Z$, if this can be derived with the following clauses:

$$\begin{array}{rcll}
s & \trianglerighteq_A^{safe} & Z & \text{if } s \triangleright_A^{safe} Z \text{ or } s = Z \text{ or} \\
 & & & s = Z(\vec{x}) \text{ with all } x_i \text{ variables} \\
f(s_1,\ldots,s_n)\cdot s_{n+1}\cdots s_m & \triangleright_A^{safe} & Z & \text{if some } s_i \trianglerighteq_A^{safe} Z \text{ and } f:i \in A \\
x\cdot s_1\cdots s_m & \triangleright_A^{safe} & Z & \text{if some } s_i \trianglerighteq_A^{safe} Z \\
\lambda x.s & \triangleright_A^{safe} & Z & \text{if } s \trianglerighteq_A^{safe} Z \\
(\lambda x.s)\cdot t\cdot q_1\cdots q_n & \triangleright_A^{safe} & Z & \text{if } s[x:=t]\cdot\vec{q} \triangleright_A^{safe} Z
\end{array}$$

For a variable y, we say that it is safe in a term s, notation $s \trianglerighteq_A^{safe} y$, if this can be derived with the same clauses (merely omitting the case where $s = Z(\vec{x})$).

As in Section 7.6, we consider $\mathcal{R}^{++}$, which is the set $\mathcal{R} \cup \{l\cdot Z_1\cdots Z_n \Rightarrow r\cdot Z_1\cdots Z_n \mid l \Rightarrow r \in \mathcal{R}$, all Z_i fresh meta-variables and $l\cdot\vec{Z}$ well-typed$\}$. Let A be the largest set of function symbol / argument position pairs such that, for any rule $f(l_1,\ldots,l_n)\cdot l_{n+1}\cdots l_m \Rightarrow r$ in $\mathcal{R}^{++}$: if $f:k \in A,\ k \leq m$ and l_k contains a meta-variable Z, then $r \trianglerighteq_A^{safe} Z$. Since $\mathcal{R}$ is finite, we can always calculate A (starting from $\mathcal{F}$ and removing elements until nothing further needs removing).

The idea of the relation $\triangleright_A^{safe}$ is this: if $s \trianglerighteq_A^{safe} x$ for some term s and variable x, then for all reducts t of s also x is safe in t. That is, using reductions, we cannot ever get rid of the variable x.

Now we are ready to give a meaning to the vague notion "a term of the form $p_i\gamma$ may reduce to a term of the form $l_i\delta$ provided δ respects B". We say s *may reduce under substitution* to t, B if:

- s and t have the same type;
- all variables occurring freely in t also occur in s, unless the variable occurrence in t may be "eaten": this is the case if it only occurs in a context $F(q_1,\ldots,C_i[x],\ldots,q_k)$ and $F:i$ is not in B;
- for any free variable x such that $s \trianglerighteq_A^{safe} x$, also x occurs in t;
- if $s = f(s_1,\ldots,s_n)\cdot s_{n+1}\cdots s_m$ and f is a constructor symbol, then $t = f(t_1,\ldots,t_n)\cdot t_{n+1}\cdots t_m$ and each s_i may reduce under substitution to t_i, B;
- if $s = x\cdot s_1\cdots s_n$ with x a variable, then $t = x\cdot t_1\cdots t_n$, and each s_i may reduce under substitution to t_i, B;

- if $s = \lambda x.s'$ then $t = \lambda x.t'$ and s' reduces under substitution to t', B;
- if $s = f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ with f not a constructor symbol, and $s : \sigma$, then:
  - if $t = \lambda x.t'$, then $\langle f, \sigma \rangle \sqsupseteq^*_{top} ABS$;
  - if t is a variable, then $\langle f, \sigma \rangle \sqsupseteq^*_{top} VAR$;
  - if $t = g(t_1, \ldots, t_k) \cdot t_{k+1} \cdots t_j$, then $\langle f, \sigma \rangle \sqsupseteq^*_{top} g$.

For each of these cases, we can easily see that it is indeed required to be able to reduce $s\gamma$ to $t\delta$ if δ is a substitution which respects B, provided the variables in s do not occur in domain or range of either substitution.

Now we can formally complete the definition of the dependency graph. We add a node from a dependency pair ρ_1 to a dependency pair ρ_2 if either ρ_1 is collapsing, or:

- ρ_1 has the form $l \Rrightarrow f(p_1, \ldots, p_n) \cdot p_{n+1} \cdots p_m$ (A), and
- ρ_2 has the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \Rrightarrow p$ (B), and
- for all i, if χ is a substitution which maps all free variables in p_i to fresh constructors $\mathtt{c}^\sigma_j$, then $p_i\chi$ may reduce under substitution to l_i.

Example 8.2. Consider the following toy example:

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathtt{a} & \Rightarrow & \mathtt{f}(\lambda x.\mathtt{g}(x)) \\ \mathtt{f}(\lambda x.X) & \Rightarrow & \mathtt{a} \\ \mathtt{g}(X) & \Rightarrow & \mathtt{h}(X, X) \\ \mathtt{h}(\mathtt{0}, X) & \Rightarrow & X \\ \mathtt{h}(\mathtt{s}(X), \mathtt{0}) & \Rightarrow & \mathtt{g}(X) \end{array} \right\}$$

In this system, the set A of "safe positions" of $\mathcal{R}$ should be chosen as $\{\mathtt{g} : 1,\ \mathtt{h} : 1,\ \mathtt{h} : 2\}$. Consequently, in the dependency graph (whether in the static or in the dynamic case), there is *not* an edge from the dependency pair $\mathtt{a}^\sharp \Rrightarrow \mathtt{f}^\sharp(\lambda x.\mathtt{g}(x))$ to the pair $\mathtt{f}^\sharp(\lambda x.X) \Rrightarrow \mathtt{a}^\sharp$. Which is lucky, because neither polynomial interpretations nor path orderings are equipped to deal with termination through such a technicality!

The dependency graph approximation, therefore, has but one cycle: the set $\{\mathtt{g}^\sharp(X) \Rrightarrow \mathtt{h}^\sharp(X, X),\ \mathtt{h}^\sharp(\mathtt{s}(X), \mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(X)\}$. This cycle is easily disposed of using for instance polynomial interpretations.

It is worth noting, however, that the real dependency graph does not have an edge from $\mathtt{g}^\sharp(X) \Rrightarrow \mathtt{h}^\sharp(X, X)$ to $\mathtt{h}^\sharp(\mathtt{s}(Z), \mathtt{0}) \Rrightarrow \mathtt{g}^\sharp(Z)$ either: the system is deterministic, so it is impossible to reduce the same term both to $\mathtt{0}$ and something of the form $\mathtt{s}(t)$. This is an improvement to consider for future upgrades to the dependency graph module.

Example 8.3. Consider the AFSM with $\mathtt{0} : \mathtt{nat}$, $\mathtt{f} : [\mathtt{nat}] \longrightarrow \mathtt{nat}$ and $\mathtt{g} : [\mathtt{nat} \to \mathtt{nat}] \longrightarrow \mathtt{nat}$, and two rules:

$$\begin{array}{rcl} \mathtt{f}(\mathtt{0}) & \Rightarrow & \mathtt{g}(\lambda x.\mathtt{0}) \\ \mathtt{g}(\lambda x.F(x)) & \Rightarrow & F(\mathtt{f}(\mathtt{0})) \end{array}$$

This system is strong plain function passing, so we can use static dependency pairs and do not need to consider the collapsing dependency pair $\mathtt{g}^\sharp(\lambda x.F(x)) \Rrightarrow F(\mathtt{f}(\mathtt{0}))$. This leaves just two dependency pairs:

$$\begin{array}{rcl} \mathtt{f}^\sharp(\mathtt{0}) & \Rrightarrow & \mathtt{g}^\sharp(\lambda x.\mathtt{0}) \\ \mathtt{g}^\sharp(\lambda x.F(x)) & \Rrightarrow & \mathtt{f}^\sharp(\mathtt{0}) \ \{F:1\} \end{array}$$

Now, $\mathtt{0}$ may not reduce under substitution to $F(x),\ \{F:1\}$, because $F(x)$ freely contains x (and cannot eat it!), while $\mathtt{0}$ does not. Thus, there is no edge in WANDA's dependency graph approximation from the first to the second pair. As the graph does not contain any cycles, we can immediately conclude termination.

The Subterm Criterion. We have yet to see how to use the subterm criterion automatically, and how to find a suitable reduction pair. The latter question will be discussed in some detail in Sections 8.5 and 8.6. The former is done by a conversion to a satisfiability problem, which is fed to an external SAT-solver.

SAT-solvers take as input a propositional formula in *Conjunctive Normal Form* (CNF): a formula of the form $\varphi_1 \wedge \ldots \wedge \varphi_n$, where each φ_i is a *clause*. A clause has the form $\psi_{i,1} \vee \ldots \vee \psi_{i,k_n}$, where each $\psi_{i,j}$ is either a propositional variable or the negation of one. The solver determines whether there is an assignment of true/false values to the variables which makes the input formula true and if so, returns this assignment. Although satisfiability is NP-complete, there are several automatic SAT-solvers which handle this problem. Due to clever use of heuristics, even problems with thousands of variables can often be solved in a short time.

Therefore, it should not be surprising that WANDA, following the examples of several first-order tools, does not try clever tricks to make choices to find for instance a projection function herself. Rather, she transforms the question into a satisfiability problem, and passes it off to the experts.

Recall that a projection function for a set $\mathcal{P}$ of non-collapsing dependency pairs is a function ν which assigns to all function symbols f a number i such that $1 \leq i \leq m$ for any number m such that $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ is the left- or right-hand side of a dependency pair in $\mathcal{P}$. For all relevant function symbols f, let m_f be the smallest such number m. We introduce the following variables:

- for all relevant function symbols f the variables $X_{f,1}, \ldots, X_{f,m_f}$, which indicate the choice for $\nu(f)$: if $X_{f,i}$ is true, then $\nu(f) = i$;
- for all dependency pairs $l \Rrightarrow p$ a variable $Y_{l,p}$, which indicates whether this dependency pair is strictly oriented: if $Y_{l,p}$ is true, then $\overline{\nu}(l) \rhd \overline{\nu}(p)$, otherwise $\overline{\nu}(l) = \overline{\nu}(p)$.

The satisfiability formula takes care of three things. First, that ν is well-defined; that is, for all f there is exactly one i such that $X_{f,i}$ is true. This holds if for all symbols f the following clauses are satisfied:

$$\begin{array}{ll} X_{f,1} \vee \ldots \vee X_{f,m_f} & \\ \neg X_{f,i} \vee \neg X_{f,j} & \text{for all } i,j \text{ with } 1 \leq i < j \leq m_f \end{array}$$

Second, for all dependency pairs $l \Rrightarrow p$, we must have that $\overline{\nu}(l) \trianglerighteq \overline{\nu}(p)$, and if the inequality is strict, then $Y_{l,p}$ must be true. This holds if for all dependency pairs $l \Rrightarrow p$ with $l = f(l_1, \ldots, l_m) \cdot l_{m+1} \cdots l_n$ and $p = g(p_1, \ldots, p_l) \cdot p_{k+1} \cdots p_k$, and for all i, j with $1 \leq i \leq n$ and $1 \leq j \leq k$, the following clauses are satisfied:

$$\begin{array}{rll} \neg X_{f,i} \vee \neg X_{g,j} \vee Y_{l,p} & \text{if} & l_i \rhd p_j \\ \neg X_{f,i} \vee \neg X_{g,j} \vee \neg Y_{l,p} & \text{if} & l_i = p_j \\ \neg X_{f,i} \vee \neg X_{g,j} & \text{if} & l_i \not\trianglerighteq p_j \end{array}$$

Finally, at least one dependency pair must be strictly oriented. This is the case if the following formula is satisfied:

$$Y_{l_1,p_1} \vee \ldots \vee Y_{l_n,p_n} \quad \text{if } \mathcal{P} = \{l_1 \Rrightarrow p_1, \ldots, l_n \Rrightarrow p_n\}$$

The conjunction of these clauses is passed to a SAT-solver. If the problem is satisfiable, then the assignment of the $Y_{l,p}$ tells us which dependency pairs to remove.

8.5 Automated Polynomial Interpretations

Now let us move on to WANDA's implementation of polynomial interpretations. As with the subterm criterion, WANDA uses a SAT-solver to decide on the actual interpretation – but there are quite a few steps before that point.

WANDA's implementation of polynomial interpretations is somewhat minimal: it is limited essentially to second-order AFSMs (the actual restriction is a bit stronger, but in practice coincides with second-order), and only tries quite simple interpretation shapes. There is no use of input-dependent heuristics, and some choices are less than optimally efficient. The restriction to second-order AFSMs does not seem to come at a high price; it is a rather common class, including 151 out of 156 benchmarks in the current version of the termination problem database (TPDB), 8.0.1. Different interpretation shapes (in particular non-polynomial interpretations using for instance the $\max$ function) might have a larger influence, however. This is left for future work.

Although given input in applicative format, and with possibly more than one base type, the module immediately calculates a maximum arity and considers all base types as the same type. This is allowed by the transformations from Theorem 2.26 and 2.29.

As we have seen, the module for polynomial interpretations may be called in three different ways, given by a parameter either in the rule removal or dependency pair algorithm:

STRONGLY MONOTONIC Called for rule removal. The module must give a *strong* reduction pair which, by Theorem 4.16, is the case if for all function symbols $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau$ the interpretation of f is strongly monotonic in its first n arguments.

WEAKLY MONOTONIC Called for a non-collapsing set of dependency pairs in the dependency pair framework. The module must give a *weak* reduction pair, and has no further requirements.

SEMI-WEAKLY MONOTONIC Called for a collapsing set of dependency pairs in the dependency pair framework. The module must give a *weak* reduction pair, which moreover satisfies the default requirements from Definition 6.72 with regards to $\mathcal{J}(0_\sigma)$ and $\mathcal{J}(0_\sigma)$, and which has the property that $\mathcal{J}(f) \sqsupseteq \boldsymbol{\lambda} x_1 \ldots x_n \vec{y}.x_i(\vec{0})$ for all $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in S$ and $i \leq n$, where S is some given set.

Moreover, the module is presented with a number of constraints of the form $l \mathrel{(\gtrsim)} r$ and $l \gtrsim r$, and charged to find an interpretation such that $[\![l]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![r]\!]_{\mathcal{J},\alpha}$ for all constraints $l \gtrsim r$ and $[\![l]\!]_{\mathcal{J},\alpha} \sqsupseteq [\![r]\!]_{\mathcal{J},\alpha}$ or $[\![l]\!]_{\mathcal{J},\alpha} \sqsupset [\![r]\!]_{\mathcal{J},\alpha}$ for all constraints $l \mathrel{(\gtrsim)} r$. At least one of the $\mathrel{(\gtrsim)}$ constraints must be oriented strictly.

Overview. We will seek a polynomial interpretation in the natural numbers, following the definitions of Chapter 4.3. To find these interpretations automatically, WANDA uses the following steps:

1. make default choices for $\mathcal{J}(@^\sigma)$ for all σ, and for all $\mathcal{J}(\mathsf{c}_i^\sigma)$, depending on the monotonicity parameters passed to the module;
2. assign every function symbol a higher-order polynomial with *parameters* as coefficients; we only consider polynomials of certain simple forms;
3. for all requirements $l \mathrel{(\gtrsim)} r$ and $l \gtrsim r$, calculate $[\![l]\!]_{\mathcal{J},\alpha}$ and $[\![r]\!]_{\mathcal{J},\alpha}$ as a function on parameters and variables – this gives constraints $P_i \mathrel{(\geq)} Q_i$ and $P_i \geq Q_i$;
4. introduce a parameter o_i for all requirements of the form $P_i \mathrel{(\geq)} Q_i$, and replace these requirements by $P_i \geq Q_i + o_i$ – if we also introduce the constraint $o_1 + \ldots + o_n \geq 1$ then, when all constraints are satisfied, at least one of the interpretations is strictly oriented;
5. simplify the constraints until they no longer contain variables;
6. impose maximum values on the search space of the parameters (otherwise the problem whether $P \geq Q$ is undecidable), and encode the numeric requirements as a formula in proposition logic; send this problem to an external SAT-solver to find a suitable value for the parameters.

These steps are detailed in the sections below, using the AFSM `map` as a running example.

In the algorithm, we assume that the AFSM under consideration has the following property: for all function symbols $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \sigma_{n+1} \to \ldots \to \sigma_m \longrightarrow \iota$ occurring in any constraint, each σ_i has order 1 or 2. This restriction posed on a set of rules implies that the system is second-order, but also excludes for instance rules where either side has a sub-meta-term such as $f(\lambda x.(x \cdot Z))$.

8.5.1 Default Choices

Let us start by fixing the interpretations of all $@^\sigma$ and c_i^σ. The c_i^σ occur only in the right-hand sides of constraints (if they occur at all), and application occurs far more on the right-hand side of a constraint than on the left (especially in the AFS formalism that is used in the termination problem database). Therefore it makes sense to choose both values as small as possible.

- for all types σ and numbers i, let $\mathcal{J}(\mathsf{c}_i^\sigma) = 0_\sigma$;
- when given a parameter `STRONGLY MONOTONIC`, let $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn\vec{m}.f(n, \vec{m}) + n(\vec{0})$;
- when given a parameter `SEMI-WEAKLY MONOTONIC`, let $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn\vec{m}.\max(f(n, \vec{m}), n(\vec{0}))$;
- when given a parameter `WEAKLY MONOTONIC`, let $\mathcal{J}(@^\sigma) = \boldsymbol{\lambda} fn\vec{m}.f(n, \vec{m})$;

Note that, if $s_1, \ldots, s_n$ have base type, then with these choices, $[\![t \cdot \vec{s}]\!]$ is either $[\![t]\!]([\![s_1]\!], \ldots, [\![s_n]\!]) + [\![s_1]\!] + \ldots + [\![s_n]\!]$ in the `STRONGLY MONOTONIC` case, or $\max([\![t]\!]([\![s_1]\!], \ldots, [\![s_n]\!]), [\![s_1]\!], \ldots, [\![s_n]\!])$ in the `SEMI-WEAKLY MONOTONIC` case, or $[\![t]\!]([\![s_1]\!], \ldots, [\![s_n]\!])$ in the `WEAKLY MONOTONIC` case. Having fixed these choices, we have that $\mathcal{J}(@^\sigma) \sqsupseteq \boldsymbol{\lambda} fn.f(n)$ as required by Theorem 4.10, and the default requirements are satisfied in the `SEMI-WEAKLY MONOTONIC` case.

To satisfy all requirements imposed by the parameter for monotonicity, we additionally have either a strong monotonicity requirement (in the case `STRONGLY MONOTONIC`) or a requirement on the elements of S left. In the first case, assign $S := \mathcal{F}$ (and in the `WEAKLY MONOTONIC` case, let $S := \emptyset$). To each element f of S we will assign a higher-order polynomial of the form $\boldsymbol{\lambda}\vec{x}\vec{y}.a_1 \cdot x_1(\vec{b}(\vec{x}, \vec{y})) + \ldots + a_n \cdot x_n(\vec{b}(\vec{x}, \vec{y})) + P(\vec{x}, \vec{y})$ with all $a_i > 0$, where n is the arity of f. With this choice, we both have that $\mathcal{J}(f)$ is strongly monotonic in its first n arguments (by Theorem 4.20), and that $\mathcal{J}(f) \sqsupseteq \boldsymbol{\lambda}\vec{x}\vec{y}.x_i(\vec{0})$ for each $i \leq n$.

Thus, we can furthermore ignore the monotonicity parameter, and focus only on orienting the constraints (as long as we respect the restriction on the interpretation of elements of S).

8.5.2 Choosing Parametric Polynomial Interpretations

The polynomial class in WANDA uses numeric constants, *parameters*, which can be thought of as *unknown* constants, addition, multiplication, base-type variables and function application. Every function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \sigma_{n+1} \rightarrow \ldots \rightarrow \sigma_m \rightarrow \mathtt{o} \in \mathcal{F}$ is assigned a function of the form $\boldsymbol{\lambda} x_1 \ldots x_m . p_1 + p_2 + a$, where:

- a is a parameter;
- p_1 has the form $a_1 \cdot x_1(\vec{0}) + \ldots + a_m \cdot x_m(\vec{0})$, where the a_i are parameters (this is well-typed because of the restriction on the function symbols, where argument positions have a type of the form $\mathtt{o} \rightarrow \ldots \rightarrow \mathtt{o} \rightarrow \mathtt{o}$);
  - if $f \in S$ we pose the following constraints: $a_1 \geq 1, \ldots, a_n \geq 1$
- $p_2 = q_1 + \ldots + q_k$, where each q_j has the form $b_j \cdot x_{i_1} \cdots x_{i_k} \cdot x_j(x_{i_1}, \ldots, x_{i_k}) + c_j \cdot x_j(x_{i_1}, \ldots, x_{i_k})$, with b_j and c_j parameters, the x_{i_l} first-order variables and x_j a higher-order variable; every combination of a higher-order variable with first-order variables occurs.[3]

During the algorithm we keep a running list of constraints of the form $P \geq Q$ which must be satisfied to find an interpretation function. If the requirements on the a_i are satisfied, then the interpretations for the elements of S have the required properties.

Running Example. To demonstrate the technique, consider rule removal on the common `map` example.

$$\begin{array}{lcl} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) & \underset{(\sim)}{\succ} & \mathtt{nil} \\ \mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y)) & \underset{(\sim)}{\succ} & \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y)) \end{array}$$

We assign:

$$\begin{array}{rcl} \mathcal{J}(\mathtt{nil}) & = & a_1 \\ \mathcal{J}(\mathtt{cons}) & = & \boldsymbol{\lambda} nm.a_2 \cdot n + a_3 \cdot m + a_4 \\ \mathcal{J}(\mathtt{map}) & = & \boldsymbol{\lambda} fn.a_5 \cdot f(0) + a_6 \cdot n + a_7 \cdot n \cdot f(n) + a_8 \cdot f(n) + a_9 \end{array}$$

We save the inequalities $a_2 \geq 1, a_3 \geq 1, a_5 \geq 1, a_6 \geq 1$ as constraints.

[3] In case the constraint solver does not find a solution for this interpretation shape, WANDA additionally includes non-linear monomials $d_{i,j} \cdot x_i \cdot x_j$ (where $i < j$) without functional variables in the parametric higher-order polynomials and tries again. In general, here one can use arbitrary parametric polynomials, but the more complex polynomials are used, the longer it usually takes to determine whether there is a solution.

8.5.3 Calculating Constraints

From these parametric higher-order polynomials, we can calculate the interpretations of meta-terms and simplify the resulting polynomials into a sum of monomials. For the requirements $l \mathrel{(\succsim)} r$ we use constraints $[\![l]\!]_{\mathcal{J},\alpha} \geq [\![r]\!]_{\mathcal{J},\alpha} + o$ for some fresh *bit* o: a parameter whose value ranges over $\{0, 1\}$. To make sure that at least one constraint of this form is oriented strictly, we require that the sum of these new bits is positive.

Although all function symbols are interpreted with a polynomial, the application symbol is not, in the case `SEMI-WEAKLY MONOTONIC`. Thus, the result of an interpretation might just be a function including the $\max$ symbol. In the right-hand side of a constraint, this is not going to give much problems (as $P \geq \max(Q_1, Q_2)$ if and only if $P \geq Q_1$ and $P \geq Q_2$), but in the left-hand side it is harder to deal with. Although this can be dealt with neatly (as for instance done in [38]), the resulting complexity is much higher than is justified by the gain from using $\max$ instead of $+$ for interpreting application.

However, due to the pattern restriction, as well as the restriction on the type declarations of function symbols, we know that application is a rare occurrence in the left-hand sides of constraints: they cannot have a subterm $Z(\vec{x}) \cdot s$ or a subterm $x \cdot s$. The only way application can occur in a pattern is in the form $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$. But, this is a rare occurrence: in most common examples, function symbols have a base type as output type.

Let us make one alteration, then. In all left-hand sides of constraints, replace occurrences of $\max(a, b_1, \ldots, b_n)$ simply by a. This is sound because $l \geq r$ certainly holds if $l \geq l'$ and $l' \geq r$. In practice, this means that in the left-hand side of rules, $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ is interpreted by $\mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![s_m]\!]_{\mathcal{J},\alpha})$, while in the right-hand side it is interpreted by $\max(\mathcal{J}(f)([\![s_1]\!]_{\mathcal{J},\alpha}, \ldots, [\![s_m]\!]_{\mathcal{J},\alpha})$, $[\![s_{n+1}]\!]_{\mathcal{J},\alpha}, \ldots, [\![s_m]\!]_{\mathcal{J},\alpha})$. Depending on how we choose $\mathcal{J}(f)$, these values may well be equal.

Running Example. In the `map` example, the list of integer constraints is updated as follows:

$$\begin{array}{rcl} a_2 & \geq & 1 \\ a_3 & \geq & 1 \\ a_5 & \geq & 1 \\ a_6 & \geq & 1 \\ o_1 + o_2 & \geq & 1 \end{array}$$

$$\begin{array}{rcl} a_1 \cdot a_6 + a_9 + \\ a_1 \cdot a_7 \cdot f(a_1) + a_5 \cdot f(0) + a_8 \cdot f(a_1) & \geq & a_1 + o_1 \end{array}$$

$$\begin{array}{rcl} a_4 \cdot a_6 + a_9 + & & a_3 \cdot a_9 + a_4 + o_2 + \\ a_2 \cdot a_6 \cdot n + a_3 \cdot a_6 \cdot m + a_5 \cdot f(0) + & & a_3 \cdot a_6 \cdot m + \\ a_2 \cdot a_7 \cdot n \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) + & \geq & a_2 \cdot f(n) + a_3 \cdot a_5 \cdot f(0) + \\ a_3 \cdot a_7 \cdot m \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) + & & a_3 \cdot a_7 \cdot m \cdot f(m) + \\ a_4 \cdot a_7 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) + & & a_3 \cdot a_8 \cdot f(m) \\ a_8 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) & & \end{array}$$

8.5.4 Simplifying Polynomial Constraints

The constraints we thus obtain contain variables as well as parameters; these constraints should be read as "there exist a_i, o_k such that for all n, m, F, this inequality holds". To avoid dealing with claims over all possible numbers or functions, we simplify the requirements until they do not contain variables any more. To a large extent, these simplifications correspond to the ones used with automations of polynomial interpretations for first-order rewriting, but function application presents an extra difficulty. To deal with function application of the higher-order variables, we will use the following lemma:

Lemma 8.4. *Let F be a weakly monotonic functional and let p, q, p_i, q_i, s_i, r_i all be polynomials. Then:*

1. $F(r_1, \ldots, r_k) \cdot p \geq F(s_1, \ldots, s_k) \cdot q$ *if* $r_1 \geq s_1, \ldots, r_k \geq s_k, p \geq q$.
2. $r_1 \cdot p_1 + \ldots + r_n \cdot p_n \geq s_1 \cdot q_1$ *if there are bits* $o_1, \ldots, o_n$ *such that* $r_1 \cdot o_1 + \ldots + r_n \cdot o_n \geq s_1$ *and for each* i*: either* $o_i = 0$ *or* $p_i \geq q_1$.
3. $r_1 \cdot p_1 + \ldots + r_n \cdot p_n \geq s_1 \cdot q_1 + \ldots + s_m \cdot q_m$ *if there are* $e_{i,j}$ *for* $1 \leq i \leq n, 1 \leq j \leq m$ *with:*
   a) *for all* i: $r_i \geq e_{i,1} + \ldots + e_{i,m}$;
   b) *for all* j: $e_{1,j} + \ldots + e_{n,j} \geq s_j$;
   c) *either* $e_{i,j} = 0$ *or* $p_i \geq q_j$.

Proof. (1) holds by weak monotonicity of F.

For (2), let $i_1, \ldots, i_l$ be the indexes j where o_j is 1. Then $r_{i_1} + \ldots + r_{i_l} \geq s_1$, and each $p_{i_j} \geq q_1$. But then:

$$\begin{array}{rcl} r_1 \cdot p_1 + \ldots + r_n \cdot p_n & = & r_{i_1} \cdot p_1 + \ldots + r_{i_l} \cdot p_{i_l} \\ & \geq & r_{i_1} \cdot q_1 + \ldots + r_{i_l} \cdot q_1 \\ & = & (r_{i_1} + \ldots + r_{i_l}) \cdot q_1 \\ & \geq & s_1 \cdot q_1 \end{array}$$

As for (3),

$$\begin{aligned} r_1 \cdot p_1 + \ldots + r_n \cdot p_n &\geq \sum_{i=1}^{n} \sum_{j=1}^{m} e_{i,j} \cdot p_i \\ &\geq \sum_{i=1}^{n} \sum_{j=1}^{m} e_{i,j} \cdot q_j \text{ (as } e_{i,j} = 0 \text{ if not } p_i \geq q_j) \\ &= \sum_{j=1}^{m} \sum_{i=1}^{n} e_{i,j} \cdot q_j \\ &= \sum_{j=1}^{m} (e_{1,j} + \ldots + e_{n,j}) \cdot q_j \\ &\geq \sum_{j=1}^{m} s_j \cdot q_j \\ &= s_1 \cdot q_1 + \ldots + s_m \cdot q_m \end{aligned}$$

□

Lemma 8.4, together with a number of observations used in first-order polynomial interpretations, supplies the theory we need to simplify the constraints to ones which do not contain any variables. WANDA will try to simplify a polynomial by the first rule of this list which makes a difference. In the following rules for simplifying, a "component" of a monomial $a_1 \cdots a_n$ is any of the a_i (but p is not a component of the monomial $F(p)$).

1. Do standard simplifications on the monomials in the constraints, for instance replacing $p + B \cdot p$ by $(B + 1) \cdot p$ if B is a *known* constant, and removing monomials $0 \cdot p$.

2. If the same monomial occurs on either side, possibly multiplied by a known constant, remove it on both sides.
(*Example:* $3 \cdot F(n) \geq F(n) + a_1 \cdot n$ is replaced by $2 \cdot F(n) \geq a_1 \cdot n$.)
This is valid because $p \geq q$ is true if $p - x \geq q - x$.

3. Remove constraints $p \geq 0$ and $p \geq p$.
These always hold, so no need to keep track!

4. Split constraints $0 \geq p_1 + \ldots + p_n$ into the n constraints $0 \geq p_i$. Remove constraints $0 \geq a$ where a is a single parameter, but replace a by 0 in all other constraints.
This is valid because $0 \geq a$ implies $a = 0$, and $0 + \ldots + 0 = 0$.

5. Replace constraints $P \geq Q[\max(r, s)]$ by the two constraints $P \geq Q[r]$ and $P \geq Q[s]$.
This is valid because for any valuation $Q[\max(r, s)]$ equals $Q[r]$ or $Q[s]$.

6. Given a constraint $p_1 + \ldots + p_n \geq p_{n+1} \ldots p_m$ where some, but not all, of the monomials p_i contain a component x or $x(\vec{p})$ for some fixed variable

x, let A contain the indexes i of those monomials p_i which have x or $x(\vec{q})$. Replace the constraint by the two constraints

$$\sum_{i\in A, i\leq n} p_i \geq \sum_{i\in A, i>n} p_i$$

and

$$\sum_{i\notin A, i\leq n} p_i \geq \sum_{i\notin A, i>n} p_i.$$

(*Example:* splitting on n, the constraint $3 \cdot n \cdot m + a_2 \cdot F(a_3 \cdot n + a_4) \geq 2+a_7 \cdot m+F(n)$ is split into $3 \cdot n \cdot m \geq 0$ and $a_2 \cdot F(a_3 \cdot n+a_4) \geq 2+a_7 \cdot m+F(n)$; subsequently, splitting on F, the latter is split into $a_2 \cdot F(a_3 \cdot n+a_4) \geq F(n)$ and $0 \geq 2 + a_7 \cdot m$.)
This is valid because $p_1 + p_2 \geq q_1 + q_2$ holds if $p_1 \geq q_1$ and $p_2 \geq q_2$.

7. If all non-zero monomials on either side of a constraint have a component x, "divide out" x. Here, x is a first-order variable.
(*Example:* the constraint $a_1 \cdot n + n \cdot n \cdot f(a_3, n) \geq n + a_3 \cdot n$ is replaced by $a_1 + n \cdot f(a_3, n) \geq 1 + a_3$, and the monomial $0 \geq a_5 \cdot m$ is replaced by $0 \geq a_5$.)
This is valid because $p \cdot n \geq q \cdot n$ certainly holds for all n if $p \geq q$ (cf. the absolute positiveness criterion [57]).

8. Replace a constraint
$$s \cdot x_1(p_{1,1}, \ldots, p_{1,k_1}) \cdots x_n(p_{n,1}, \ldots, p_{n,k_n}) \geq$$
$$s \cdot x_1(q_{1,1}, \ldots, q_{1,k_1}) \cdots x_n(q_{n,1}, \ldots, q_{n,k_n})$$
by the constraints $s \cdot p_{i,j} \geq s \cdot q_{i,j}$ for all i, j. Do the same for constraints $x_1(\vec{p}_1) \cdots x_n(\vec{p}_n) \geq x_1(\vec{q}_1) \cdots x_n(\vec{q}_n)$ (where essentially $s = 1$).
This is valid by Lemma 8.4(1) and case analysis whether $s = 0$ or not.

9. Let $p_1, \ldots, p_n, q_1, \ldots, q_m$ be monomials of the form $x_1(\vec{r}_1), \ldots, x_k(\vec{r}_k)$, for fixed $x_1, \ldots, x_k$.

   a) Replace a constraint $r_1 \cdot p_1 \geq s_1 \cdot q_1$ by the two constraints $r_1 \geq s_1$ and $p_1 \geq q_1$ (unless $r_1 = s_1$, in which case clause 8 should be used).
   This is valid by Lemma 8.4(1).

   b) Replace a constraint $r_1 \cdot p_1 + \ldots + r_n \cdot p_n \geq s \cdot q_1$ with $n \geq 2$ by the $n + 1$ constraints:
   $a_1 \cdot o_1 + \ldots + a_n \cdot o_n \geq s$; (for fresh bits $o_1, \ldots, o_n$)
   for $1 \leq i \leq n$: $o_i \cdot p_i \geq o_i \cdot q_1$.
   This is valid by Lemma 8.4(2).

   c) Replace a constraint $r_1 \cdot p_1 + \ldots + r_n \cdot p_n \geq s_1 \cdot q_1 + \ldots + s_m \cdot q_m$ with $m \geq 2$ by the following constraints, where the $e_{i,j}$ are fresh parameters:
   for $1 \leq i \leq n$: $r_i \geq e_{i,1} + \ldots + e_{i,m}$
   for $1 \leq j \leq m$: $e_{1,j} + \ldots + e_{n,j} \geq s_j$
   for $1 \leq i \leq n,\ 1 \leq j \leq m$: $e_{i,j} \cdot p_i \geq e_{i,j} \cdot q_j$
   This is valid by Lemma 8.4(3).

   In all cases, simplify the resulting constraints with clause 8 where possible.

It is not hard to see that while a constraint still has variables in it, we can apply clauses to simplify or split it (taking into account that $\max$ does not appear in the left-hand side of constraints), and that the clauses also terminate on a set of constraints without variables.

These simplifications are all *sound*, but not necessarily *complete*: we might for instance split a universally valid constraint $n \cdot F(n) \geq n \cdot F(1)$ into constraints $n \geq n$ (which holds) and $F(n) \geq F(1)$ (which does not).

Running Example. Let us simplify the constraints from our running example with clause 6, grouping monomials together depending on which variables occur in them. We obtain:

$$\begin{array}{rcl}
a_2, a_3, a_5, a_6 & \geq & 1 \\
o_1 + o_2 & \geq & 1 \\
a_1 \cdot a_6 + a_9 & \geq & a_1 + o_1 \\
a_1 \cdot a_7 \cdot f(a_1) + a_5 \cdot f(0) + a_8 \cdot f(a_1) & \geq & 0 \\
a_4 \cdot a_6 + a_9 & \geq & a_3 \cdot a_9 + a_4 + o_2 \\
a_2 \cdot a_6 \cdot n & \geq & 0 \\
a_3 \cdot a_6 \cdot m & \geq & a_3 \cdot a_6 \cdot m \\
a_2 \cdot a_7 \cdot n \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) & \geq & 0 \\
a_3 \cdot a_7 \cdot m \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) & \geq & a_3 \cdot a_7 \cdot m \cdot f(m) \\
\begin{array}{r} a_5 \cdot f(0) + \\ a_4 \cdot a_7 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) + \\ a_8 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) \end{array} & \geq & \begin{array}{l} a_2 \cdot f(n) + a_3 \cdot a_5 \cdot f(0) + \\ a_3 \cdot a_8 \cdot f(m) \end{array}
\end{array}$$

The constraints $P \geq 0$ and $a_3 \cdot a_6 \cdot m \geq a_3 \cdot a_6 \cdot m$ are trivial and can be removed with clause 3. If we additionally divide away the non-functional variables (clause 7) we have the following constraints left:

$$\begin{array}{rcl}
a_2, a_3, a_5, a_6 & \geq & 1 \\
o_1 + o_2 & \geq & 1 \\
a_1 \cdot a_6 + a_9 & \geq & a_1 + o_1 \\
a_4 \cdot a_6 + a_9 & \geq & a_3 \cdot a_9 + a_4 + o_2 \\
a_3 \cdot a_7 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) & \geq & a_3 \cdot a_7 \cdot f(m) \\
\begin{array}{r} a_5 \cdot f(0) + \\ a_4 \cdot a_7 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) + \\ a_8 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) \end{array} & \geq & \begin{array}{l} a_2 \cdot f(n) + a_3 \cdot a_5 \cdot f(0) + \\ a_3 \cdot a_8 \cdot f(m) \end{array}
\end{array}$$

The first five of these are completely simplified – only the latter two remain for simplification.

The constraint $a_3 \cdot a_7 \cdot f(a_2 \cdot n + a_3 \cdot m + a_4) \geq a_3 \cdot a_7 \cdot f(m)$ is handled with clause 8, and replaced by the constraint $a_3 \cdot a_7 \cdot (a_2 \cdot n + a_3 \cdot m + a_4) \geq a_3 \cdot a_7 \cdot m$. Using clauses 6, 7 and 3 this reduces to $a_3 \cdot a_7 \cdot a_3 \geq a_3 \cdot a_7$.

For the latter constraint we use clause 9(3), and obtain:

$$\begin{array}{rcl} a_5 & \geq & e_{1,1} + e_{1,2} + e_{1,3} \\ a_4 \cdot a_7 & \geq & e_{2,1} + e_{2,2} + e_{2,3} \\ a_8 & \geq & e_{3,1} + e_{3,2} + e_{3,3} \\ e_{1,1} \cdot 0 & \geq & e_{1,1} \cdot n \\ e_{1,2} \cdot 0 & \geq & e_{1,2} \cdot 0 \\ e_{1,3} \cdot 0 & \geq & e_{1,3} \cdot m \end{array} \qquad \begin{array}{rcl} e_{1,1} + e_{2,1} + e_{3,1} & \geq & a_2 \\ e_{1,2} + e_{2,2} + e_{3,2} & \geq & a_3 \cdot a_5 \\ e_{1,3} + e_{2,3} + e_{3,3} & \geq & a_3 \cdot a_8 \\ e_{2,1} \cdot (a_2 \cdot n + a_3 \cdot m + a_4) & \geq & e_{2,1} \cdot n \\ e_{2,2} \cdot (a_2 \cdot n + a_3 \cdot m + a_4) & \geq & e_{2,2} \cdot 0 \\ e_{2,3} \cdot (a_2 \cdot n + a_3 \cdot m + a_4) & \geq & e_{2,3} \cdot m \\ e_{3,1} \cdot (a_2 \cdot n + a_3 \cdot m + a_4) & \geq & e_{3,1} \cdot n \\ e_{3,2} \cdot (a_2 \cdot n + a_3 \cdot m + a_4) & \geq & e_{3,2} \cdot 0 \\ e_{3,3} \cdot (a_2 \cdot n + a_3 \cdot m + a_4) & \geq & e_{3,3} \cdot m \end{array}$$

The constraints $e_{1,1} \cdot 0 \geq e_{1,1} \cdot n$ and $e_{1,3} \cdot 0 \geq e_{1,3} \cdot m$ are simplified using clauses 6, 1 and 7, to $0 \geq e_{1,1}$ and $0 \geq e_{1,3}$, and thus we can replace these parameters by 0 everywhere. The other clauses which contain n and m can also be simplified using clauses 1, 6, 1, 7 and 3. In the end, putting all constraints together again, we must satisfy:

$$\begin{array}{rcl} a_5 & \geq & e_{1,2} \\ a_4 \cdot a_7 & \geq & e_{2,1} + e_{2,2} + e_{2,3} \\ a_8 & \geq & e_{3,1} + e_{3,2} + e_{3,3} \\ e_{2,1} \cdot a_2 & \geq & e_{2,1} \\ e_{3,1} \cdot a_2 & \geq & e_{3,1} \\ a_2, a_3, a_5, a_6 & \geq & 1 \\ o_1 + o_2 & \geq & 1 \\ a_4 \cdot a_6 + a_9 & \geq & a_3 \cdot a_9 + a_4 + o_2 \end{array} \qquad \begin{array}{rcl} e_{2,1} + e_{3,1} & \geq & a_2 \\ e_{1,2} + e_{2,2} + e_{3,2} & \geq & a_3 \cdot a_5 \\ e_{2,3} + e_{3,3} & \geq & a_3 \cdot a_8 \\ e_{2,3} \cdot a_3 & \geq & e_{2,3} \\ e_{3,3} \cdot a_3 & \geq & e_{3,3} \\ a_1 \cdot a_6 + a_9 & \geq & a_1 + o_1 \\ a_3 \cdot a_7 \cdot a_3 & \geq & a_3 \cdot a_7 \end{array}$$

Thus, using a handful of clauses, the requirements are simplified to a number of constraints with unknowns over the natural numbers. In the actual WANDA implementation a few more small optimisations appear. For example, some of the simplifications are combined, and a "domain" is saved for every unknown, which is dynamically updated. However, these optimisations make no fundamental difference to the technique.

8.5.5 Solving Variable-Free Constraints

The final step is to find values for the parameters which satisfy the constraints. Since this is in general an undecidable problem, we pose a bound on the values, choosing that every a_i must be in $\{0, \ldots, \mathrm{MAX}\}$. In WANDA, MAX is set to 3; typically, constants in polynomial interpretations do not need to be large.

Even with bounds on the values of the a_i, finding a solution is still an NP-complete problem. But, we can now encode it as a satisfiability problem and feed it to an external SAT-solver. This is done by representing every parameter as a sequence of bits, and encoding addition and multiplication with logical connectives; the implementation is similar to the one in [37]. For example, the

requirement $a \geq b$ is encoded as:

$$(a_1 \wedge \neg b_1) \vee (a_1 \equiv b_1 \wedge (a_0 \vee \neg b_0))$$

Here, a_0 is the least relevant bit of a. This leads to a formula which can be transformed into CNF using Tseitin's transformation [122].

As an alternative, we might have used SMT-based techniques [23]. Here we used a SAT-based modelling because other modules of WANDA also use SAT.

8.5.6 Final Remarks

The implementation of polynomial interpretations has been kept as simple as possible, in order to be easy to implement, understand, and adapt. This is visible in the limitation to, essentially, second-order systems, and the explicit removal of $\max$ in the left-hand side of an ordering constraint. Slightly more hidden, the simplicity is present in the choice to use only constraints of the form $P \geq Q$, combined with the Boolean connective $\wedge$. This leads for instance to phrasing "$a = 0$ or $P \geq Q$" as "$a \cdot P \geq a \cdot Q$".

There are many directions for future improvement. We might consider a greater variation of polynomials. At the moment we do not, for instance, try monomials of the form $F(F(n))$ for F and n variables. However, such an extension would likely have to be combined with some heuristics of which interpretations are likely to lead to good results, because increasing the number of functional variables in the interpretation of a function symbol has an exponential effect on the number of constraints we obtain from simplifying. Exponential blowup in the constraints during simplification is already present: consider how many constraints are added by clause 9 for a polynomial of the form $p_1 \cdot F(p_2) + p_3 \cdot F(p_4) \geq q_1 \cdot F(q_2) + q_3 \cdot F(q_4)$). Thus, any additional interpretation shapes have to be considered very well!

8.6 Automated Path Orderings

Next, let us consider WANDA's implementation of path orderings. We will use the recursive definition of $\mathrm{StarHorpo}$, and translate the ordering constraints into a satisfiability problem, which is sent to a SAT-solver. The path ordering module may be called in three different ways, given by a parameter either in the rule removal or dependency pair algorithm:

PRESERVING Called for rule removal. The module must give a *strong* reduction pair which, by Theorem 5.39, is the case if an argument preserving function is used.

NON-PRESERVING Called for a non-collapsing set of dependency pairs in the dependency pair framework. The module must give a *weak* reduction pair, and has no further requirements.

PARTLY PRESERVING Called for a collapsing set of dependency pairs in the dependency pair framework. The module must give a *weak* reduction pair, which moreover satisfies the standard properties from Chapter 6.6.2 with regards to π and $\blacktriangleright$. Moreover, for all symbols in the given set S, we must have $f(\vec{s}) \cdot \vec{t} \gtrsim s_i \cdot \vec{c}$ if both sides have base type.

Moreover, the module is presented with a number of constraints of the form $l \ {}_{(}\gtrsim_{)} r$ and $l \gtrsim r$, and charged to find a precedence, status function and argument function, which together generate a reduction pair. At least one of the ${}_{(}\gtrsim_{)}$ constraints should be oriented with $\succ$, and all other constraints with $\gtrsim$.

Overview. We will keep track of two lists of constraints: the main list consists of formulas of proposition logic, all of which must be satisfied; the second list has pairs of the form $\langle l \succeq_\star r, \texttt{TODO}\rangle$ or $\langle l \succ_\star r, \texttt{DONE}\rangle$. The algorithm iteratively ticks off items in the second list, and replaces them by propositional formulas in the first. To this end, we take the following steps:

1. initialise the lists: for all constraints of the form $l \gtrsim r$ store $\langle l \gtrsim r, \texttt{TODO}\rangle$ in the list of ordering constraints, for all constraints of the form $l \ {}_{(}\gtrsim_{)} r$ store both $\langle l \gtrsim r, \texttt{TODO}\rangle$ and $\langle l \succ r, \texttt{TODO}\rangle$ in this list; in the list of propositional formulas, store $\texttt{GE}_{l,r}$ for all constraints, and also $\texttt{GR}_{l_1,r_1} \vee \ldots \vee \texttt{GR}_{l_n,r_n}$ for all ${}_{(}\gtrsim_{)}$ constraints;
2. assign standard shapes for the argument functions, and encode both the possible shapes, the status function and the precedence with propositional formulas;
3. simplify the $\mathrm{StarHorpo}$ constraints to smaller constraints until all items of the second list are marked as done;
4. send the conjunction of all formulas in the list to an external SAT-solver; if the solver finds a solution, then this shows that the requirements can all be oriented, and the values of the $\texttt{GR}_{l_i,r_i}$ indicate which constraints are oriented strictly.

8.6.1 Initialisation

The idea of the algorithm is as follows: in every step, if all propositional formulas in the main list are satisfied, *and* if for all pairs $\langle l \succeq_\star r, \texttt{TODO}\rangle$ in the second list we have: "if $\texttt{GE}_{l,r}$ is satisfied, then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ holds", *and* something similar holds for the other constraints $\langle p, \texttt{TODO}\rangle$ in the second list... *then* the constraints are satisfied and we have found a suitable reduction pair.

Thus, to start off, the very first step in the module is to translate the given set of constraints to application-free meta-terms (using the function μ from Definition 5.36). Having done this, for all constraints $l \gtrsim r$ and $l \ {}_{(}\gtrsim_{)} r$, introduce a variable $\texttt{GE}_{l,r}$, and store it in the list of propositional formulas. Store $\langle l \gtrsim r, \texttt{TODO}\rangle$

in the second list, which we may refer to as *todo list*. This action corresponds with posing the requirement that $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ for all constraints $l \succsim r$ and $l \mathrel{(\succsim)} r$. Since $\succ_\star$ is a subrelation of $\succeq_\star$ posing this requirement does not lose generality. We promise to add formulas which guarantee that if $\mathtt{GE}_{l,r}$ is true, then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$.

In addition, to guarantee that at least one of the $\mathrel{(\succsim)}$ constraints is oriented strictly, let $l_1 \mathrel{(\succsim)} r_1, \ldots, l_n \mathrel{(\succsim)} r_n$ be the list of all these constraints. We introduce a variable $\mathtt{GR}_{l_i,r_i}$ for all of them, and add the following propositional formula to the list of formulas:

$$\mathtt{GR}_{l_1,r_1} \vee \ldots \vee \mathtt{GR}_{l_n,r_n}$$

This, in combination with adding $\langle l_i \succ r_i, \mathtt{TODO}\rangle$ to the todo list guarantees that at least one of these constraints is oriented with $\succ_\star$.

To be able to simplify the formulas, we merely need to know – or rather, express with propositional variables – the argument function, status function and precedence.

8.6.2 Encoding Status, Precedence and Argument Function

To start, let $\mathcal{F}$ be the set of function symbols which occur in the (now application-free) constraints in the todo list. We must find an argument function mapping $\mathcal{F}$ to terms over some Σ, and a precedence and status function on Σ.

Argument Function. The first decision to make is what kind of argument functions we shall use, as this determines what symbols the precedence and the status function have to deal with. We shall limit attention to simple argument functions:

- every function symbol $f : \sigma$ with arity 0 that appears only in the right-hand side of a constraint (including all symbols $\mathtt{c}_i^\sigma$) is automatically mapped to $\bot_\sigma$;
- every other function symbol $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau$ is either mapped to $\lambda x_1 \ldots x_n.x_i$ for some i with $\sigma_i = \tau$, or to $\lambda x_1 \ldots x_n.f'(x_{i_1}, \ldots, x_{i_k})$, where $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ and all i_j are distinct, or to $\lambda x_1 \ldots x_k.\bot_\sigma$;
- if f is $@^{\sigma_1}$, or if $f \in \mathcal{F}$ or the path ordering module is given a parameter `PRESERVING`, then the assignment above must be an argument preserving function, so either $\pi(f) = \lambda x.x$ (which is only allowed if f has arity 1), or $\pi(f) = \lambda x_1 \ldots x_n.f'(x_{i_1}, \ldots, x_{i_k})$ with $\{i_1, \ldots, i_k\} = \{1, \ldots, n\}$, or $\pi(f) = \bot_\sigma$ (which is only allowed if f has arity 0).

We will encode the argument function using the following variables:

- $\mathtt{ArgFiltered}_{f,i}$: indicates that $\pi(f) = \lambda\vec{x}.s$ and $x_i \notin FV(s)$;
- $\mathtt{SymbolFiltered}_f$: indicates that $\pi(f)$ has the form $\lambda\vec{x}.x_i$;

- $\texttt{Permutation}_{f,i,j}$: indicates that $\pi(f) = \lambda\vec{x}.f'(x_{l_1}, \ldots, x_{l_k})$, and $i = l_j$;
- $\texttt{ArgLengthMin}_{f,i}$: indicates that $\pi(f) = \lambda\vec{x}.f'(x_{l_1}, \ldots, x_{l_k})$ and $l_k \geq i$;
- $\texttt{Minimal}_f$: indicates that $\pi(f) = \lambda\vec{x}.\bot_\sigma$.

There is some redundancy in these variables: for instance, if $\texttt{Permutation}_{f,i,j}$ is true, then clearly $\texttt{ArgFiltered}_{f,i}$ must be false, and if $\texttt{SymbolFiltered}_f$ is true, then each $\texttt{ArgLengthMin}_{f,i}$ must be false. This is deliberate: we shall need these variables quite often, and it is better to have a few extra formulas than expressing a commonly used statement as a more advanced formula.

To enforce that the argument function is well-defined, we add the following formulas for all $f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in \mathcal{F}$. First, to make sure that if f is "symbol filtered", exactly one argument is chosen, and it has the right type:

$$\begin{aligned}
\neg\texttt{SymbolFiltered}_f \vee \texttt{ArgFiltered}_{f,i} \vee \texttt{ArgFiltered}_{f,j} &\quad \forall\, 1 \leq i < j \leq n \\
\neg\texttt{SymbolFiltered}_f \vee \bigvee_{i=1}^{n} \neg\texttt{ArgFiltered}_{f,i} & \\
\neg\texttt{SymbolFiltered}_f \vee \texttt{ArgFiltered}_{f,i} &\quad \forall\, i \text{ with } \sigma_i \neq \tau \\
\neg\texttt{SymbolFiltered}_f \vee \neg\texttt{Permutation}_{f,i,j} &\quad \forall\, 1 \leq i, j \leq n \\
\neg\texttt{SymbolFiltered}_f \vee \neg\texttt{Minimal}_f &
\end{aligned}$$

Second, to make sure that mapping symbols to minimal symbols means to remove all their arguments (the values for `Permutation` and `ArgLengthMin` will follow automatically):

$$\neg\texttt{Minimal}_f \vee \texttt{ArgFiltered}_{f,i} \quad \forall\, 1 \leq i \leq n$$

Third, to make sure that `Permutation` respects `ArgFiltered`:

$$\begin{aligned}
\texttt{SymbolFiltered}_f \vee \neg\texttt{ArgFiltered}_{f,i} \vee \neg\texttt{Permutation}_{f,i,j} &\quad \forall\, 1 \leq i, j \leq n \\
\texttt{SymbolFiltered}_f \vee \texttt{ArgFiltered}_{f,i} \vee \bigvee_{j=1}^{n} \texttt{Permutation}_{f,i,j} &\quad \forall\, 1 \leq i \leq n
\end{aligned}$$

Fourth, to guarantee that the permutation π always maps to the numbers $1, \ldots, k$ for some k without gaps, and that the `ArgLengthMin` variables reflect this:

$$\begin{aligned}
\neg\texttt{Permutation}_{f,i,j} \vee \texttt{ArgLengthMin}_{f,j} &\quad \forall 1 \leq i, j \leq n \\
\texttt{Permutation}_{f,1,j} \vee \ldots \vee \texttt{Permutation}_{f,n,j} \vee \neg\texttt{ArgLengthMin}_{f,j} &\quad \forall 1 \leq j \leq n \\
\texttt{ArgLengthMin}_{f,j} \vee \neg\texttt{ArgLengthMin}_{f,j+1} &\quad \forall 1 \leq j < n
\end{aligned}$$

Fifth, to make sure that `Permutation` actually defines a permutation on the non-filtered arguments of f (this is also valid in case $\texttt{SymbolFiltered}_f$ or $\texttt{Minimal}_f$ is true, as the empty permutation is also a permutation):

$$\begin{aligned}
\neg\texttt{Permutation}_{f,i_1,j} \vee \neg\texttt{Permutation}_{f,i_2,j} &\quad \forall 1 \leq i_1 < i_2 \leq n, \forall\, 1 \leq j \leq n \\
\neg\texttt{Permutation}_{f,i,j_1} \vee \neg\texttt{Permutation}_{f,i,j_2} &\quad \forall 1 \leq j_1 < j_2 \leq n, \forall\, 1 \leq i \leq n
\end{aligned}$$

These constraints state that π is injective and a function respectively. By the constraints on `ArgLengthMin` we already know that π maps to the numbers $1, \ldots, k$ for some k without gaps.

Finally, if $\pi(f)$ must be an argument preserving function (either because $f \in S$, because f is an application symbol or because the module must generate a strong reduction pair), then we additionally add formulas:

$$\neg \mathtt{ArgFiltered}_{f,i} \quad \forall\, 1 \leq i \leq n$$

If f has arity 0 and occurs only in the right-hand side of any constraint, then we add the formula:

$$\mathtt{Minimal}_f$$

Precedence. Next, we must define a precedence relation $\trianglerighteq$. This relation must be a quasi-ordering on the function symbols which occur in the constraints after applying $\overline{\pi}$ (except the minimal symbols), and additionally all symbols $@^\sigma$ even if they do not occur. We shall define $\triangleright$ as a relation on $\mathcal{F}$. This induces a well-founded relation on the symbols f' which occur in the $\overline{\pi}$-altered constraints.

Following the approach in [85], let $\mathtt{Prec}_{f,g}$ indicate that $f \trianglerighteq g$. Then all we need to define is reflexivity and transitivity.

$$\begin{array}{rl} \mathtt{Prec}_{f,f} & \forall\, f \in \mathcal{F} \\ \neg\mathtt{Prec}_{f,g} \vee \neg\mathtt{Prec}_{g,h} \vee \mathtt{Prec}_{f,h} & \forall f,g,h \in \mathcal{F} \end{array}$$

If either we have been given a parameter `PRESERVING` or `NON-PRESERVING`, there are no further restrictions on $\trianglerighteq$; only well-foundedness of $\triangleright$. But this is evident, if we assume that for those symbols f which do not occur in $\mathcal{F}$ neither $f \trianglerighteq g$ nor $g \trianglerighteq f$ for any f (after all, $\trianglerighteq$ is not required to be total!). For if $\triangleright$ is not well-founded, say $f_1 \triangleright f_2 \triangleright \ldots$, then all f_i are in $\mathcal{F}$, and since $\mathcal{F}$ is finite there are i, j such that $i < j$ and $f_i = f_j$. But by transitivity $f_i \trianglerighteq f_{j-1} \triangleright f_i$, which is impossible because $\triangleright = \trianglerighteq \setminus \trianglelefteq$.

If, however, we have been given a parameter `PARTLY PRESERVING`, then we must have $@^{\sigma\to\tau} \triangleright @^\sigma, @^\tau$ for all types σ, τ, even when some of these symbols do not occur in the given constraints (that is, they are not in the set $\mathcal{F}$). In this case, we add the additional clauses:

$$\begin{array}{rl} \mathtt{Prec}_{f,g} & \forall f,g \in \mathcal{F} \text{ where } f = @^\sigma, g = @^\tau,\ \mathit{length}(\tau) \leq \mathit{length}(\sigma) \\ \neg\mathtt{Prec}_{f,g} & \forall f,g \in \mathcal{F} \text{ where } f = @^\sigma, g = @^\tau,\ \mathit{length}(\tau) > \mathit{length}(\sigma) \end{array}$$

Note that these clauses are added *only* for the symbols $@^\sigma, @^\tau$ which actually occur in $\mathcal{F}$, so this is a finite number of propositional formulas. Here, length returns the number of arrows which occur in a given type; certainly $\mathit{length}(\sigma \to \tau) > \mathit{length}(\sigma), \mathit{length}(\tau)$. We might also use any other metric which has this property. A precedence which satisfies the given constraints can be extended to a precedence $\trianglerighteq'$ on $\mathcal{F} \cup \{@^\sigma \mid \sigma \text{ any type}\}$, which has the property that $@^{\sigma\to\tau} \triangleright' @^\sigma, @^\tau$ for all types σ, τ.

Lemma 8.5. *Let $\blacktriangleright$ be a precedence on the elements of $\mathcal{F}$ such that its strict part $\blacktriangleright$ is well-founded, and such that for any two symbols of the form $@^\sigma, @^\tau$ which occur in $\mathcal{F}$ we have $@^\sigma \blacktriangleright @^\tau$ if and only if $length(\sigma) \geq length(\tau)$.*

Let $\trianglerighteq$ be the relation on symbols in $\{@^\sigma \mid \sigma \in \mathcal{T}\}$ given by: $@^\sigma \trianglerighteq @^\tau$ if $length(\sigma) \geq length(\tau)$.

Let $\blacktriangleright'$ be the relation on symbols in $\mathcal{F} \cup \{@^\sigma \mid \sigma \in \mathcal{T}\}$ which is the union of $\blacktriangleright$, $\trianglerighteq$, $\blacktriangleright \cdot \trianglerighteq$ and $\trianglerighteq \cdot \blacktriangleright$. Then $\blacktriangleright'$ is a well-founded precedence as well.

Proof. Reflexivity is inherited from reflexivity of $\blacktriangleright$ and $\trianglerighteq$. As for transitivity, this holds because $\blacktriangleright \cdot \trianglerighteq \cdot \blacktriangleright$ is included in $\blacktriangleright$, and $\trianglerighteq \cdot \blacktriangleright \cdot \trianglerighteq$ is included in $\trianglerighteq$:

- If $f \blacktriangleright g \trianglerighteq h \blacktriangleright i$, then g, h are both in $\mathcal{F}$. We can write $g = @^\sigma$ and $h = @^\tau$ with $length(\sigma) \geq length(\tau)$, and by the restrictions on $\blacktriangleright$ we thus have $g \blacktriangleright h$. Consequently, $f \blacktriangleright h$ by transitivity of $\blacktriangleright$.
- If $f \trianglerighteq g \blacktriangleright h \trianglerighteq i$, then $g = @^\sigma$ and $h = @^\tau$ (by the definition of $\trianglerighteq$). Since $@^\sigma \blacktriangleright @^\tau$ can only hold if $length(\sigma) \geq length(\tau)$, we thus have $f \trianglerighteq g \trianglerighteq h \trianglerighteq i$, and therefore $f \trianglerighteq i$.

Thus, using this combination and transitivity of $\blacktriangleright$ and $\trianglerighteq$, any sequence of $\blacktriangleright$ and $\trianglerighteq$ steps can be transformed into a sequence of at most two steps.

Finally, well-foundedness. Suppose $f_1 \blacktriangleright' f_2 \blacktriangleright' \ldots$, and never $f_{i+1} \blacktriangleright' f_i$. We can rewrite this to a sequence $g_1 \; R \; g_2 \; R \; \ldots$, where R is the union of $\blacktriangleright$ and $\triangleright$. Since $\blacktriangleright$ is well-founded, there are infinitely many indexes i where $g_i \triangleright g_{i+1}$. Let $i_1, i_2, \ldots$ be these indexes. By transitivity and reflexivity of $\blacktriangleright$, we see that $g_{i_1} \triangleright g_{i_1+1} \blacktriangleright g_{i_2} \triangleright g_{i_2+1} \blacktriangleright \ldots$. Using the observation made above, that in these instances $\blacktriangleright$ actually coincides with $\trianglerighteq$, this is an infinitely decreasing $\triangleright$ sequence, which provides the required contradiction as no type has infinite length. □

Thus, in the `PARTLY PRESERVING` case, if we satisfy the given propositional formulas, we obtain a well-founded precedence which has the property that always $@^{\sigma\to\tau} \blacktriangleright @^\sigma, @^\tau$, even for symbols $@^{\sigma\to\tau}, @^\sigma, @^\tau$ which do not occur in $\mathcal{F}$. The remaining requirement for this case, that always $f(\vec{s}) \cdot \vec{t} \gtrsim s_i \cdot \vec{c}$ if $f \in S$ and both sides have base type, is satisfied if we additionally add the constraints:

$$\texttt{Prec}_{f,g} \quad \forall\, f : [\sigma_1 \times \ldots \times \sigma_n] \longrightarrow \tau \in S, f \in \mathcal{F}, \text{ if } g = @^{\sigma_i} \text{ for some } i$$

Note that we already had a constraint that $\pi(f)$ for such functions is argument preserving, so either has the form $\lambda\vec{x}.f'(x_{i_1}, \ldots, x_{i_k})$, or $\lambda x.x$. In the former case, the additional requirement on the precedence gives the required inequality $f(\vec{s}) \cdot \vec{t} \gtrsim s_i \cdot \vec{c}$ as demonstrated in Section 6.6.2. In the latter case, note that the one argument of f has the same type. Therefore, using the function μ from Definition 5.36, we have $\overline{\pi}(\mu(f(s) \cdot \vec{t})) \gtrsim \overline{\pi}(\mu(f(s) \cdot \vec{c}))$ by clause (Bot), $\gtrsim \overline{\pi}(\mu(s \cdot \vec{c}))$ by clause (Select) (see Definition 5.19).

Status Function. For all symbols f, we introduce a symbol $\mathtt{Lex}_f$, which indicates that $stat(f') = \mathtt{lex}$, if defined. If f' is not defined (that is, if $\pi(f)$ has the form $\lambda\vec{x}.x_i$), then this variable has no meaning. For this status function, we only have to take care that it respects the equality induced by the precedence. Thus, we add the following constraints:

$$\begin{array}{ll} \neg\mathtt{Prec}_{f,g} \vee \neg\mathtt{Prec}_{g,f} \vee \neg\mathtt{Lex}_f \vee \mathtt{Lex}_g & \forall\, f,g \in \mathcal{F} \\ \neg\mathtt{Prec}_{f,g} \vee \neg\mathtt{Prec}_{g,f} \vee \mathtt{Lex}_f \vee \neg\mathtt{Lex}_g & \forall\, f,g \in \mathcal{F} \end{array}$$

8.6.3 Simplifying Constraints

Now, at last, we are prepared to tackle the todo list. We have an argument function, status function and symbol precedence, all defined by propositional variables. We also have a number of propositional variables of the form $\mathtt{GR}_{l,r}$ and $\mathtt{GE}_{l,r}$, and also (which we did not yet speak about) propositional variables $\mathtt{GE}_{l,r,A}$ where A is one of Fun, Select, F-Abs, Copy, Stat or stdr, and propositional variables of the form $\mathtt{GE}_{l,r,\mathrm{RST},l',n}$. Moreover, we have a todo list which will contain items of the following form:

- $l \succ r$, which must be simplified to: if $\mathtt{GR}_{l,r}$ holds, then $\overline{\pi}(l) \succ_\star \overline{\pi}(r)$ provided this is defined (**);
- $l \succsim r$, which must be simplified to: if $\mathtt{GE}_{l,r}$ holds, then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ provided this is defined (**);
- $l \succsim r$ (standard right), which must be simplified to: if $\mathtt{GE}_{l,r,\mathrm{stdr}}$ holds, and the root symbol of r is neither filtered away nor mapped to a minimal symbol (if it has one), then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$;
- $l \succsim r$ (A), where A is one of Fun, Select, F-Abs, Copy or Stat; this must be simplified to: if $\mathtt{GE}_{l,r,A}$ holds, then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ by clause (A) from Definition 5.19, provided the root symbol of l is not filtered away or mapped to a minimal symbol, and provided the same holds for the root symbol of r if it has one, and $\overline{\pi}(l), \overline{\pi}(r)$ are defined (**);
- $l \succsim r$ (restricted l', n) (where l' is a meta-term and n a number), which must be simplified to: if $\mathtt{GE}_{l,r,\mathrm{RST},l',n}$ holds, then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ provided the root symbol of r is neither filtered away nor mapped to a minimal symbol (if it has one).

  Although this interpretation is the same as that of $l \succsim r$ (standard right), this last constraint will be used in a more restrictive matter (where n and l' play a role). Constraints of this form are necessary to avoid infinite loops.

(**) We extend $\overline{\pi}$ to terms which might contain stars, by posing that if $\pi(f) = \lambda\vec{x}.f'(x_{i_1}, \ldots, x_{i_k})$, then $\overline{\pi}(f^\star(s_1, \ldots, s_n)) = f'^\star(s_{i_1}, \ldots, s_{i_k})$. This is not defined if $\pi(f)$ does not have this form.

For items in the todo list which are marked as DONE, suitable propositional formulas which guarantee the required inequalities are already present in the list of formulas. For items marked as TODO, they still need to be added.

In the rest of this section, we shall simplify constraints of one of these forms to a number of constraints in propositional formulas. The idea is this: for any item $\langle p, \texttt{TODO}\rangle$ in the todo list, which corresponds to a propositional variable X_p, we find a number of propositional formulas of the form $X_p \to \varphi_i$.[4] The formulas are designed in such a way that if each of these φ_i is satisfied, then the corresponding constraint must also hold. We store them in the list of formulas. We can then replace the item by $\langle p, \texttt{DONE}\rangle$, and for any propositional formulas of the form $\texttt{GR}_{l,r}$ or $\texttt{GE}_{l,r}$ or $\texttt{GE}_{l,r,stdr}$ or $\texttt{GE}_{l,r,A}$ or $\texttt{GE}_{l,r,\mathrm{RST},l',n}$ in the resulting φ_i, we add new constraints $\langle l \succ r, \texttt{TODO}\rangle$ or $\langle l \succsim r, \texttt{TODO}\rangle$ or $\langle l \succsim r \text{ (standard right)}\rangle$ or $\langle l \succsim r \ (A), \texttt{TODO}\rangle$ or $\langle l \succsim r \text{ (restricted } l', n), \texttt{TODO}\rangle$ to the todo list. If any of these constraints are actually already present in the todo list, whether with a tag TODO or DONE, then nothing needs to be added for them.

To guarantee that this process terminates, let us pose the following metric: the *norm* of a meta-term s is the sum of the number of variable, meta-variable and abstraction symbols occurring in it, plus twice the number of unmarked symbols, and once the number of marked symbols (so a symbol $f^\star$ counts only half as heavy as f). We will take care of the following restriction: if a constraint p is simplified to a formula containing a variable $\texttt{GE}_{l,r}$ or $\texttt{GE}_{l,r,A}$ or $\texttt{GE}_{l,r,\mathrm{RST},l',n}$ or $\texttt{GR}_{l,r}$, then the weight of the corresponding constraint is lower than the weight of the original constraint. Here, the *weight* of a constraint is a sequence, ordered lexicographically:

- $l \succ r$ has weight $\langle 2 \cdot norm(r) + 1, norm(l), 6\rangle$;
- $l \succsim r$ has weight $\langle 2 \cdot norm(r) + 1, norm(l), 6\rangle$;
- $l \succsim r$ (standard right) has weight $\langle 2 \cdot norm(r) + 1, norm(l), 5\rangle$;
- $l \succsim r \ (A)$ has weight $\langle 2 \cdot norm(r), norm(l), 4\rangle$ if A is one of Fun, F-Abs, Copy or Stat standard right or Select;
- $l \succsim r$ (Select) has weight $\langle 2 \cdot norm(r) + 1, norm(l), 4\rangle$
- $l \succsim r$ (restricted l', n) has weight; $\langle 2 \cdot norm(r) + 1, norm(l'), n, norm(l)\rangle$

With this restriction, we can never simplify a constraint infinitely long. Moreover, since every constraint is expressed as a formula over smaller constraints, we cannot get a proof of the form "$l \succsim r$ because $l \succsim r$".

[4] Since WANDA's formula class is minimal, and does not include an implication arrow, this is encoded as $\neg X_p \vee \varphi_i$. This has the added bonus that the formula generated by the StarHorpo module immediately has conjunctive normal form, so will not need additional transforming.

Simplifying By Type Restrictions. Any constraint $l \succ r$ or $l \succsim r$ or $l \succsim r$ $(strd)$ or $l \succsim r$ (A) or $l \succsim r$ (restricted l', n) can only hold if l and r have the same type. If this is not the case, store either $\neg\mathtt{GR}_{l,r}$ or $\neg\mathtt{GE}_{l,r}$ or $\neg\mathtt{GE}_{l,r,strdr}$ or $\neg\mathtt{GE}_{l,r,A}$ or $\neg\mathtt{GE}_{l,r,RST,l',n}$ respectively, and mark the constraint as done.

Simplifying $\succ$ constraints. To simplify an item $l \succ r$ from the todo list, if l does not have the form $\lambda\vec{x}.f(l_1,\ldots,l_n)$ (with $\vec{x}$ possibly empty), then we add the formula $\neg\mathtt{GR}_{l,r}$. For if l does not have this form, then whatever π is like, $\overline{\pi}(l)$ cannot be markable.

If, however, l does have this form, then we add the following formulas:

$$\begin{gathered}\neg\mathtt{GR}_{l,r} \vee \neg\mathtt{Minimal}_f \\ \neg\mathtt{GR}_{l,r} \vee \mathtt{SymbolFiltered}_f \vee \mathtt{GE}_{l^\star,r} \\ \neg\mathtt{GR}_{l,r} \vee \neg\mathtt{SymbolFiltered}_f \vee \mathtt{ArgFiltered}_{f,i} \vee \mathtt{GR}_{\lambda\vec{x}.l_i,r} \quad \forall\, 1 \leq i \leq n\end{gathered}$$

We could also write the second formula as $\mathtt{GR}_{l,r} \rightarrow (\mathtt{SymbolFiltered}_f \vee \mathtt{GE}_{l^\star,r})$, or as $(\mathtt{GR}_{l,r} \wedge \neg\mathtt{SymbolFiltered}_f) \rightarrow \mathtt{GE}_{l^\star,r}$. I will simply use a disjunctive form to express formulas like this, as it saves in parentheses, and this is also what happens in WANDA.

Thus, if $\mathtt{GR}_{l,r}$ holds, then if the root symbol of l is not filtered, we must have $\overline{\pi}(l^\star) = \overline{\pi}(l)^\star \succsim \overline{\pi}(r)$. If the root symbol of l *is* filtered away, so $\overline{\pi}(l) = \overline{\pi}(\lambda\vec{x}.l_i)$ for some i, we must have $\overline{\pi}(\lambda\vec{x}.l_i) \succ \overline{\pi}(r)$.

Simplifying Basic $\succsim$ Constraints. To simplify a todo item $l \succsim r$, we have a number of possibilities, depending on the form of l and r. To start, let us split this constraint into three cases: the root symbol of r is filtered, minimal, or it is neither.

If r does not have the form $g(r_1,\ldots,r_m)$, then we can immediately add an stdr tag, which boils down to adding the following constraint:

$$\neg\mathtt{GE}_{l,r} \vee \mathtt{GE}_{l,r,\mathrm{stdr}}$$

If r does have this form, we add the following constraints:

$$\begin{gathered}\neg\mathtt{GE}_{l,r} \vee \mathtt{SymbolFiltered}_g \vee \mathtt{Minimal}_g \vee \mathtt{GE}_{l,r,\mathrm{stdr}} \\ \neg\mathtt{GE}_{l,r} \vee \neg\mathtt{SymbolFiltered}_g \vee \mathtt{ArgFiltered}_{g,i} \vee \mathtt{GE}_{l,r_i} \quad \forall\, 1 \leq i \leq n\end{gathered}$$

Simplifying $\succsim$ Constraints with Standard Right. To simplify a todo item $l \succsim r$ (standard right), we have to look in particular at the form of l.

If l is a variable and r is the same variable, then we can simply mark the constraint as done (we always have $\overline{\pi}(l) = l \succeq_\star r = \overline{\pi}(r)$).

If l and r are both meta-variable applications with the same meta-variable, so $l = Z(l_1,\ldots,l_n)$ and $r = Z(r_1,\ldots,r_n)$, then we add the constraints:

$$\neg\mathtt{GE}_{l,r,\mathrm{stdr}} \vee \mathtt{GE}_{l_i,r_i} \quad \forall\, 1 \leq i \leq n$$

If l and r are both abstractions, and we can write them using α-conversion as $\lambda x.l'$ and $\lambda x.r'$, then we add the constraint $\neg\mathtt{GE}_{l,r} \vee \mathtt{GE}_{l',r'}$.

If l is a variable, meta-variable or abstraction, but none of these cases apply, then $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ cannot hold if the root symbol of r may not be filtered away. Thus, we simply add the formula $\neg\mathtt{GE}_{l,r,\mathrm{stdr}}$.

If l is a functional term $f(l_1,\ldots,l_n)$ with an unmarked root symbol, then the root symbol f might be filtered, in which case the question $l \succsim r$ reduces to a smaller problem. Or the root symbol f is not filtered, in which case $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ may hold either by the (Put) or (Fun) clause. However, f may definitely not be mapped to a minimal symbol, since r is not mapped to a minimal symbol.

$$\begin{gathered}\neg\mathtt{GE}_{l,r,\mathrm{stdr}} \vee \neg\mathtt{Minimal}_f \\ \neg\mathtt{GE}_{l,r,\mathrm{stdr}} \vee \neg\mathtt{SymbolFiltered}_f \vee \mathtt{ArgFiltered}_{f,i} \vee \mathtt{GE}_{l_i,r,\mathrm{stdr}} \quad \forall\, 1 \le i \le n \\ \neg\mathtt{GE}_{l,r,\mathrm{stdr}} \vee \mathtt{SymbolFiltered}_f \vee \mathtt{GE}_{l^\star,r,\mathrm{stdr}} \vee \mathtt{GE}_{l,r,\mathrm{Fun}}\end{gathered}$$

Finally, if l is a functional term with marked root symbol, $f^\star(l_1,\ldots,l_n)$, we may assume that f itself is not filtered. Taking into account the clauses which may be used to derive $l^\star \succsim r$, we can simply use the following formula:

$$\neg\mathtt{GE}_{l,r,\mathrm{stdr}} \vee \mathtt{GE}_{l,r,\mathrm{Select}} \vee \mathtt{GE}_{l,r,\text{F-Abs}} \vee \mathtt{GE}_{l,r,\mathrm{Copy}} \vee \mathtt{GE}_{l,r,\mathrm{Stat}}$$

Simplifying Fun $\succsim$ Constraints. Now we come to a more interesting form of constraints to simplify, for we will have to take permutations into account. To simplify a todo item $l \succsim r$ (Fun), we must find a formula that expresses that $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$, provided the root symbols of neither l nor r are filtered away, and using the (Fun) clause from Definition 5.36:

$$\begin{aligned} f(s_1,\ldots,s_n) \succeq_\star g(t_1,\ldots,t_m) &\text{ if n = m and } f \approx g \text{ and} \\ &\textit{either } stat(f) = Lex \text{ and } [s_1,\ldots,s_n] \succeq_{\star gLex} [t_1,\ldots,t_n] \\ &\quad \textit{or } stat(f) = Mul \text{ and } \{\{s_1,\ldots,s_n\}\} \succeq_{\star gMul} \{\{t_1,\ldots,t_n\}\} \end{aligned}$$

Here we use the *generalised lexicographic extension* and *generalised multiset extension* from the start of Section 5.1.

If either l or r is not an unmarked functional term, this is impossible, so we simply add the constraint $\neg\mathtt{GE}_{l,r,\mathrm{Fun}}$. Otherwise, write $l = f(l_1,\ldots,l_n)$ and $r = g(r_1,\ldots,r_m)$.

Unfortunately, we neither know exactly what the argument function does with the l and r, nor what status f and g have. What we do know (or at least, may assume) is that $\overline{\pi}(l) = f'(\overline{\pi}(l_{i_1}),\ldots,\overline{\pi}(l_{i_k}))$ and $\overline{\pi}(r) = g'(\overline{\pi}(l_{j_1}),\ldots,\overline{\pi}(l_{j,l}))$ for some l, k. We definitely need that $f' \approx g'$, and that $k = l$.

$$\begin{aligned} \neg\mathtt{GE}_{l,r,\mathrm{Fun}} \vee \mathtt{Prec}_{f,g} & \\ \neg\mathtt{GE}_{l,r,\mathrm{Fun}} \vee \mathtt{Prec}_{g,f} & \\ \neg\mathtt{GE}_{l,r,\mathrm{Fun}} \vee \neg\mathtt{ArgLengthMin}_{f,i} \vee \mathtt{ArgLengthMin}_{g,i} & \quad \forall\, 1 \le i \le \min(n,m) \\ \neg\mathtt{GE}_{l,r,\mathrm{Fun}} \vee \neg\mathtt{ArgLengthMin}_{g,i} \vee \mathtt{ArgLengthMin}_{f,i} & \quad \forall\, 1 \le i \le \min(n,m) \\ \neg\mathtt{GE}_{l,r,\mathrm{Fun}} \vee \neg\mathtt{ArgLengthMin}_{f,m+1} & \quad \text{if } n > m \\ \neg\mathtt{GE}_{l,r,\mathrm{Fun}} \vee \neg\mathtt{ArgLengthMin}_{g,n+1} & \quad \text{if } m > n \end{aligned}$$

Furthermore, we must have that $\overline{\pi}(l_{i_1}), \ldots, \overline{\pi}(l_{i_k}) \succeq_{\star gstat(f')} \overline{\pi}(r_{j_1}), \ldots, \overline{\pi}(r_{j_l})$. If $stat(f')$ is Lex, then this holds if and only if $\overline{\pi}(l_a) \succeq_\star \overline{\pi}(r_b)$ whenever $a = i_c$ and $b = j_c$ for some c. In formula:

$$\neg \mathtt{GE}_{l,r,\mathrm{Fun}} \vee \neg \mathtt{Lex}_f \vee \neg \mathtt{Permutation}_{f,i,j} \vee \neg \mathtt{Permutation}_{g,k,j} \vee \mathtt{GE}_{l_i,r_k}$$
$$\forall 1 \leq i \leq n, 1 \leq k \leq m, 1 \leq j \leq \min(n,m)$$

On the other hand, if $stat(f')$ is Mul, we must see that $\overline{\pi}(l_{\alpha(i_1)}) \succeq_\star \overline{\pi}(r_{j_1}), \ldots, \overline{\pi}(l_{\alpha(i_k)}) \succeq_\star \overline{\pi}(r_{j_k})$ for some fresh permutation α. In this case, the permutation of $\vec{l}$ and $\vec{r}$ does not matter at all, only which arguments are preserved by the argument function π. We might as well say: there is a function α which maps those elements of $\{1, \ldots, m\}$ which are not filtered away as arguments of f to those elements of $\{1, \ldots, n\}$ which are not filtered away as arguments of g, and always $\overline{\pi}(l_{\alpha(j)}) \succeq_\star \overline{\pi}(r_j)$. This function must be surjective; injectivity follows automatically because we already know that $k = l$.

To implement this, we introduce $n \cdot m$ new variables $\mathtt{A}_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$; we interpret $\mathtt{A}_{i,j}$ as "$\alpha(j) = i$". First we pose the constraints on $\mathtt{A}$:

$$\begin{aligned}
\neg \mathtt{ArgFiltered}_{f,i} \vee \neg \mathtt{A}_{i,j} &\qquad \forall 1 \leq i \leq n\ \forall 1 \leq j \leq m \\
\neg \mathtt{ArgFiltered}_{g,j} \vee \neg \mathtt{A}_{i,j} &\qquad \forall 1 \leq i \leq n\ \forall 1 \leq j \leq m \\
\neg \mathtt{GE}_{l,r,\mathrm{Fun}} \vee \mathtt{Lex}_f \vee \mathtt{ArgFiltered}_{f,i} \vee \mathtt{A}_{i,1} \vee \ldots \vee \mathtt{A}_{i,m} &\qquad \forall 1 \leq i \leq n \\
\neg \mathtt{GE}_{l,r,\mathrm{Fun}} \vee \mathtt{Lex}_f \vee \mathtt{ArgFiltered}_{g,j} \vee \mathtt{A}_{1,j} \vee \ldots \vee \mathtt{A}_{n,j} &\qquad \forall 1 \leq i \leq m \\
\neg \mathtt{A}_{i,j} \vee \neg \mathtt{A}_{k,j} &\qquad \forall 1 \leq i < k \leq n\ \forall 1 \leq j \leq m
\end{aligned}$$

Thus, the domain and range of the permutation cover only arguments which are not filtered away, and cover *all* such arguments; if the status is not Lex, or the constraint does not hold, we can just choose all $\mathtt{A}_{i,j}$ false.

Now we can use $\mathtt{A}$ to express that if $stat(f') = Mul$, then $l \gtrsim r$ (Fun) can only hold if $\overline{\pi}(l_{\alpha(j)}) \gtrsim \overline{\pi}(r_j)$ whenever j is not filtered away.

$$\neg \mathtt{GE}_{l,r,\mathrm{Fun}} \vee \mathtt{Lex}_f \vee \neg \mathtt{A}_{i,j} \vee \mathtt{GE}_{l_i,r_j} \quad \forall\ 1 \leq i \leq n, \forall\ 1 \leq j \leq m$$

Note that these fresh variables $\mathtt{A}_{i,j}$ are specific to this one simplification step; more accurately they might be expressed as $\mathtt{A}_{l,r,i,j}$. The next time a (Fun) step should be simplified, new propositional variables are introduced.

Simplifying Stat $\gtrsim$ Steps. The (Stat) case is similar to the (Fun) case, although there is a bit of additional complexity. Compared to the first-order case discussed in e.g. [27] there are not many novelties, although of course we must adapt for the fact that here the *generalised* lexicographic and multiset extension are considered.

We have $l = f^\star_{\sigma_{k+1},\ldots,\sigma_n,\tau}(l_1, \ldots, l_k, \ldots, l_n)$ with $k = ar(f)$. If r does not have the form $g(r_1, \ldots, r_m)$, then it cannot hold that $\overline{\pi}(l) \succeq_\star \overline{\pi}(r)$ using the (Stat) clause, so we simply store the variable $\neg \mathtt{GE}_{l,r,\mathrm{Stat}}$ and continue with the next

constraint. We do the same if r does have this form, but $k = 0$ (here, k is the arity of f).

Otherwise, r has the right form and $k > 0$. We may assume that g is not filtered away. Thus, we can write $\overline{\pi}(l) = f'^{\star}(\overline{\pi}(l_{i_1}), \ldots, \overline{\pi}(l_{i_f}), \overline{\pi}(l_{k+1}), \ldots, \overline{\pi}(l_n))$ and $\overline{\pi}(r) = g'(\overline{\pi}(r_{j_1}), \ldots, \overline{\pi}(r_{j_g}))$. We must find a number of formulas which together express that if $\mathtt{GE}_{l,r,\mathrm{Stat}}$ holds, then $f \approx g$, and:

- if $stat(f) = Lex$ then $[\overline{\pi}(l_{i_1}), \ldots, \overline{\pi}(l_{i_f})] \succ_{\star gLex} [\overline{\pi}(r_{j_1}), \ldots, \overline{\pi}(r_{j_g})]$;
- if $stat(f) = Mul$ then $\{\{l_1, \ldots, l_k\}\} \succ_{\star gMul} \{\{r_{j_1}, \ldots, r_{j_g}\}\}$

Moreover, for all $i \in \{j_1, \ldots, j_g\}$ we must have: if $r_i : \rho$ then $\overline{\pi}(f^{\star}_{\vec{\sigma},\rho}(\vec{l})) \succeq_{\star} \overline{\pi}(r_i)$.

The basic constraints are shared. Whatever the status of f, we must satisfy:

$$\begin{gathered}
\neg\mathtt{GE}_{l,r,\mathrm{Stat}} \vee \mathtt{Prec}_{f,g} \\
\neg\mathtt{GE}_{l,r,\mathrm{Stat}} \vee \mathtt{Prec}_{g,f} \\
\neg\mathtt{GE}_{l,r,\mathrm{Stat}} \vee \mathtt{ArgFiltered}_{g,i} \vee \mathtt{GE}_{f^{\star}(\vec{l}),r_i} \quad \forall\, 1 \leq i \leq m
\end{gathered}$$

For the rest, let us first consider lexicographic status. We have the required inequality $[\overline{\pi}(l_{i_1}), \ldots, \overline{\pi}(l_{i_f})] \succ_{\star gLex} [\overline{\pi}(r_{j_1}), \ldots, \overline{\pi}(r_{j_g})]$ if there is some number p such that $\overline{\pi}(l_{i_1}) \succsim \overline{\pi}(r_{j_1}), \ldots, \overline{\pi}(l_{i_{p-1}}) \succsim \overline{\pi}(r_{j_{p-1}})$ and either $j_g = p - 1$ while $i_f \geq p$, or $\overline{\pi}(l_{i_p}) \succ \overline{\pi}(r_{j_p})$.

We introduce new variables $P_1, \ldots, P_{\min(k,m+1)+1}$. Intuitively, the meaning of P_i is: the inequalities above hold for some $p \geq i$. Of course, $P_{\min(k,m+1)+1}$ is always false, but by using this propositional variable, we need fewer exceptions. First, we pose some formulas which guarantee that the intuition for the P_i formulas holds: $1 \leq p \leq i_f$ and $p \leq j_g + 1$. If not $\mathtt{GE}_{l,r,\mathrm{Stat}}$ or $stat(f) = Mul$, then we can always take all P_i false.

$$\begin{gathered}
\neg\mathtt{GE}_{l,r,\mathtt{Stat}} \vee \neg Lex_f \vee P_1 \\
\neg P_i \vee \mathtt{ArgLengthMin}_{f,i} \quad \forall\, 1 \leq i \leq \min(k, m+1) \\
\neg P_{i+1} \vee \mathtt{ArgLengthMin}_{g,i} \quad \forall\, 1 \leq i < \min(k, m+1) \\
P_i \vee \neg P_{i+1} \quad \forall\, 1 \leq i < \min(k, m) \\
\neg P_{\min(k,m+1)+1}
\end{gathered}$$

To express that $\overline{\pi}(l_{i_q}) \succsim \overline{\pi}(r_{j_q})$ for all $q < p$ and that if $p \geq j_f$ also $\overline{\pi}(l_{i_p}) \succ \overline{\pi}(r_{j_p})$, we add the following formulas for all $1 \leq a \leq k, 1 \leq b \leq m$ and $1 \leq i \leq \min(k, m)$:

$$\begin{gathered}
\neg P_{i+1} \vee \neg\mathtt{Permutation}_{f,a,i} \vee \neg\mathtt{Permutation}_{g,b,i} \vee \mathtt{GE}_{l_a,r_b} \\
\neg P_i \vee P_{i+1} \vee \neg\mathtt{Permutation}_{f,a,i} \vee \neg\mathtt{Permutation}_{g,b,i} \vee \mathtt{GR}_{l_a,r_b}
\end{gathered}$$

Next, let us consider multiset status. As in the (Fun) case, we can ignore the order of the arguments of $\overline{\pi}(l)$ and $\overline{\pi}(r)$. Combining the definition of the multiset extension with the possibility of filtered arguments, we observe that the required inequality holds if and only if there is a subset C of $\{1, \ldots, k\}$ and a partial function ζ which maps $\{1, \ldots, m\}$ to $\{1, \ldots, k\}$, such that:

- C contains only elements of $\{i_1, \ldots, i_f\}$, so argument positions of f which are not filtered away;
- ζ maps only to elements of $\{i_1, \ldots, i_f\}$, so argument positions of f which are not filtered away;
- the domain of ζ contains only elements of $\{j_1, \ldots, j_g\}$, so argument positions of g which are not filtered away, and contains *all* such positions;
- C is non-empty;
- if $\zeta(j) = i$ and $i \in C$, then $\overline{\pi}(l_i) \succ_\star \overline{\pi}(r_j)$;
- if $\zeta(j) = i$ and $i \notin C$, then $\overline{\pi}(l_i) \succeq_\star \overline{\pi}(r_j)$;
- if $\zeta(j_1) = \zeta(j_2)$ for two different indexes j_1 and j_2, then $\zeta(j_1) \in C$.

Encoding ζ with variables $Z_{i,j}$ for all $1 \leq i \leq k$ and $1 \leq j \leq m$, and encoding C with variables C_i for all $1 \leq i \leq k$, these properties can be expressed with the following propositional formulas:

$$\begin{array}{cl}
\neg C_i \vee \neg \mathtt{ArgFiltered}_{f,i} & \forall\, 1 \leq i \leq k \\
\neg Z_{i,j} \vee \neg \mathtt{ArgFiltered}_{f,i} & \forall\, 1 \leq j \leq m, \forall\, 1 \leq i \leq k \\
\neg Z_{i,j} \vee \neg \mathtt{ArgFiltered}_{g,j} & \forall\, 1 \leq j \leq m, \forall\, 1 \leq i \leq k \\
\mathtt{GE}_{l,r,\mathrm{Stat}} \vee \mathtt{Lex}_f \vee \mathtt{ArgFiltered}_{g,j} \vee & \\
Z_{1,j} \vee \ldots \vee Z_{k,j} & \forall\, 1 \leq j \leq m \\
\mathtt{GE}_{l,r,\mathrm{Stat}} \vee \mathtt{Lex}_f \vee C_1 \vee \ldots \vee C_k & \\
\neg Z_{i,j} \vee \neg C_i \vee \mathtt{GR}_{l_i,r_j} & \forall\, 1 \leq j \leq m, \forall\, 1 \leq i \leq k \\
\neg Z_{i,j} \vee C_i \vee \mathtt{GE}_{l_i,r_j} & \forall\, 1 \leq j \leq m, \forall\, 1 \leq i \leq k \\
\neg Z_{a,i} \vee \neg Z_{b,i} \vee C_i & \forall\, 1 \leq a < b \leq m, \forall\, 1 \leq i \leq k
\end{array}$$

Simplifying F-Abs and Copy Steps. The F-Abs case is a very straightforward matter. If r does not have the form $\lambda x.r'$, then we simply add the formula $\neg\mathtt{GE}_{l,r,\text{F-Abs}}$. If r does have this form, then τ has the form $\rho \to \alpha$. Writing $l' := f^\star_{\sigma_{k+1},\ldots,\sigma_n,\rho,\alpha}$, we have that $\overline{\pi}(l) \succsim \overline{\pi}(r)$ by clause (F-Abs) if and only if $\overline{\pi}(l') \succsim \overline{\pi}(r')$. Thus, we add:

$$\neg\mathtt{GE}_{l,r,\text{F-Abs}} \vee \mathtt{GE}_{l',r'}$$

The Copy case is also simple, although it gives a few more constraints. The right-hand side r must have the form $g(r_1, \ldots, r_m)$ (if not, we simply put down $\neg\mathtt{GE}_{l,r,\mathrm{Copy}}$), and we need that $f \blacktriangleright g$, and that $f^\star(\vec{l}) \succeq_\star r_i$ for all i (where the type declaration of $f^\star$ is updated as necessary):

$$\begin{array}{cl}
\neg\mathtt{GE}_{l,r,\mathrm{Copy}} \vee \mathtt{Prec}_{f,g} & \\
\neg\mathtt{GE}_{l,r,\mathrm{Copy}} \vee \neg\mathtt{Prec}_{g,f} & \\
\neg\mathtt{GE}_{l,r,\mathrm{Copy}} \vee \mathtt{GE}_{f^\star_{\vec{\sigma},\rho_i}(\vec{l}),r_i} & \forall\, 1 \leq i \leq m
\end{array}$$

Simplifying Select $\succsim$ Steps. The (Select) step, however, requires special attention. In all other cases, we can easily reduce the constraint to a formula which uses constraints with a lower weight, but in the (Select) case, this is not so. In fact, if we do not watch out, we might well end up in a loop: $\mathtt{f}^\star(\lambda x.\mathtt{g}(x)) \succsim s$ because $\mathtt{g}(\mathtt{f}^\star(\lambda x.\mathtt{g}(x)) \succsim s$ because $\mathtt{g}^\star(\mathtt{f}^\star(\lambda x.\mathtt{g}(x)) \succsim s$ because $\mathtt{f}^\star(\lambda x.\mathtt{g}(x)) \succsim s$ because ...

This is where the restricted constraints "$l \succsim r$ (restricted l', n)" come in. The meaning of this constraint is the same as $l \succsim r$ (standard right), except that we consider a restriction on how many (Select) steps may be taken before we have to take a really decreasing step. We do not need to allow for many successive non-decreasing steps; let us say 3 suffices. Let $l = f^\star(l_1, \ldots, l_m)$. Introducing new variables $\mathtt{ArgSelected}_i$ for $1 \leq i \leq m$, we add the following formulas:

$$\begin{aligned} \neg\mathtt{GE}_{l,r,\mathrm{Select}} \vee \mathtt{ArgSelected}_1 \vee \ldots \vee \mathtt{ArgSelected}_m & \\ \neg\mathtt{ArgSelected}_i \vee \neg\mathtt{ArgFiltered}_{f,i} & \quad \forall 1 \leq i \leq m \\ \neg\mathtt{ArgSelected}_i \vee \mathtt{GE}_{l_i,r,\mathrm{RST},l,3} \vee \ldots \vee \mathtt{GE}_{l_i\langle f^\star(\vec{l}),\ldots,f^\star(\vec{l})\rangle,r,\mathrm{RST},l,3} & \quad \forall 1 \leq i \leq m \end{aligned}$$

In the last group of formulas, all well-defined, well-typed meta-terms of the form $l_i\langle f^\star(\vec{l}), \ldots, f^\star(\vec{l})\rangle$ occur.

The constraints corresponding to the $\mathtt{GE}_{l_i\langle\ldots\rangle,r,\mathrm{RST},l}$ variables have a lower weight in the third component than the $l \succsim r$ (Select) constraint. If we had chosen a larger bound than 3, then we would have had to alter the third component in the definition of weight a bit. However, as it takes some effort to construct examples where even a bound of 3 is necessary, it is not likely that we will need a larger bound in practical situations.

Simplifying Restricted $\succsim$ Steps. The restricted constraints $l \succsim r$ (restricted l', n) are very similar in nature to the "standard right" constraints – they are merely more restrictive. Thus, if weight permits, we will simply use the latter instead. That is, if $norm(l) < norm(l')$, we merely add the following formula:

$$\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \mathtt{GE}_{l,r,\mathrm{stdr}}$$

Otherwise, suppose $norm(l) \geq norm(l')$. Because we are doing *restricted* evaluation, if $n = 0$ we simply choose not to go on. That is, we merely add the following formula:

$$\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n}$$

Otherwise, additionally suppose that $n > 0$. We have to deal with most of the cases we also had in the standard right situation. Mostly, we can do this in the same way. If l is a variable and r the same variable, we do not have to do anything. If $l = Z(l_1, \ldots, l_n)$ and $r = Z(r_1, \ldots, r_n)$, then we add the constraints:

$$\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \mathtt{GE}_{l_i,r_i} \quad \forall\, 1 \leq i \leq n$$

Note that the first component of the weight is decreased. If $l = \lambda x.s$ and $r = \lambda x.t$, then we add the formula $\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \mathtt{GE}_{s,t}$. If l is a variable, meta-variable application or abstraction and these cases do not apply, then we add the formula $\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n}$. If l is a functional term $f(l_1,\ldots,l_n)$ with an unmarked root symbol, then we add constraints:

$$\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \neg\mathtt{Minimal}_f$$
$$\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \neg\mathtt{SymbolFiltered}_f \vee \neg\mathtt{ArgFiltered}_{f,i} \vee \mathtt{GE}_{l_i,r,\mathrm{RST},l',n}$$
$$\forall\, 1 \leq i \leq n$$
$$\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \mathtt{SymbolFiltered}_f \vee \mathtt{GE}_{l^\star,r,\mathrm{RST},l',n} \vee \mathtt{GE}_{l,r,Fun}$$

Finally, and most interestingly, suppose $l = f^\star(l_1,\ldots,l_m)$, a functional term with marked root symbol. In this case, to avoid a need to complicate the weight function any further, let us include a (Select) step immediately. We once more add fresh variables $\mathtt{ArgSelected}_i$ for all i in $\{1,\ldots,n$, and add the formula:

$$\neg\mathtt{GE}_{l,r,\mathrm{RST},l',n} \vee \mathtt{GE}_{l,r,\text{F-Abs}} \vee \mathtt{GE}_{l,r,\mathrm{Copy}} \vee \mathtt{GE}_{l,r,\mathrm{Stat}} \vee$$
$$\mathtt{ArgSelected}_1 \vee \ldots \vee \mathtt{ArgSelected}_m$$

And in addition for all $1 \leq i \leq m$:

$$\neg\mathtt{ArgSelected}_i \vee \neg\mathtt{ArgFiltered}_{f,i}$$
$$\neg\mathtt{ArgSelected}_i \vee \mathtt{GE}_{l_i,r,\mathrm{RST},l',n-1} \vee \ldots \vee \mathtt{GE}_{l_i\langle f^\star(\vec{l}),\ldots,f^\star(\vec{l})\rangle,r,\mathrm{RST},l',n-1}$$

8.6.4 Finishing Up

In the end, we have ticked off all items of the todo list, and the list of formulas encodes the StarHorpo problem. Since all these formulas are clauses, we can pass their conjunction to a SAT-solver unmodified. If the SAT-solver finds a solution, the values for $\mathtt{GR}_{l,r}$ give us at least one constraint which is strictly oriented.

Most of the ideas used in this encoding of StarHorpo are not new; the encoding is in many ways similar to the encoding of RPO in [27]. However, we have to deal with new challenges: the use of star-marked terms, a definition based on a reduction pair $(\succsim, \succ)$ rather than a pair $(\sim, \succ)$ of an equivalence relation and a well-founded ordering. And, very importantly, a definition which is not by nature decreasing, due to the special (Select) rule: we had to circumvent possible loops in the derivation with "restricted" constraints.

Another relevant difference is the way variables corresponding to formulas are used: where the authors of [27] express the meaning of such variables in the form $\mathtt{GE}_{l,r} \leftrightarrow \varphi$ for some formula φ, WANDA essentially uses $\mathtt{GE}_{l,r} \rightarrow \varphi$ (the conjunction of the clauses involving $\neg\mathtt{GE}_{l,r}$ can be expressed in such a form). This choice has both up- and downsides. By not using an "if and only if" arrow, WANDA obtains easier formulas. However, the eventual assignment of propositional variables may have false negatives. In particular, it may be that $\mathtt{GR}_{l,r}$ is false, even if $\overline{\pi}(l) \succ_\star \overline{\pi}(r)$ holds with the given argument function, precedence and status. However, this is hardly an issue as we can simply test the value of φ, and set $\mathtt{GR}_{l,r}$ to true if necessary!

8.7 Experimental Results

To empirically assess the power both of the techniques in this work and of their implementation, WANDA has been run on the higher-order part of the termination problem database (TPDB) version 8.0.1 [126], which is used in the annual termination competition. This version of the database has 156 benchmarks for the higher-order category. Of these, 46 are designed as higher-order termination problems, originating for instance in papers on higher-order termination techniques. The other 110 problems are originally applicative TRSs, which have been assigned a simple type and uncurried. These typically have a larger first-order part. The benchmarks are all presented in the AFS formalism of Section 3.4.

In order to work, WANDA needs to be coupled with a SAT-solver and, ideally, a first-order termination tool. Because of the termination competition [125] and the international SAT competition [28], there are standard input formats for both. This makes it possible to pick an arbitrary existing tool and couple it with WANDA. For the experiments, WANDA uses the SAT-solver MiniSat [32] and the first-order termination tool AProVE [45] as back-ends.

Non-termination. The non-termination module of WANDA finds 9 non-terminating benchmarks. The use of a first-order tool in the dependency pair framework surprisingly does not bring up any non-termination results, even though step 2 of the dependency pair algorithm has a case where a `NO` for the first-order part gives non-termination for the higher-order system. Unfortunately, in the cases where the first-order part is non-terminating, this first-order part is generally not an overlay TRS. Thus, non-termination of the TRS does not imply non-termination of the AFSM, as types might make a difference. At present, WANDA does not analyse the counterexample from the first-order prover to see whether it is typable. However, this is an obvious direction for future extensions.

Termination. Of the remaining 147 benchmarks, at least 12 are non-terminating. WANDA can handle most of the others: 123 benchmarks are proved terminating. As in the termination competition, a one minute timeout is imposed, but the timeout makes little difference; where WANDA succeeds, she typically succeeds within seconds. This is despite the implementation choices, where efficiency takes a back seat to simplicity of the code. The timeouts are mostly caused by WANDA's strategy of trying more advanced polynomial interpretation shapes when easier shapes and StarHorpo do not succeed.

Where does WANDA fail? Most importantly on systems whose termination is not known. When a first-order termination prover fails on the first-order part, WANDA invariably fails as well, even regarding types. WANDA's use of type information is limited; sometimes type changing functions are employed, but this is not enough for systems like Example 5.8. Minor improvements may be possible by for instance implementing more exciting interpretation shapes for weakly monotonic functionals. But all in all, WANDA is doing pretty well as it is!

Comparative Analysis. To see which techniques give the most power, Figure 8.1 lists various configurations of WANDA.

Configuration	#YES
Full WANDA	123
Rule removal only	77
Rule removal with polynomials only	46
Rule removal with StarHorpo only	71
Dependency pairs only (no rule removal beforehand)	123
Dynamic dependency pairs only	120
Static dependency pairs only	105
Static dependency pairs only plus rule removal	114
Dynamic dependency pairs only, no tagging	99

Figure 8.1: Evaluation of WANDA on the TPDB 8.0.1 with various settings

It is evident from this table that, although we already get quite far with rule removal, a dependency pair framework is fundamental to WANDA's power. Rule removal makes no difference, although it often leads to shorter and more easily understandable termination proofs than we obtain using dependency pairs.

In the "dynamic dependency pairs only" configuration, only static dependency pairs (and rule removal beforehand) were disabled; the rest of the dependency pair framework was allowed. Interestingly, and somewhat unexpectedly, we lose relatively little power. Even though collapsing dependency pairs give many problems – a denser dependency graph, more dependency pairs and a more limited possibility to use the subterm criterion and usable rules – WANDA can handle almost all the benchmarks that she can normally deal with in this configuration. In contrast, the restriction to static dependency pairs loses a fair bit. In many cases this is because the system under consideration is not plain function passing; as the next setup setup shows, combining static dependency pairs with rule removal gives a relevant improvement.

The last setup uses the dependency pair framework as normal, but both without static dependency pairs, and without the results given by the abstraction-simplicity restriction. If we don't use tagging, we are still better off than in the rule removal case, but significantly worse than in the full dynamic approach.

Both polynomial interpretations and StarHorpo can tackle a fair few systems without the help of the dependency pair framework. Putting polynomial interpretations and StarHorpo next to each other, we see two things. First, the approaches are incomparable: neither method can deal with all systems which can be handled with the combination. Second, StarHorpo clearly trumps polynomial interpretations. To a large extent, this is due to the nature of many typical higher-order examples: when some form of primitive recursion is implemented (such as Example 5.7), higher-order polynomials fail, because the functions grow too fast for polynomials. StarHorpo has no problem with such systems.

However, polynomial interpretations really find their power in the dependency pair framework, where the strong monotonicity restriction can be dropped. Figure 8.2 compares the respective power of polynomial interpretations and StarHorpo in the full dependency pair framework.

Configuration	**#YES**
Full WANDA without polynomials	120
Full WANDA without StarHorpo	116

Figure 8.2: Comparing the power of polynomial interpretations and StarHorpo

Now we see somewhat more equal results. StarHorpo still outperforms polynomial interpretations, but with a less solid margin. Interestingly, almost all of the 123 benchmarks which can be handled, can be handled by either one or the other. The only exception is the system `twice` from Example 6.13. Although this system *can* be handled with only polynomial interpretations, WANDA's limited interpretation shapes do not suffice.

Finally, let us consider the respective power of the various minor techniques in the dependency pair framework. The details are given in Figure 8.3.

Configuration	**#YES**
Dependency pair framework	123
Dependency pairs without first-order termination tool	117
Dependency pairs without formative rules	121
Dependency pairs without usable rules	123
Dependency pairs without subterm criterion	122
Dependency pairs without graph	122
Dependency pairs with *only* subterm criterion and graph	73
Dependency pairs with *only* SC, graph and first-order tool	102
Dynamic dependency pairs with only SC, graph, and FO-tool	8

Figure 8.3: Evaluation of minor techniques in the dependency pair framework

This figure shows that the use of a first-order termination tool helps a fair bit. The dependency graph, formative rules and the subterm criterion make a minor difference, and usable rules turn out not to make a difference at all.

However, it may be argued that this is not the fault of the techniques. Rather, the reduction pairs WANDA uses are so powerful that they do not need the help! This is why the last three tests were included. In this setting, the subterm criterion was used, together with the dependency graph and possibly a first-order termination tool. The experiments show that in the dependency pair framework (including the graph), the subterm criterion on its own, or maybe combined with a first-order termination prover, but *without the use of a reduction pair*, can already handle a rather large part of the database. As the last experiment shows, this is mostly due to the use of static dependency pairs.

8.8 Overview

In this chapter we have seen the fundamental ideas of how the techniques in this thesis can be implemented in an automatic termination tool. Not only *can* this be done, it *has* been done. The termination tool WANDA, which was written as a part of the research for this thesis, implements nearly all techniques discussed in this work (and nothing that was not discussed). WANDA is written in C++ and totals approximately 17000 lines of code. Both the code and the executable can be downloaded at:

http://wandahot.sourceforge.net/

Despite her current simplicity – many advanced features of the first-order world have not been extended, and the implementation focuses on straightforward and easily extendable code over efficiency – WANDA can handle most terminating benchmarks in the higher-order category of the termination problem database. Since 2010 WANDA has participated in the annual termination competition; in 2011 she ended first.

It is worth noting that WANDA is designed for a more general setup than she currently receives: built on the AFSMs of this thesis, she should be able to handle more than the monomorphic AFSs in the TPDB. However, due to lack of available input, the performance on for instance HRSs remains to be evaluated (and perhaps optimised) in the future. Other future extensions will probably include polymorphism.

Conclusions 9

Or, Where does this bring us?

Termination of higher-order term rewriting is an interesting topic, both for theoretical and for practical reasons. Due to the presence of β-reduction, and the resulting explosion of complexity, the field is often spurned as too difficult. However, as I hope this work has demonstrated, this is unjust, for not only can many results from the first-order world be extended, it is also possible to obtain several results unique to the higher-order setting.

9.1 Overview

In the preceding chapters we have been through a whirlwind adventure of higher-order termination techniques. A recurring theme in this was the use of *transformations*. Transformations often make it possible to bring a problem into an easier shape, or to phrase termination techniques in a simple way.

First, in Chapters 2 and 3, we have seen the new AFSM formalism and many other formalisms of higher-order rewriting which are used in some area or other. Using termination-reflecting transformations, we can swap between the various formalisms, and transform systems, for instance η-expanding them.

In the next two chapters, we have seen ways to generate a reduction pair. The first (Chapter 4), *weakly monotonic algebras*, is an existing termination technique, which can be transposed to the AFSM formalism using a transformation. Moreover, we have considered the class of polynomial interpretations, which is highly suitable for automation (as witnessed in Chapter 8). The second reduction pair (Chapter 5), the *higher-order iterative path ordering* together with its recursive counterpart, extends the first-order iterative and recursive path ordering, and has both advantages and disadvantages compared to existing higher-order path orderings. A transformation to "application-free" terms makes it possible to phrase this path ordering in a relatively simple way.

Finally, Chapters 6 and 7 consider the method of *dependency pairs*. In this method, the termination question is reduced to a number of *dependency pair problems*, which are iteratively simplified. Transformations play a major role in

this approach, as dependency pairs and dependency chains are transformed, in order to reduce a dependency pair problem to a smaller one.

All these methods combine to give a powerful toolkit for termination analysis. This is demonstrated by the termination tool WANDA (Chapter 8), which implements almost all contributions of this work. Testing WANDA on the termination problem database shows that termination of most typical examples of higher-order term rewriting systems can be tackled with this theory, even automatically.

9.2 Practical Applications

Higher-order term rewriting, while an interesting theoretical topic that generalises for instance standard first-order term rewriting, applicative term rewriting and the λ-calculus, also serves some practical interest. In particular, it is the underlying structure of many functional programming languages. Higher-order termination analysis could therefore be used to determine whether a program in certain functional languages will eventually reach a solution regardless of input – a very useful feature which future compilers may automatically test.

A particular example of this is the functional CRSX language discussed in Chapter 3.5. Unlike many other functional programming languages, CRSXs do not come with an evaluation strategy, so they are particularly suitable for handling with general techniques (like the ones considered here). In the future, it may be possible to determine automatically whether or not a compiler specified in CRSX always finishes compiling – perhaps even using extensions of WANDA.

Other examples are Coq and Isabelle, where functions *have* to be proved terminating before they can be used. The more we can do this automatically, the more pleasant it is for the user. In [84] IsaFoR interfaces between Isabelle and the first-order termination provers AProVE and $\mathrm{T_TT_2}$ to automatically prove termination (and thus allow the use) of first-order functions in Isabelle. In the future, it may be possible to do something similar with higher-order functions as well.

9.3 Polymorphism

The simple types used in this work suffice for many practical purposes, but in a fair few applications other type systems are used. A common example is the use of *polymorphic* types. With polymorphic types, one might for instance declare function symbols $\texttt{cons} : [\alpha \times \texttt{list}(\alpha)] \longrightarrow \texttt{list}(\alpha)$ and $\texttt{nil} : \texttt{list}(\alpha)$, and create both terms $\texttt{cons}(\texttt{37}, \texttt{nil})$ with $\texttt{37} : \texttt{int}$ and $\texttt{cons}(\texttt{true}, \texttt{cons}(\texttt{false}, \texttt{nil}))$ with $\texttt{true}, \texttt{false} : \texttt{bool}$. Let us consider a possible definition of polymorphic AFSMs, and see how much effort it would take to derive termination results for it.

Given a set of *type constructors* $\mathcal{B}$, each with a fixed arity, and an infinite set of *type variables* $\mathcal{A}$, *polymorphic types* are built according to the grammar:

$$\mathcal{T} ::= \alpha \mid b(\mathcal{T}^n) \mid \mathcal{T} \to \mathcal{T} \qquad (\alpha \in \mathcal{A},\ b \in \mathcal{B},\ ar(b) = n)$$

A *monomorphic* type does not contain type variables. Examples of monomorphic types are $\mathtt{nat}$ or $\mathtt{list}(\mathtt{nat}) \to \mathtt{list}(\mathtt{bool} \to \mathtt{nat})$, and an example of a non-monomorphic type is $\alpha \to \mathtt{list}(\alpha)$.

A type substitution θ is a mapping $[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]$, and the application $\tau\theta$ of a type substitution θ to a type τ replaces each α_i in τ by σ_i. The application of a type substitution to a type declaration is defined similarly.

A *polymorphic signature* is a signature $\mathcal{F}$ where the type declarations of the function symbols may be polymorphic. For any polymorphic signature, let $\mathcal{F}^{\text{mono}}$ be the set $\{f_{\sigma\theta} : \sigma\theta \mid f : \sigma \in \mathcal{F},\ \theta \text{ a type substitution, } \sigma\theta \text{ monomorphic}\}$. Polymorphic meta-terms are built from $\mathcal{F}^{\text{mono}}$ as in Section 2.2.

Rule schemes are rules defined using polymorphic meta-terms. A set of rule schemes R defines a set of rules as follows: $\mathcal{R}_R := \{l\theta \Rightarrow r\theta \mid l \Rightarrow r \in R$ and θ is a type substitution such that $l\theta$ and $r\theta$ are both monomorphic meta-terms$\}$ (applying a type substitution to a meta-term applies the type substitution to the types of function symbols, variables and meta-variables occurring in the meta-term). It is not hard to see that $\Rightarrow_{\mathcal{R}_R}$ is terminating if and only if it is terminating on monomorphic terms.

With this definition, polymorphic types are *only* used in rule schemes. To prove termination of the relation generated by a set R of rule schemes, we simply use termination methods for monomorphic AFSMs and apply them on $\mathcal{R}_R$. The only trouble with this simple solution is that $\mathcal{R}_R$ will invariably be infinite.

Thus, we might have to extend the methods a little: derive results of the form "if $l \gtrsim^! r$, then $l\theta \gtrsim r\theta$ for all type substitutions θ mapping all type variables in l, r to monomorphic terms". Here, $\gtrsim^!$ is some new relation on polymorphic terms. It is certainly possible to find results like this: for example if $f : [\sigma] \longrightarrow \sigma$, then $f(s) \succeq_\star s$ using the StarHorpo reduction pair for any type instantiation of σ. The dependency pair approach, too, easily bends to the polymorphic case: we just have to make groups of dependency pairs (this is discussed in [80]).

Although the generalisations to the polymorphic case still need to be done, and will likely take some effort, strong results in this area seem to be in reach.

Example 9.1. Consider a polymorphic version of `map`, with function symbols $\mathtt{nil} : \mathtt{list}(\alpha)$, $\mathtt{cons} : [\alpha \times \mathtt{list}(\alpha)] \longrightarrow \mathtt{list}(\alpha)$ and $\mathtt{map} : [(\alpha \to \alpha) \times \mathtt{list}(\alpha)] \longrightarrow \mathtt{list}(\alpha)$, and the following rules:

$$\begin{array}{lcl} \mathtt{map}(\lambda x.F(x), \mathtt{nil}) & \Rightarrow & \mathtt{nil} \\ \mathtt{map}(\lambda x.F(x), \mathtt{cons}(X, Y)) & \Rightarrow & \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y)) \end{array}$$

Suppose we choose a type changing function of the form $\zeta(\mathtt{list}(\sigma)) = \zeta(\sigma)$. Then for any instance of the `map` rules we can use StarHorpo as follows:

1. $\mathtt{map}^\star_{\zeta(\mathtt{list}(\sigma))}(\lambda x.F(x), \mathtt{cons}(X, Y)) \succ_\star \mathtt{cons}(F(X), \mathtt{map}(\lambda x.F(x), Y))$ because $\mathtt{map} \blacktriangleright \mathtt{cons}$ and 2, 5 (using (Copy))
2. $\mathtt{map}^\star_{\zeta(\sigma)}(\lambda x.F(x), \mathtt{cons}(X, Y)) \succ_\star F(X)$ because 3 (using (Select))

3. $\mathtt{map}^\star_{\zeta(\sigma)}(\lambda x.F(x), \mathtt{cons}(X,Y)) \succ_\star X$ because 4 (using (Select))
4. $\mathtt{cons}(X,Y) \succ_\star X$ because $\zeta(\mathtt{list}(\sigma)) = \zeta(\sigma)$ and $\mathtt{cons}^\star(X,Y) \succ_\star X$ by (Select)
5. $\mathtt{map}^\star_{\zeta(\mathtt{list}(\sigma))}(\lambda x.F(x), \mathtt{cons}(X,Y)) \succ_\star \mathtt{map}(\lambda x.F(x), Y)$ because 6 (using (Stat))
6. $\mathtt{cons}(X,Y) \succ_\star Y$ because $\mathtt{cons}^\star(X,Y) \succ_\star Y$ by (Select).

Note that the type changing function is essential in step 4; it is used to relate a list to the elements of that list. This example seeks to demonstrate that the techniques defined so far can be used to prove termination of polymorphic systems as well – but doing so does require a bit of extra thought.

9.4 Future Work

The study of higher-order termination is by no means complete – far from it. One obvious direction of further research is the extension of existing termination methods for first-order rewriting. Some of the more common ones have already been extended, but there are decades of first-order results to generalise.

In a different direction, we might consider termination analysis for formalisms with a more complex typing system. One such extension is an extension with *polymorphic types*, as discussed in Section 9.3. We may also for instance consider *dependent types* (see e.g. [58]) or type systems with *subtyping* (see e.g. [60]).

Or we could look into modularity results. One such result we saw in Chapter 7.7: we can split off the first-order part of a higher-order system, and analyse it separately, provided we use dependency pairs for the remainder. But could we perhaps go further, and also for instance consider (part of) an AFSM as an applicative higher-order system without β-reduction?

Finally, an important direction for future research is to consider higher-order term rewriting with a particular strategy. As many functional programming languages use an evaluation strategy, such as lazy, weak or innermost evaluation, it would be useful to have dedicated methods for it. For although general termination implies termination with any strategy, we do not yet have methods to deal with systems whose termination *depends* on the strategy.

9.5 Final Remarks

In this thesis, we have seen numerous techniques to analyse termination of higher-order term rewriting systems, and have also run into some of their limitations. Higher-order termination analysis is an interesting field, which provides many challenges. Looking to the future, we have a long way to go yet, as many open questions lie unexplored. But there is every reason to believe these problems can be tackled!

Thesis Summary

Hogere Orde Terminatie – Samenvatting in het Nederlands

Hogere orde termherschrijven is een samenvattende naam voor diverse formalismen. Deze formalismen kunnen gebruikt worden als een hogere orde logica, of als het onderliggende systeem voor functionele programmeertalen. Terminatie van termherschrijfsystemen is een belangrijk onderzoekspunt, omdat het nauw samenhangt met correctheid.

In dit proefschrift hebben we in het bijzonder gekeken naar het *AFSM* formalisme. Dit formalisme, dat hier geïntroduceerd is (maar sterk overeenkomt met een eerdere definitie in [13]) kan gebruikt worden om diverse bestaande hogere orde formalismen tegelijk te bestuderen. We hebben gezien, met name in Hoofdstuk 2 en 3, hoe *transformaties* gebruikt kunnen worden om een hogere orde termherschrijfsysteem in een andere (en vaak eenvoudigere!) vorm te presenteren. Ook kunnen zij gebruikt worden om bestaande terminatietechnieken een nieuw tintje te geven.

Hogere orde termherschrijfsystemen vormen een uitbreiding van *eerste orde termherschrijfsystemen*. Eerste orde systemen zijn over het algemeen makkelijker te analyseren dan hogere orde systemen, en zodoende bestaan er veel resultaten. Het ligt voor de hand om te proberen deze resultaten ook in de hogere orde omgeving af te leiden. Hiervoor hebben we met name drie methodes onderzocht: *polynoominterpretaties*, *padordeningen* en *afhankelijkheidsparen*. Dit zijn enkele van de belangrijkste terminatietechnieken in de eerste orde wereld, en daarbij zijn ze automatiseerbaar: er bestaan diverse programma's die in veel gevallen automatisch kunnen afleiden of een gegeven eerste orde systeem termineert.

Automatisering is een belangrijk aspect van terminatie-onderzoek, ook in de hogere orde wereld. Het is immers prachtig om te weten dat een programma altijd een oplossing vindt, maar het is nog beter als je zelf geen moeite hoeft te doen om dit aan te tonen! Daarom zijn vrijwel alle technieken die in dit werk zijn geïntroduceerd (of overgenomen van andere formalismen) geïmplementeerd in het terminatietool WANDA. WANDA is te vinden op:

http://wandahot.sourceforge.net

Higher Order Termination: Summary in English

Higher-order term rewriting is a collective name for a number of different formalisms. These formalisms can for instance be used as a higher-order logic, or as the underlying framework of functional programming languages. Termination of term rewriting systems is an important research topic, as it is closely connected with correctness.

This thesis studies termination of higher-order term rewriting. In particular, we consider the *AFSM formalism*, which can be used to analyse various existing formalisms in one go. A recurring theme in these approaches is the use of transformations, which can often be used to bring a termination problem in an easier shape. Transformations are explored in particular in the early chapters, 2 and 3.

Higher-order term rewriting systems are an extension of *first-order term rewriting systems*. First-order systems are typically easier to analyse, and consequently many termination results already exist. An obvious direction for research is to extend these existing methods to the higher-order setting. In this thesis we study three such methods: *polynomial interpretations*, *path orderings* and *dependency pairs*. These are some of the most prominent termination techniques in the first-order world, and moreover they are automatable: there are numerous tools which can automatically prove or disprove termination of a large number of first-order term rewriting systems.

Automation is a crucial aspect of termination research, in the higher-order setting as well as the first-order one. For while it is nice to know that a program will always find a solution, it is much nicer if you don't have to make a manual effort to show this! For that reason, almost all techniques introduced in this work have been implemented in the termination tool WANDA, available at:

http://wandahot.sourceforge.net

Bibliography

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In C. Hankin and D. Schmidt, editors, *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '01)*, pages 104–115. ACM, 2001.

[2] A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.

[3] A. Abel. Semi-continuous sized types and termination. In Z. Ésik, editor, *Proceedings of the 20th International Workshop on Computer Science Logic (CSL '06)*, volume 4207 of *LNCS*, pages 72–88. Springer, 2006.

[4] P. Aczel. A general Church-Rosser theorem. Unpublished Manuscript, University of Manchester. http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf, 1978.

[5] Y. Akama. On Mint's reduction for ccc-calculus. In M. Bezem and J. Groote, editors, *Proceedings of the 1st International Conference on Typed Lambda Calculus and Applications (TLCA '93)*, volume 664 of *LNCS*, pages 1–12. Springer, 1993.

[6] T. Aoto and Y. Yamada. Dependency pairs for simply typed term rewriting. In J. Giesl, editor, *Proceedings of the 16th International Conference on Rewriting Techniques and Applications (RTA '05)*, volume 3467 of *LNCS*, pages 120–134. Springer, 2005.

[7] T. Aoto and Y. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the 7th International Symposium on Frontiers of Combining Systems (FroCoS '09)*, volume 5749 of *LNAI*, pages 117–132. Springer, 2009.

[8] T. Arts and J. Giesl. Automatically proving termination where simplification orderings fail. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '97)*, volume 1214 of *LNCS*, pages 261–272. Springer, 1997.

[9] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[10] F. Baader and F. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[11] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[12] J. Bergstra and J.W. Klop. Algebra of communicating processes. *Theoretical Computer Science*, 37(1):171–199, 1985.

[13] F. Blanqui. Termination and confluence of higher-order rewrite systems. In L. Bachmair, editor, *Proceedings of the 11th International Conference on Rewriting Techniques and Applications (RTA '00)*, volume 1833 of *LNCS*, pages 47–61. Springer, 2000.

[14] F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA '04)*, volume 3091 of *LNCS*, pages 24–39. Springer, 2004.

[15] F. Blanqui. Inductive types in the Calculus of Algebraic Constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005.

[16] F. Blanqui. Higher-order dependency pairs. In A. Geser and H. Søndergaard, editors, *Proceedings of the 8th International Workshop on Termination (WST '06)*, pages 22–26, 2006.

[17] F. Blanqui. Computability closure: Ten years later. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of his 60th Birthday*, volume 4600 of *LNCS*, pages 68–88. Springer, 2007. Festschrift.

[18] F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.

[19] F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In M. Kaminski and S. Martini, editors, *Proceedings of the 17th EACSL Annual Conference on Computer Science Logic (CSL '08)*, volume 5213 of *LNCS*, pages 1–14. Springer, 2008.

[20] F. Blanqui and C. Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In M. Hermann and A. Voronkov, editors, *Proceedings of the 13th International Conference on Logic for Programming, Artifical Intelligence and Reasoning (LPAR '06)*, volume 4246 of *LNAI*, pages 105–119. Springer, 2006.

[21] F. Blanqui and C. Roux. On the relation between sized-types based termination and semantic labelling. In E. Grädel and R. Kahle, editors, *Proceedings of the 23rd International Workshop on Computer Science Logic and the 18th Annual Conference of the EACSL (CSL/EACSL '09)*, volume 5771 of *LNCS*, pages 147–162. Springer, 2009.

[22] M. Bofill, C. Borralleras, E. Rodríguez-Carbonell, and A. Rubio. The recursive path and polynomial ordering for first-order and higher-order terms. *Journal of Logic and Computation*, 23(1):263–305, 2012.

[23] C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *Journal of Automated Reasoning*, 48(1):107–131, 2012.

[24] C. Borralleras and A. Rubio. A monotonic higher-order semantic path ordering. In R. Nieuwenhuis and A. Voronkov, editors, *Proceedings of the 8th International Conference on Logic for Programming, Artifical Intelligence and Reasoning (LPAR '01)*, volume 2250 of *LNAI*, pages 531–547. Springer, 2001.

[25] C. Borralleras and A. Rubio. Orderings and constraints: Theory and practice of proving termination. In H. Comon-Lundh, C. Kirchner, and H. Kirchner, editors, *Rewriting, Computation and Proof, Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of his 60th Birthday*, volume 4600 of *LNCS*, pages 28–43. Springer, 2007. Festschrift.

[26] N. Çağman and J.R. Hindley. Combinatory weak reduction in lambda-calculus. *Theoretical Computer Science*, 198(1-2):239–247, 1998.

[27] M. Codish, J. Giesl, P. Schneider-Kamp, and R. Thiemann. SAT solving for termination proofs with recursive path orders and dependency pairs. *Journal of Automated Reasoning*, 49(1):53–93, 2012.

[28] Community. The international SAT competitions web page. http://www.satcompetition.org/.

[29] R. Cosmo and D. Kesner. A confluent reduction for the extensional typed λ-calculus with pairs, sums, recursion and terminal object. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (ICALP '93)*, volume 700 of *LNCS*, pages 645–656. Springer, 1993.

[30] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[31] N. Dershowitz. Termination by abstraction. In B. Dernoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming (ICLP '04)*, volume 3132 of *LNCS*, pages 1–18. Springer, 2004.

[32] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004. See also http://minisat.se/.

[33] F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In B. Gramlich, D. Miller, and U. Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning (IJCAR '12)*, volume 7364 of *LNAI*, pages 225–240. Springer, 2012.

[34] J. Endrullis. Jambox – a first-order termination tool. http://joerg.endrullis.de/.

[35] J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[36] M. Ferreira and H. Zantema. Syntactical analysis of total termination. In G. Levi and M. Rodríguez-Artalejo, editors, *Proceedings of the 4th International Conference on Algebraic and Logic Programming (ALP '94)*, volume 850 of *LNCS*, pages 204–222. Springer, 1994.

[37] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT '07)*, volume 4501 of *LNCS*, pages 340–354. Springer, 2007.

[38] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In A. Voronkov, editor, *Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA '08)*, volume 5117 of *LNCS*, pages 110–125. Springer, 2008.

[39] C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. Proving termination by dependency pairs and inductive theorem proving. *Journal of Automated Reasoning*, 47(2):133–160, 2011.

[40] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In R. Treinen, editor, *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA '09)*, volume 5595 of *LNCS*, pages 32–47. Springer, 2009.

[41] C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Proceedings of the 8th International Symposium on Frontiers of Combining Systems (FroCoS '11)*, volume 6989 of *LNAI*, pages 147–162. Springer, 2011.

[42] C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In A. Tiwari, editor, *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications (RTA '12)*, volume 15 of *LIPIcs*, pages 176–192. Dagstuhl, 2012.

[43] J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.

[44] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):7:1–7:39, 2011.

[45] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNAI*, pages 281–286. Springer, 2006.

[46] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR '04)*, volume 3452 of *LNAI*, pages 301–331. Springer, 2005.

[47] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In B. Gramlich, editor, *Proceedings of the 5th International Symposium on Frontiers of Combining Systems (FroCoS '05)*, volume 3717 of *LNAI*, pages 216–231. Springer, 2005.

[48] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[49] J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving termination by bounded increase. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction (CADE '07)*, volume 4603 of *LNAI*, pages 443–459. Springer, 2007.

[50] E. Giménez. Structural recursive definitions in type theory. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP '98)*, volume 1443 of *LNCS*, pages 397–408. Springer, 1998.

[51] B. Gramlich. Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24(1-2):3–23, 1995.

[52] M. Hamana. Higher-order semantic labelling for inductive datatype systems. In M. Leuschel and A. Podelski, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and practice of declarative programming (PPDP '07)*, pages 97–108. ACM, 2007.

[53] N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA '04)*, volume 3091 of *LNCS*, pages 249–268. Springer, 2004.

[54] N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

[55] N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

[56] N. Hirokawa, A. Middeldorp, and H. Zankl. Uncurrying for termination. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artifical Intelligence and Reasoning (LPAR '08)*, volume 5330 of *LNAI*, pages 667–681. Springer, 2008.

[57] H. Hong and D. Jakuš. Testing positiveness of polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.

[58] X. Hongwei and F. Pfenning. Dependent types in practical programming. In A. Appel and A. Aiken, editors, *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '99)*, pages 214–227. ACM, 1999.

[59] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In H. Boehm and G. Steele, editors, *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '96)*, pages 410–423. ACM, 1996.

[60] J. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In G. Kahn, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 350–361. IEEE, 1991.

[61] J. Jouannaud and M. Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997.

[62] J. Jouannaud and A. Rubio. A recursive path ordering for higher-order terms in η-long β-normal form. In H. Ganzinger, editor, *Proceedings of the 7th International Conference on Rewriting Techniques and Applications (RTA '96)*, volume 1103 of *LNCS*, pages 108–122. Springer, 1996.

[63] J. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS '99)*, IEEE, pages 402–411, 1999.

[64] J. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *Journal of the ACM*, 54(1):1–48, 2007.

[65] S. Kahrs. Confluence of curried term-rewriting systems. *Journal of Symbolic Computation*, 19(6):601–623, 1995.

[66] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, 1980.

[67] R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Comparing curried and uncurried rewriting. *Journal of Symbolic Computation*, 21(1):15–39, 1996.

[68] Z. Khasidashvili. Expression Reduction Systems. In K. Rukhaia, editor, *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, 1990.

[69] Z. Khasidashvili and V. van Oostrom. Context-sensitive Conditional Expression Reduction Systems. *Electronic Notes in Theoretical Computer Science*, 2:167–176, 1995.

[70] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, CWI Netherlands, 1980.

[71] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279 – 308, 1993.

[72] J.W. Klop, V. van Oostrom, and R. de Vrijer. Iterative path orders. Unpublished Manuscript, Utrecht University and VU University Amsterdam, 2004.

[73] J.W. Klop, V. van Oostrom, and R. de Vrijer. Iterative lexicographic path orders. In K. Futatsugi, J. Jouannaud, and J. Meseguer, editors, *Essays dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*, volume 4060 of *LNCS*, pages 541–554. Springer, 2006. Festschrift.

[74] C. Kop. WANDA – a higher-order termination tool. http://wandahot.sourceforge.net/.

[75] C. Kop. Simplifying algebraic functional systems. In F. Winkler, editor, *Proceedings of the 4th International Conference on Algebraic Informatics (CAI '11)*, volume 6742 of *LNCS*, pages 201–215. Springer, 2011.

[76] C. Kop and F. van Raamsdonk. A higher-order iterative path ordering. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Proceedings of the 15th International Conference on Logic for Programming, Artifical Intelligence and Reasoning (LPAR '08)*, volume 5330 of *LNAI*, pages 697–711. Springer, 2008.

[77] C. Kop and F. van Raamsdonk. An iterative path ordering. In J.W. Klop, V. van Oostrom, and F. Raamsdonk, editors, *Liber Amicorum for Roel de Vrijer's 60th Birthday*, pages 145–153, 2009. Festschrift.

[78] C. Kop and F. van Raamsdonk. Higher-order dependency pairs with argument filterings. In *Proceedings of the 11th Workshop on Termination (WST '10)*, 2010. Available at http://hdl.handle.net/1871/33234.

[79] C. Kop and F. van Raamsdonk. Higher order dependency pairs for algebraic functional systems. In M. Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11)*, volume 10 of *LIPIcs*, pages 203–218. Dagstuhl, 2011.

[80] C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012. Special Issue for RTA '11.

[81] A. Koprowski. Coq formalization of the higher-order recursive path ordering. *Applicable Algebra in Engineering, Communication and Computing*, 20(5):379–425, 2009.

[82] A. Koprowski and J. Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybernetica*, 19(2):357–392, 2009.

[83] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In R. Treinen, editor, *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA '09)*, volume 5595 of *LNCS*, pages 295–304. Springer, 2009.

[84] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, and J. Giesl. Termination of Isabelle functions via termination of rewriting. In M. Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP '11)*, volume 6898 of *LNCS*, pages 152–167. Springer, 2011.

[85] M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In B. Orchard, C. Yang, and M. Ali, editors, *Proceedings of the 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE '04)*, volume 3029 of *LNAI*, pages 827–837. Springer, 2004.

[86] K. Kusakari. On proving termination of term rewriting systems with higher-order variables. *IPSJ Transactions on Programming*, 42(SIG 7 PRO11):35–45, 2001.

[87] K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.

[88] K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In G. Nadathur, editor, *Proceedings of the 1st International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '99)*, volume 1702 of *LNCS*, pages 47–61. Springer, 1999.

[89] K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 18(5):407–431, 2007.

[90] K. Kusakari and M. Sakai. Static dependency pair method for simply-typed term rewriting and related techniques. *IEICE Transactions*, 92-D(2):235–247, 2009.

[91] C. Laneve. *Optimality and Concurrency in Interaction Systems*. PhD thesis, Università di Pisa, 1993.

[92] D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, 1979.

[93] J. Lévy. *Reductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université de Paris, 1978.

[94] C. Loría-Saénz. *A Theoretical Framework for Reasoning about Program Construction based on Extensions of Rewrite Systems*. PhD thesis, Fachbereich Informatik der Universität Kaiserslautern, 1993.

[95] C. Loría-Saénz and J. Steinbach. Termination of combined (rewrite and λ-calculus) systems. In M. Rusinowitch and J. Rémy, editors, *Proceedings of the Third International Workshop on Conditional Term Rewriting Systems (CTRS '92)*, volume 656 of *LNCS*, pages 143–147. Springer, 1993.

[96] S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO - Theoretical Informatics and Applications*, 39(3):547–586, 2005.

[97] O. Lysne and J. Piris. A termination ordering for higher order rewrite systems. In J. Hsiang, editor, *Proceedings of the 6th International Conference on Rewriting Techniques and Applications (RTA '95)*, volume 914 of *LNCS*, pages 26–40. Springer, 1995.

[98] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192(1):3–29, 1998.

[99] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[100] F. Moller. Infinite results. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*, pages 195–216. Springer, 1996.

[101] T. Nipkow. Higher-order critical pairs. In G. Kahn, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 342–349. IEEE, 1991.

[102] V. van Oostrom. *Confluence for abstract and higher-order rewriting*. PhD thesis, VU University Amsterdam, 1994.

[103] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA '10)*, volume 6 of *LIPIcs*, pages 259–276. Dagstuhl, 2010.

[104] J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996.

[105] J.C. van de Pol and H. Schwichtenberg. Strict functionals for termination proofs. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA '95)*, volume 902 of *LNCS*, pages 350–364. Springer, 1995.

[106] F. van Raamsdonk. On termination of higher-order rewriting. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA '01)*, volume 2051 of *LNCS*, pages 261–275. Springer, 2001.

[107] K. Rose. *Combinatory Reduction Systems with Extensions*. PhD thesis, University of Copenhagen, 1996.

[108] K. Rose. CRSX - combinatory reduction systems with extensions. In M. Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11)*, volume 10 of *LIPIcs*, pages 81–90. Dagstuhl, 2011.

[109] C. Roux. Refinement types as higher-order dependency pairs. In M. Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11)*, volume 10 of *LIPIcs*, pages 299–312. Dagstuhl, 2011.

[110] C. Roux. *Terminaison à base de tailles: Sémantique et généralisations*. PhD thesis, Université Henri Poincairé – Nancy 1, 2012.

[111] M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.

[112] M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.

[113] C. Sternagel and R. Thiemann. Generalized and formalized uncurrying. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Proceedings of the 8th International Symposium on Frontiers of Combining Systems (FroCoS '11)*, volume 6989 of *LNAI*, pages 243–258. Springer, 2011.

[114] S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Transactions on Programming*, 4(2):1–12, 2011.

[115] W. Tait. Intensional interpretation of functionals of finite type. *Journal of Symbolic Logic*, 32(2):187–199, 1967.

[116] M. Takahashi. λ-calculi with conditional rules. In M. Bezem and J.F. Groote, editors, *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, volume 664 of *LNCS*, pages 306–317. Springer, 1993.

[117] V. Tannen and G.H. Gallier. Polymorphic rewriting conserves algebraic strong normalization. *Theoretical Computer Science*, 83(1):3–28, 1991.

[118] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[119] R. Thiemann. *The DP framework for proving termination of term rewriting*. PhD thesis, RWTH Aachen, 2007. Available as Technical Report AIB-2007-17.

[120] R. Thiemann, G. Allais, and J. Nagele. On the formalization of termination techniques based on multiset orderings. In A. Tiwari, editor, *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications (RTA '12)*, volume 15 of *LIPIcs*, pages 339–354. Dagstuhl, 2012.

[121] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs '09)*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.

[122] G. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125. Nauka, Leningrad, 1968. Reprinted in Siekmann, J. and Wrightson, G. (editors), *Automation of Reasoning*, 2:466–483, 1983.

[123] X. Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, 32(4):315–355, 2004.

[124] I. Wehrman and A. Stump. Mining propositional simplification proofs for small validating clauses. In A. Armando and A. Cimatti, editors, *Proceedings of the 3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *ENTCS*, pages 79–91. Elsevier, 2006.

[125] Wiki. Termination Portal. http://www.termination-portal.org/wiki/Termination_Competition.

[126] Wiki. Termination Problem DataBase (tpdb). http://termination-portal.org/wiki/TPDB.

[127] D. Wolfram. *The Clausal Theory of Types*, volume 21 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[128] H. Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.

[129] T. Yamada. Confluence and termination of simply typed term rewriting systems. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA '01)*, volume 2051 of *LNCS*, pages 338–352. Springer, 2001.

Index

Contributions of the Thesis

This thesis contains several contributions to the field of higher-order termination, both published results (a list of which is supplied in the introduction) and yet unpublished ones (in particular the results of Chapters 3 and 5, and parts of the other chapters). The contributions can be summarised as follows:

- building on Blanqui's IDTS formalism [13], the introduction of the *AFSM formalism* (Chapter 2.2), and the notion of weak and strong *reduction pairs* for this class (Chapter 2.4);
- a first (to my knowledge) definition of *rule removal* for higher-order term rewriting (Chapter 2.4);
- *transformations* on AFSMs, which allow termination techniques to make certain assumptions: η-expanding rules, changing arities, flatting meta-variable applications, changing type (Chapter 2.3, 2.4);
- *transformations* from many common formalisms of higher-order rewriting to AFSMs, which make it possible to immediately extend termination results on AFSMs to many other systems (Chapter 3);
- building on van de Pol's weakly monotonic algebras [104], a transformation of the core method to the AFSM formalism, and a definition of the class of *higher-order polynomials* (Chapter 4);
- building on the original definition of the higher-order recursive path ordering [63] and the first-order iterative path ordering, a definition of a *higher-order iterative path ordering* (Chapter 5.2, 5.3, 5.5);
- a recursive variation of the iterative path ordering, *StarHorpo*, which strictly extends the higher-order recursive path ordering of [63] and is incomparable with the existing computability path ordering [19] (Chapter 5.4, 5.5);
- generalising on argument filterings [88, 114], the notion of *argument functions*, both as part of a recursive path ordering transformation and in a more general form in a dependency pair approach (Chapters 5.6, 6.6);
- building on a basic dependency pair definition for HRSs [112], a *dynamic dependency pair approach* for AFSMs, including the new features of meta-

variable conditions, reduction triples and type changing, as well as an alternative definiton of the subterm property, and a systematic way to find reduction pairs which satisfy dependency pair constraints (Chapter 6.3, 6.6);

- inspired by the idea of usable rules [48, 53], the notion of *formative rules* which, though based on a type reasoning, may also be usable for untyped first-order termination analysis (Chapter 6.4);
- a completely new definition of tagged dependency chains and results which allow the subterm property to be significantly weakened, which makes e.g. argument filterings possible for dynamic dependency pairs (Chapter 6.5);
- building on the first-order *dependency pair framework* [46], a first extension of this framework to a higher-order setting (Chapter 7.2);
- a number of transformations of collapsing dependency pairs (Chapter 7.3);
- building both on first-order definitions [9, 48, 53] and higher-order definitions for a static dependency pair approach [87, 114], adaptations of the notions of a dependency graph, subterm criterion and usable rules for the AFSM formalism and the new style of dependency pairs (Chapter 7.4–7.6);
- using a reasoning due to Gramlich [51], two results which make it possible to use termination of the first-order part of a higher-order term rewriting system (as a TRS on terms without λ-abstraction and application!) to eliminate dependency pairs (Chapter 7.7);
- a translation of the *static dependency pairs* from [87] to the AFSM setting, which makes it possible to use formative rules and meta-variable conditions with the static approach pairs, and gives a completeness result for this approach for the class of strong plain function passing systems (Chapter 7.8);
- some basic techniques for automatic non-termination analysis, in particular the detection of systems with which the untyped λ-calculus might (to some extent) be simulated (Chapter 8.2);
- a new approach for calculating the dependency graph, with particular attention to the treatment of bound variables (Chapter 8.4);
- building on first-order methods for automatic polynomial interpretations (see e.g. [37, 57]), techniques to automatically find polynomial interpretations suitable for a given set of constraints (Chapter 8.5);
- building on first-order methods for the recursive path ordering (see e.g. [27]), techniques to automatically find the components of the higher-order path ordering of Chapter 5 (Chapter 8.6);
- the implementation of almost all techniques in this thesis, which provides a powerful and fully automatic higher-order termination tool, WANDA [74].

Titles in the IPA Dissertation Series since 2006

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*.

Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of

Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development*. Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14