Bachelor thesis
Computing Science

Radboud University

The Impact Of Sorts On Non-Termination Analysis Of Term Rewriting Systems

Author:
Bob Ruiken
s4721306

First supervisor/assessor:
dr. C.L.M. (Cynthia) Kop
c.kop@cs.ru.nl

Second assessor:
prof. dr. H. (Hans) Zantema
h.zantema@cs.ru.nl

January 16, 2020

Abstract

Many techniques for detecting non-termination in term rewriting systems exist. None of these techniques, however, are adapted to work for many-sorted term rewriting systems. In this paper the unfolding technique is adapted to be able to work with those systems and implemented into a new tool called Mara (MAny-sorted Rewriting Analyser). We will see that we do not need many large changes to make the existing technique work. We have also conducted a number of experiments with variations and limitations to the unfolding technique. From these experiments we were even able to detect non-termination in an example where no other tool has.

Contents

Chapter 1

Introduction

Term rewriting systems are simple yet powerful tools used in for example definitions of functional languages and other calculi. The rewriting systems have been around for decades and most definitely will be around for many more. They are still actively researched.

The basic notion of term rewriting systems can be seen as a set of rules. These rules can be used to "translate" terms into other terms. Terms, in their place, consist of functions and variables. An example could be as follows.

$$\begin{aligned} r_1 &: \quad f(x) \quad \rightarrow \quad g(x) \\ r_2 &: \quad\quad 0 \quad \rightarrow \quad 1 \end{aligned}$$

Here, f, g, 0 and 1 are functions, and x is a variable. These rules define how terms can be rewritten. If we were to have a term such as $g(f(1))$, we can use r_1 to rewrite the term to $g(g(1))$. We can do this because in r_1 the argument of f is a variable. So if we set x to be 1, then we should rewrite $f(1)$ to $g(1)$. Which is exactly what we did with $g(f(1))$. While 0 and 1 might not seem like functions, they are. We can also write them down as 0() and 1(). They are functions with no arguments, so they can be seen as constants. When using constants we usually omit the parenthesis.

One of these many branches of research focuses on termination and non-termination of term rewriting systems. Termination and non-termination are actively researched since they are properties we want to know about: we can for example translate (functional) languages into term rewriting systems to then look at the termination properties of that code. Even though we know that (non-)termination of programs is generally undecidable, we are increasing the class of programs of which we *can* prove those properties further and further.

Previous research focused on unsorted term rewriting systems, where variables and functions do not have specific type definitions. In many-sorted term rewriting systems, variables and functions do have specific type definitions, specifically first-order types.

For example, without types we could create a term such as $add(5,\ \text{red})$. This can be done since we have never said what type of input the *add* function gets. We of course know that 5 is a number and red is a colour, but by default rewriting systems do not. By adding types, we can define that the *add* function gets two numbers as arguments.

In previous work, many techniques for detecting (non-)termination have been created. These techniques are not made for (many-)sorted terms, and therefore will not work for them without adaptations. Certain steps in the techniques or algorithms might be illegal when working in a sorted rewriting system, since they do not account for types by design. Think for example about the *add* function. Some technique might create a term such as $add(4, blue)$, but we do not want those anymore.

These adaptations, even though they might only be small, can have impact on the strength of the technique. For example, a technique that focuses on creating as many rules as possible might create much fewer rules when they have type bounds. This could be an improvement to the technique but it might also result in a decrease of its effectiveness. With fewer rules to analyse the analyser runs faster but gets fewer opportunities to find non-termination in a system. It is also worth mentioning that some term rewriting systems are non-terminating without types and terminating *with* types.

We will take a look at a number of techniques to prove non-termination and adapt them to make them work for many-sorted rewriting systems. The adaptations will be implemented in an analyser in Java called Mara. Mara stands for MAny-sorted Rewriting Analyser. As a basis for parsing and basic functionality Cora will be used. Cora is an open source tool and stands for COnstrained Rewriting Analyser. Cora focuses on higher order term rewriting but still can be used as a basis for Mara since it also contains parsing and typing for many-sorted terms.

In chapter 2 the necessary preliminaries will be given. Chapter 3 gives some more information on the analysis methods that will be implemented. Then chapters 4 through 7 are about transforming a number of existing techniques to make them work in the many-sorted rewriting setting. These are matching and unification in chapter 4. Then semi-unification, concrete unfolding and abstract unfolding for the chapters 5 through 7 respectively. Experiments are given in chapter 8. These experiments are mainly to as-

sess the strength of different techniques, but we can also compare it to the existing tools in competitions. Then chapter 9 gives some background information on existing techniques and tools. Finally chapter 10 concludes the work and experiments and chapter 11 covers future work.

Chapter 2

Preliminaries

2.1 Unsorted Term Rewriting Systems

Before we can talk about the non-termination of term rewriting systems, we first need to know what exactly they are. We will consider two types of rewriting systems: the "normal" Term Rewriting Systems and secondly Many-Sorted Term Rewriting Systems. They are both needed in order to be able to alter existing non-termination techniques (for the "normal" systems) into the many-sorted case. Note that this section is called "Unsorted Term Rewriting Systems", this is done for clarity, in the literature these are usually called "Term Rewriting Systems". We will start off with the definition of the ordinary term rewriting system. For more details, see for example [18].

2.1.1 Alphabet

Intuitively, a term rewriting system (TRS) is a set of rewriting rules that can be applied on terms, which in its place can be seen as either variables or functions with terms as arguments. More formally it can be seen as a tuple containing an alphabet (Σ) and a set of rewriting rules (R). Σ typically contains an infinite set of variables ($\mathcal{V}$) combined with a set of function symbols ($\mathcal{F}$). Every function symbol has an arity, which is the number of arguments it takes. The arity may be zero, in which case it can also be seen as a constant term.

Definition 2.1. *An alphabet Σ is the union of two disjoint sets $\mathcal{V}$ and $\mathcal{F}$. These are the variables and functions respectively.*

2.1.2 Terms

We define the possible terms in an alphabet Σ as $Ter(\Sigma)$. This set firstly contains the variables in Σ. Secondly, any function in Σ of which all its arguments are in $Ter(\Sigma)$, is also added to the set. More formally:

Definition 2.2. *The set $Ter(\Sigma)$ is inductively defined as follows:*

- *each variable $v \in \mathcal{V}$ is in $Ter(\Sigma)$*
- *if $f \in \Sigma$ has arity n and $s_1, \ldots, s_n$ are in $Ter(\Sigma)$, then $f(s_1, \ldots, s_n) \in Ter(\Sigma)$*

2.1.3 Positions

Positions can be seen as pointers to specific subterms within a term. We define these positions by $Pos(t)$:

Definition 2.3. *for $t \in Ter(\Sigma)$ we define*

$$Pos(t) = \begin{cases} \{\epsilon\}, & \text{if } t \in \mathcal{V} \\ \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in Pos(t_i)\}, & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

We denote $t|_p$ when we want to indicate the position p within term t. The identity case is $t|_\epsilon = t$.

Example 2.1: Suppose we have variables $\{x, y\} \subseteq \mathcal{V}$ and function symbols $\mathcal{F} = \{plus, suc, 0\}$ with arities of 2, 1 and 0 respectively. With these variables and function symbols we can create an alphabet $\Sigma = \mathcal{V} \cup \mathcal{F}$, then we can create terms from $Ter(\Sigma)$. Possible terms include $t_1 = x$, $t_2 = 0$, $t_3 = plus(x, y)$, $t_4 = suc(plus(0, x))$. Positions of these terms are as follows: $Pos(t_1) = \{\epsilon\}$, $Pos(t_2) = Pos(t_1), Pos(t_3) = \{\epsilon, 1.\epsilon, 2.\epsilon\}$ and $Pos(t_4) = \{\epsilon, 1.\epsilon, 1.1.\epsilon, 1.2.\epsilon\}$. We can for example take $t_3|_{2.\epsilon} = y$.

We also denote $t_1\,[t_2]_p$ when position p of term t_1 gets replaced by t_2.

2.1.4 Substitutions

Substitutions are mappings from variables to terms. This means that a substitution applied to a function symbol is equal to a substitution applied to the arguments of the function symbol separately:

Definition 2.4. *A substitution is a mapping* $\sigma = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ *(*$\{v_1, \ldots, v_n\} \subseteq \mathcal{V}$*,* $\{t_1, \ldots, t_n\} \subseteq Ter(\Sigma)$*), which, when applied to a term, replaces all instances in the term of* v_i *with* t_i*.*

We can also write substitutions as $t\sigma$, which is the same as saying $\sigma(t)$.

Example 2.2: *Continuation of example 2.1.* Let us say we want to create a substitution $\sigma_1 = \{x \leftarrow 0, y \leftarrow suc(y)\}$. We can apply this substitution to terms as follows: $\sigma_1(t_1) = \sigma_1(t_2) = 0$, $\sigma_1(t_3) = plus(\sigma_1(x), \sigma_1(y)) = plus(0, suc(y))$ and $\sigma_1(t_4) = suc(\sigma_1(plus(0, x))) = suc(plus(0, \sigma_1(x))) = suc(plus(0, 0))$.

Definition 2.5. *Two given terms* t_1 *and* t_2 *are said to be matching, when there exist a substitution* σ *for which holds that* $\sigma(t_1) = t_2$*.*

2.1.5 Rewrite rules

We can now create a notion for rewrite rules. These rules have a left and right side, the left side cannot be a variable, otherwise the rule could be applied to any term. Also, the right side cannot contain any variables that do not exist on the left side. This is because otherwise, the variable cannot be given a value when applying the rule.

Definition 2.6. *A rewrite rule is defined as* $r_i : l \to r$*. Here,* l *and* r *are terms in* $Ter(\Sigma)$*.* r_i *is an optional name for the rule. Furthermore,* l *cannot be a variable and if* $Var(t)$ *defines the variables in a term* t*,* $Var(r) \subseteq Var(l)$*.*

Example 2.3: *Continuation of example 2.1.* We can now create rules from terms generated by the alphabet Σ:

$$\begin{aligned} r_1 : \quad & plus(x, 0) && \to \quad x \\ r_2 : \quad & plus(x, suc(y)) && \to \quad suc(plus(x, y)) \end{aligned}$$

These rules define simple addition. Examples of rules that are not allowed are:

$$\begin{aligned} r_3 : \quad & x && \to \quad s(x) \\ r_4 : \quad & plus(x, 0) && \to \quad plus(x, y) \end{aligned}$$

The first rule has a variable as a left-hand side, which is disallowed. The second rule has a variable on the right-hand side that is not used in the left-hand side. Apart from valid rules that make sense and rules that are disallowed there also are nonsensical rules:

$$\begin{aligned} r_5 : \quad & plus(x, x) && \to \quad plus(s(x), s(x)) \\ r_6 : \quad & suc(x) && \to \quad suc(0) \end{aligned}$$

These rules are valid, but still do not make a lot of sense.

2.1.6 Rewrite relation

We can now define how these rules can be used to rewrite terms. Intuitively, we can apply a rule to some term when there is a subterm of that term that matches the left side of a given rule. Then the resulting term is the original term where the chosen subterm gets replaced by the right side of the rule.

Definition 2.7. *We denote $t_1 \to_R t_2$ if a term t_1 can be rewritten, using a set of rules R, to a term t_2. A term t_1 can be rewritten using the rule $l \to r \in R$ to t_2, if there exist a position p of t_1 and a substitution σ such that $t_1|_p = \sigma(l)$. Then $t_2 = t_1\,[\sigma(r)]_p$.*

Example 2.4: *Continuation of example 2.3* Suppose we have a term rewrite system with the rules r_1 and r_2 we defined previously. We can represent $1+2$ in a term t as $plus(suc(0), suc(suc(0)))$. We can rewrite t using r_2. We choose position $p = \epsilon$. Then $\sigma = \{x \leftarrow suc(0), y \leftarrow suc(0)\}$. The resulting term then becomes $t\,[\sigma(suc(plus(x, y)))]_\epsilon = t\,[suc(plus(suc(0), suc(0)))]_\epsilon = suc(plus(suc(0), suc(0)))$. We then get:

$$\begin{aligned} & plus(suc(0), suc(suc(0))) \\ \to_R\ & suc(plus(suc(0), suc(0))) \\ \to_R\ & suc(suc(plus(suc(0), 0))) \\ \to_R\ & suc(suc(suc(0))) \end{aligned}$$

2.1.7 Term Rewriting System

Formally, a TRS is a pair $(Ter(\Sigma), \to_R)$ of a set of terms and a relation on that set. We will usually refer to a TRS as the pair (Σ, R) (the alphabet and the set of rewrite rules) generating it, or just the set R, leaving Σ implicit.

2.2 Many-Sorted Term Rewriting System

2.2.1 Many-Sorted or First-Order

Many-Sorted term rewriting systems are much like the previously explained term rewriting systems. The difference is that in many-sorted term rewriting

systems, all elements of the alphabet Σ have types. More precisely, they are first-order typed.

A first-order type is an identifier that puts terms in classes. We define arrow types and base types. Arrow types are used to give types to function symbols. A function symbol f could be typed as $\alpha \to \beta \to \gamma$. We say that γ is the output type of f. We can also see that f has two arguments, the first one must be of type α, the second one of type β. Base types are types that do not contain arrows.

First-order types constrict variables to use base types. Functions must have a base type as their output type and all of their arguments must be first-order typed.

Definition 2.8. *First-order types are defined as follows:*
Arrow types are defined as $\tau_1 \to \cdots \to \tau_n$. *Here,* τ_i *is a base type for* $i \in [1, n]$. *Base types are types without arrow types.*

We could for example have the function *plus* typed as $nat \to nat \to nat$, denoted $plus :: nat \to nat \to nat$. Now *plus* is a function that takes two arguments of type *nat* and gives back a term of type *nat*. Another function *eq* could be typed as follows: $eq :: nat \to nat \to bool$. This function takes two arguments of type *nat* and gives back a term of type *bool*.

2.2.2 Changes to the definitions of unsorted rewriting systems

We do not need to change many of the previous definitions, we just need to make sure that every element of the alphabet gets a type and that all operations defined are not allowed to break the typing. For example, a term of type α may not be substituted with one of type β.

Alphabet

We define the alphabet of a many-sorted TRS as Σ, containing the set of variables $\mathcal{V}$ and the set of function symbols $\mathcal{F}$ as before. Now we extend this by saying that every variable has a base type. The function symbols are extended such that every function gets first-order types as well.

Definition 2.9. *An alphabet* Σ *is the union of two disjoint sets* $\mathcal{V}$ *of typed variables, and* $\mathcal{F}$ *of typed function symbols, satisfying the requirements below:*

- *each variable* $v \in \mathcal{V}$ *has a base type* τ

- *each function* $f \in \mathcal{F}$ *has type* $\tau_1 \to \cdots \to \tau_{n+1}$

We say that a fu;nction symbol of type $\tau_1 \to \cdots \to \tau_{n+1}$ *has arity* n*. Sometimes we refer to variables and functions without the type notation.*

Terms

We redefine the possible terms over an alphabet Σ as $Ter(\Sigma)$ for many-sorted term rewriting systems. This definition is much like definition 2.2 but contains extra typing checks:

Definition 2.10. *The set* $Ter(\Sigma)$ *is inductively defined as follows:*

- *each variable* $v \in \mathcal{V}$ *is in* $Ter(\Sigma)$
- *if* $f :: \alpha_1 \to \cdots \to \alpha_{n+1} \in \Sigma$, $s_1, \ldots, s_n$ *are in* $Ter(\Sigma)$ ***and*** $\boldsymbol{\alpha_i = Type(s_i)}$ ***for*** $\boldsymbol{i \in [1, n]}$ **, *then*** $\boldsymbol{f(s_1, ..., s_n) \in Ter(\Sigma)}$

For each typed variable $v :: \tau \in V$: $Type(v) = \tau$*, for each typed function* $f :: \tau_1 \to \cdots \to \tau_{n+1} \in F$: $Type(f(s_1, \ldots, s_n)) = \tau_{n+1}$.

2.2.3 Substitutions and rewrite rules

As previously explained, we also need to make sure that when using substitutions, both sides of the substitution have the same type. When it comes to rules, it is much like the case in the substitutions, both sides of the rules must have the same type.

Definition 2.11. *A substitution is a mapping* $\sigma = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ *(*$\{v_1, \ldots, v_n\} \subseteq \mathcal{V}$*,* $\{t_1, \ldots, t_n\} \subseteq Ter(\Sigma)$*), which, when applied to a term, replaces all instances in the term of* v_i *with* t_i*.* $Type(t_i) = Type(v_i)$ *for all* $i \in [0, n]$*.*

Definition 2.12. *A rewrite rule in a many-sorted TRS is defined as* r_i : $l \to r$*. Here,* l *and* r *are terms in* $Ter(\Sigma)$*.* r_i *is an optional name for the rule. Furthermore,* l *cannot be a variable and if* $Var(t)$ *defines the variables in a term* t*,* $Var(r) \subseteq Var(l)$ *and* $\boldsymbol{Type}(l) = \boldsymbol{Type}(r)$*.*

Example 2.5: Let us create an alphabet Σ consisting of the variables $\mathcal{V} = \{x, y, z\}$. The types of these variables are as follows: x and z have type *nat* and y has type *bool*. Now we create the set of function symbols $\mathcal{F} = \{eq, suc, true\}$. eq has type $nat \to nat \to bool$, suc has type $nat \to nat$

and *true* is of type *bool*. Substitutions such as $\sigma_1 = \{y \leftarrow true\}$ and $\sigma_2 = \{y \leftarrow eq(suc(x), x)\}$ are possible, but we cannot create a substitution like $\sigma_2 = \{x \leftarrow y\}$, since the type of x (α) does not match the type of y (β).

Example 2.6: *Continuation of Example 2.5.* To create an entire TRS, we also need some rewriting rules. With the alphabet we just created we can create the following rule:

$$\begin{aligned} r_1 : \quad & eq(x, x) && \rightarrow \quad true \\ r_2 : \quad & eq(suc(x), suc(z)) && \rightarrow \quad eq(x, z) \end{aligned}$$

These rules are correctly typed as the output type of *eq* is *bool* and the types of *true* is also *bool*. Also the arguments supplied to *eq* all have type *nat*.

2.2.4 Unifiers

A unifier of two terms is a substitution σ such that, when applied to two (or more) terms, the resulting terms are all equal.

Definition 2.13. *Given two terms t_1 and t_2, a unifier σ is a substitution such that $\sigma(t_1) = \sigma(t_2)$.*

We will omit types in the following example. In further examples we always talk about many-sorted systems, but sometimes types are omitted.

Example 2.6 Let us take two terms $t_1 = f(g(x), y)$ and $t_2 = f(z, g(z))$. To unify these two terms, we can use a unifier $\sigma_1 = \{y \leftarrow g(z), z \leftarrow g(x)\}$.

Most General Unifiers

We will now look at what happens when we apply multiple substitutions on one term. If we have multiple substitutions to apply to t, for example σ_1 and σ_2, we denote it as $t\sigma_1\sigma_2$. If we see the substitution as a function we can write $\sigma_2(\sigma_1(t))$.

Example 2.7: Let us take a term $t = f(x, g(y))$ and substitutions $\sigma_1 = \{f(x, x) \leftarrow x\}$, $\sigma_2 = \{y \leftarrow x, 0 \leftarrow y\}$. If we want to know $t\sigma_1\sigma_2$, or equivalently, $\sigma_2(\sigma_1(t))$, we can first take $t\sigma_1 = f(f(x, x), g(y))$. Then from there we apply σ_2, which brings us to $\sigma_2(f(f(x, x), g(y))) = f(f(x, x), g(0))$.

We can also create a definition for a most general unifier:

Definition 2.14. *Given two terms t_1 and t_2, a most general unifier σ is a substitution such that:*

- *σ unifies t_1 and t_2*
- *given any other unifier σ' that unifies t_1 and t_2, σ' can be created using a substitution ω on σ*

If a unifier for two terms exist, then there also exists a most general unifier[4]. The most general unifier is also unique under variable renaming.

Example 2.8: Take terms $s = f(x, h(y, z))$ and $t = f(z, h(0, x))$, it is clear that we can create a substitution that will unify these terms: $\sigma_1 = \{x \leftarrow z, y \leftarrow 0\}$, then $s\sigma_1 = t\sigma_1$. This substitution is also the most general unifier of these terms. To illustrate this, we can take another substitution $\sigma_2 = \{x \leftarrow f(x, x), z \leftarrow f(x, x), y \leftarrow 0\}$. Note that we can create σ_2 using σ_1: $\sigma_2 = \sigma_1\omega_1$, take $\omega_1 = \{z \leftarrow f(x, x)\}$.

2.3 Cora

Cora, standing for COnstrained Rewriting Analyser[21], will be used as the basis of the non-termination analyser resulting from this paper. Cora is a work in progress analysis tool for various kinds of TRSs. Mara uses Cora mainly as a parser for TRSs. The only usable parts for Mara in Cora is what is defined in section 2.1. Cora is written in Java and uses ANTLR[25] as its parsing tool.

2.3.1 Terms

Perhaps the most important feature we can use in Cora are the predefined Term types. The terms come in two major types: variables and functions. Each implementation of the Term interface has to provide the following (listed are the most interesting ones) functionality:

1. Getting (all possible) subterms,
2. (for non variables) obtaining the function symbol,
3. Equality comparison against other terms,
4. Replacing a position with a term,
5. Substitutions,
6. Matching a term to another term.

All of these functionalities have typing checks within them.

2.3.2 Rules

To create rules for use in rewriting systems, Cora has an interface Rule. The most interesting functionalities are:

1. Getting the left or right side of the rule,
2. Checking if the rule can be applied to some term,
3. Actually applying the rule to some term.

Just as with the terms, all of these actions have typing checks.

2.3.3 Term rewriting system

Within Cora, a term rewriting system is basically just a list of rules. Cora can import term rewriting systems using three file formats: `.mstrs` (Many-Sorted TRSs), `.trs` (untyped TRSs) and `.cora` (an internal file type, also sorted).

Chapter 3

Analysers

Before we can implement the non-termination analysers into Mara, we need a basis to implement them onto. This will of course be done as an extension of Cora. Mara can be found on Github:

`https://www.github.com/BornoBob/Mara`

The analysers we will implement all have the same task and will get as an input only a term rewriting system. The task in our case obviously is checking non-termination. The implementations will only cover many-sorted rewriting systems. The possible results of the analysers come down to four situations: `YES`, `NO`, `MAYBE` and `TIMEOUT`. These results are also used in competitions where term rewriting systems are analysed[10]. Note that even though we ask the question "Does this TRS non-terminate" in the analysis techniques, the results answer the question "Does this TRS terminate". I.e. when a TRS does not terminate, the analyser will result in `NO`. Also note that after our analysis is done, we cannot just return `YES`, because there is no technique that encompasses all (non-)termination.

Since our analysis will only return `NO`, `MAYBE` and `TIMEOUT`, the analyser will use `NONTERMINATES` instead of `NO` for clarity.

Chapter 4

Matching and Unification

The first non-termination detection techniques we will look at are matching and unification. These two techniques have the same core idea, namely to check for all the rules if the left side matches or unifies with any non-variable position on the right side. We will see that if this is the case, we can already conclude non-termination of the rewriting system.

4.1 Matching

Unlike unification, we have not looked at matching yet. Matching can be seen as a more restricted form of unification. A term t_1 matches with another term t_2, if there exists a substitution σ_1 such that $\sigma_1(t_1) = t_2$. Instead of applying the substitution to both terms, it only is applied to one.

Matching can be used to detect non-termination, if for some rule, the left side matches with a non variable subterm on the right, we know it is non-terminating. This is because we can repeatedly apply the rule on the chosen subterm.

Example 4.1: Consider the rule $r_1 : f(x) \to g(1, f(h(x)))$. We can take the subterm $t_1 = f(h(x))$ and check if there is a σ_1 for which holds that $\sigma_1(f(x)) = f(h(x))$. Choose $\sigma_1 = \{h(x) \leftarrow x\}$ and this holds. The infinite reduction then looks like (for example) $f(1) \to_R g(1, f(h(1))) \to_R g(1, g(1, f(h(h(1))))) \to_R \ldots$. So we conclude that r_1 is non-terminating.

Cora already has matching built into it since it is also required to check if a rule can be applied to a term. Note that when applying a rule, you match the left hand side of the rule with the term you want to apply it on,

and finally apply the resulting substitution on the right hand side of the rule.

The algorithm to match two terms is given in Algorithm 1 on page 17. This is the algorithm as implemented in Cora. Note that this algorithm does not mention any types. The type checks are embedded in comparisons between terms. Also note that when recursively calling `match` on line 19, σ is not initialised to $\emptyset$ in the new call. The given substitution from the recursive call is used instead.

As noted above, the matching algorithm was already built into Cora, but it was not implemented as an analyser yet. In Mara it is, called the `MatchingAnalyser`.

Algorithm 1 Matching

```
 1: Input terms t1 and t2, initialise σ = ∅
 2: if t1 is a variable then
 3:   if σ contains t1 then
 4:     if σ(t1) ≠ t2 then
 5:       throw Failure
 6:     end if
 7:   else
 8:     σ(t1) := t2
 9:     return σ
10:   end if
11: else if t1 is in the form of f(x1, ..., xn), n being the arity of f then
12:   if t2 is a variable then
13:     throw Failure
14:   else if t2 is in the form of g(y1, ..., ym), m being the arity of g then
15:     if f ≠ g then
16:       throw Failure
17:     end if
18:     for all i ∈ [1, n] do
19:       match xi with yi using σ
20:       return σ
21:     end for
22:   end if
23: end if
```

4.2 Unification

Similarly to matching, we can also apply unification to detect non-termination of term rewriting systems. Recall that unification of two terms t_1 and t_2 is possible when we can find a σ such that $\sigma(t_1) = \sigma(t_2)$. We know that if the left hand side of a rule unifies with some non variable subterm of the right hand side, the TRS is non-terminating, for the following reason: suppose that we have a rule $r_1 : l \to r$, and that there is some non-variable subterm r' of r and that there is a substitution σ such that $\sigma(l) = \sigma(r')$. We can now create an infinite loop: apply r_1 to $\sigma(l)$, then apply r_1 to the result of that and repeat this process....

Example 4.2: Take the rule $r_1 : f(g(x), z, y) \to f(y, g(x), y)$. To try to unify the left side with some non-variable subterm on the right, the most sensible subterm to choose is the entire right side. We can unify these two sides by taking $\sigma = \{g(x) \leftarrow y, g(x) \leftarrow z\}$. If we apply r_1 to $\sigma(f(g(x), z, y)) = f(g(x), g(x), g(x))$, we get $f(g(x), g(x), g(x))$ back, which can immediately be applied to r_1 again; this clearly keeps looping indefinitely.

Implementing the unification algorithm that works for many-sorted terms does not require many adaptations from an existing algorithm such as the one from Martelli and Montanari.[23] The algorithm in pseudocode can be found in Algorithm 2 on page 19. Unlike earlier, type checks are included, but they are not explicitly noted in the comparison between terms and function symbols.

This algorithm is implemented and can be used in an analyser. The only thing the analyser needs to do is check for every rule, whether the left side unifies with some non-variable subterm of the right hand side. This is implemented in the `UnificationAnalyser`.

Algorithm 2 Unification

```
1: Input terms t_1 and t_2, initialise σ = ∅
2: if t_1 is a variable then
3:    if t_2 is a variable then
4:       if t_1 ≠ t_2 then
5:          if Type of t_1 ≠ type of t_2 then
6:             throw Failure
7:          else
8:             σ(t_2) := t_1
9:          end if
10:      end if
11:      return σ
12:   else
13:      return unify t_2, t_1
14:   end if
15: else if t_1 is in the form of f(s_1, ..., s_n), n being the arity of f then
16:   if t_2 is a variable then
17:      if t_1 contains t_2 or type of t_1 ≠ type of t_2 then
18:         throw Failure
19:      else
20:         σ(t_2) := t_1
21:         return σ
22:      end if
23:   else if t_2 is in the form of g(u_1, ..., u_m), m being the arity of g then
24:      if f ≠ g then
25:         throw Failure
26:      end if
27:      for all i ∈ [1, n] do
28:         γ := unify σ(s_i) with σ(u_i)
29:         for all Variable v in domain of γ do
30:            if v in domain of σ and σ(v) ≠ γ(v) then
31:               throw Failure
32:            else
33:               σ(v) := γ(v)
34:            end if
35:         end for
36:      end for
37:      return σ
38:   end if
39: end if
```

Chapter 5

Semi-Unification

Semi-unification is yet another way to detect non-termination. As its name suggests, it is somewhat like unification. It is actually a technique that encompasses both matching and unification. If we take two terms t_1 and t_2, matching creates a substitution ρ such that $\rho(t_1) = t_2$ and unification creates a substitution σ such that $\sigma(t_1) = \sigma(t_2)$. If we combine these, we get semi-unification:

Definition 5.1. *Given two terms t_1 and t_2, t_1 and t_2 semi-unify if we can create two substitutions ρ and σ such that $\rho(\sigma(t_1)) = \sigma(t_2)$.*

It is clear that this encompasses both matching (take $\sigma = \emptyset$) and unification (take $\rho = \emptyset$).

An algorithm to compute semi-unification for two input terms is given in for example Kapur et al[16]. In the same paper a proof is also given how semi-unification can detect non-termination:

The idea is the same as with matching and unification, we will try to semi-unify the left hand side of a rule with some non-variable subterm on the right hand side. Suppose we have some rule $r_1 : l \to r$. And r' is a non-variable subterm of r. Now say that semi-unification is successful on r', such that $\rho(\sigma(l)) = \sigma(r')$. If we were to apply both the substitutions on the rule r_1 we would get $r_1' : \rho(\sigma(l)) \to \rho(\sigma(r))$, for which we know there is some subterm $\rho(\sigma(r'))$ of $\rho(\sigma(r))$. This subterm is in its place equal to $\rho(\rho(\sigma(l)))$, because of the substitution we started with. Now we can rewrite this term to $\rho(\rho(\sigma(r)))$ and further repeat the last two steps indefinitely.

The notion of semi-unification above is actually called left-unification. Right-unification can be calculated using an algorithm for left-unification

if we swap the two input terms, so we would get $\sigma(t_1) = \rho(\sigma(t_2))$. Semi-unification captures both left- and right-unification. In the case for non-termination we use left-unification. This is also what we refer to when further speaking of semi-unification.

Semi-unification can detect non-termination of term rewriting rules, but also has other use cases, for example in type inference in Milner-Mycroft calculus[13]. This calculus is used in some polymorphic functional programming languages. In the general case, semi-unification is undecidable[17]. In our case we are dealing with two input terms, for which semi-unification *is* decidable.

We will implement the algorithm described by Kapur et al for many-sorted term rewriting systems. The implementations are two separate algorithms. The first algorithm is sufficient to decide semi-unifiability, and as a result will generate a set of rewriting rules. But to obtain the substitutions, we need the second algorithm. It takes as an input the result from the first algorithm, then extracts the solution from it.

5.1 Algorithm 1

In the algorithm we will be looking for two substitutions σ and ρ, such that for two terms t_1 and t_2, $\rho(\sigma(t_1)) = \sigma(t_2)$. We have a special symbol s_x for each variable x. s_x represents $\sigma(x)$. This mapping is captured in a substitution θ, so we have for each variable x in t_1 and t_2: $\theta(x) = s_x$. Note that $\theta = \sigma$, but we are able to create the inverse of θ using the extra elements s_x. This is because these elements make it so that the mapping *and* the inverse of the mapping both are bijective.

Before we can implement the algorithm itself, we need two notions of rewriting that are used within it. These are a notion of distributivity and a notion of cancellativity.

5.1.1 Distributivity

Distributivity can be seen as "pushing down" the ρ in a term. We know that ρ is a substitution, so it will only have effect on variables. This means that we can "push down" the ρ symbol in a term to the variable level:

$$\rho(f(x_1, \ldots, x_n)) = f(\rho(x_1), \ldots, \rho(x_n))$$

Note that this notion does not need adaptations for many-sorted terms, as the output type of the ρ function is always the type of its argument.

5.1.2 Cancellativity

Cancellativity of functions can be applied when we have two terms with the same root function symbol. When this is the case, we can cancel out the function symbol and result in n new equations comparing the arguments (for n arguments):

$$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \implies x_1 = y_1, \ldots, x_n = y_n$$

As before, this does not need any adaptations for many-sorted terms, the function symbol on the left is the exact same as on the right, so we do not need any type checks.

5.1.3 Implementation

In Algorithm 3 on page 23 the pseudocode is given for the algorithm. Note that the only places where we need to check for types is in line 6, comparing two function symbols, and when rewriting the rules in line 17. The fact that these checks are added means that the algorithm in some cases might perform better, since there are fewer possible ways to rewrite the rules.

Also note that when rewriting the equations into rules (at line 14/15) there might be situations where the ordering defined is not complete enough. For example, $f(g(x), y) = f(x, g(y))$: both sides have two function symbols, and the variables appear in the order y then x from right to left. In this case we made the choice that the order does not matter. Another problem comes into place comparing terms such as $\rho(s_x)$ and s_x. Strictly speaking ρ is not a function symbol of a TRS, so the terms would be equal. The problem when writing into the rule $s_x \to \rho(s_x)$ is that this rule is non-terminating. This would lead to the algorithm not terminating. To solve this, when the comparison on function symbols does not help, another comparison is done on the number of ρ symbols. The term with the highest number of ρ symbols becomes the left-hand side.

The only ways the algorithm may report failure, is very close to the previously seen unification algorithm. Namely when either we reach a situation where we try to unify two terms with different function symbols, or when we try to unify some variable and non-variable term containing that variable. The latter is referred to as the "occurs" check.

The proof of the full algorithm can be found in the paper[16]. Intuitively, the idea is to show that the only reasons semi-unification can fail is when either of the two "failure" situations arise. If these situations do not occur,

then the first algorithm can be used to obtain the reduced canonical rewrite system. The second algorithm then can be used to extract σ and ρ.

The algorithm will succeed if semi-unification is possible and return failure if this is not the case.

5.1.4 Intuition

The algorithm works by starting with two terms. We keep trying to rewrite the terms to try to find a counterexample of semi-unification. If this counterexample cannot be found, then semi-unification is possible.

Algorithm 3 Semi-Unification (Algorithm 1)

1: Input terms t_1 and t_2
2: Define a total ordering $>_\theta$ on the range of θ (all the s_x terms)
3: Begin with the equation $\rho(\sigma(t_1)) = \sigma(t_2)$
4: Apply distributivity rules on both sides of the equation
5: Apply cancellativity rules on both sides of the equation
6: **for all** Equations resulting from the cancellativity **do**
7: **if** An equation in the form of $f(x_1, \ldots, x_n) = g(y_1, \ldots, y_n)$ is encountered, with $f \neq g$ **or** $n \neq m$ **then**
8: **Throw** Failure
9: **else if** An equation in the form of $\rho^i(s_x) = f(\ldots \rho^{i+j}(s_x) \ldots)$ is encountered, where the right side means that there is some subterm $\rho^{i+j}(s_x)$ in it, with $i, j \in \mathsf{N}$ **then**
10: **Throw** Failure
11: **end if**
12: **end for**
13: {Now we rewrite the equations into rules}:
14: Terms containing function symbols are considered lower than ones that do not.
15: Other terms are compared by their variables (in lexicographic order from right to left) using $>_\theta$.
16: Replace each equation $s = t$ by a rule $s \to t$ with t lower than s.
17: Now, for each rule, try to rewrite both sides using other rules. If this can be done, replace the rewritten rule by the result.
18: If this can be done, go back to to line 4 with this rule as the equation (keep the others).
19: Otherwise, report semi-unifiability, and **return** the resulting rules.

5.1.5 Examples

Now we will look at some examples to see the workings of both of the algorithms described above. The first example we will look at is the rule $r_1 : f(y, g(1, x)) \to f(h(y), g(x, 1))$. The second example is the rule $r_2 : g(f(x, y), h(y, z)) \to g(z, x)$, which is an adaptation of an example from [16].

Example 1

The first example we will look at is $r_1 : f(y, g(1, x)) \to f(h(y), g(x, 1))$, which is an example that is semi-unifiable, typed as follows:

$$\begin{aligned} f &:: a \to b \to c \\ g &:: c \to c \to b \\ h &:: a \to a \end{aligned}$$

We start with two input terms which are the left and right side of the rule: $t_1 = f(y, g(1, x))$ and $t_2 = f(h(y), g(x, 1))$.

We then "apply" the substitutions to get to the form $\rho(\sigma(t_1)) = \sigma(t_2)$, in our case this is $\rho(\sigma(f(y, g(1, x)))) = \sigma(f(h(y), g(x, 1)))$. We can rewrite the σ substitutions here using distribution of σ and the definition of the θ substitution: $\rho(f(s_y, g(1, s_x))) = f(h(s_y), g(s_x, 1))$.

The next step of the algorithm tells us to apply the distributivity rules to both sides of the equation, which in this case is only relevant on the left side, as the right side does not contain any ρ symbols. "Pushing down" ρ, we get: $f(\rho(s_y), g(1, \rho(s_x))) = f(h(s_y), g(s_x, 1))$.

Now that the ρ symbols are on just the variables we can apply the cancellativity rules. This firstly results in two equations: $\rho(s_y) = h(s_y)$ and $g(1, \rho(s_x)) = g(s_x, 1)$. But since we can apply cancellativity on the last equation again, we have to do so. Doing that means we replace it with the following two equations: $1 = s_x$ and $\rho(s_x) = 1$.

We did not reach a situation where the occurs check was positive nor did we find a mismatching function symbol, so we can continue with the algorithm. In the next step we have to create rules from the obtained equations. In this situation we can get away with using the fact that terms containing function symbols should be on the right. This gives us the following three

rules:

$$\begin{aligned} r_{1,1}: \ \rho(s_y) &\to h(s_y) \\ r_{1,2}: \ s_x &\to 1 \\ r_{1,3}: \ \rho(s_x) &\to 1 \end{aligned}$$

Since we can rewrite $r_{1,3}$ using $r_{1,2}$, we are not done yet. Doing this rewriting results in $\rho(1) \to 1$, which we will replace rule $r_{1,3}$ by. But before we can possibly do more rewritings, we need to apply distributivity and cancellativity to them and finally create a rule from it. Distributivity gives us $1 = 1$ and then cancellation gives us an empty set of equations (meaning that we do not add any rule). We now have the following two rules:

$$\begin{aligned} r_{1,1}: \ \rho(s_y) &\to h(s_y) \\ r_{1,2}: \ s_x &\to 1 \end{aligned}$$

There is no way to rewrite this further which means that the first algorithm succeeded with as result $r_{1,1}$ and $r_{1,2}$.

Example 2

Now we show an example where semi-unification fails. We do this with the rule $r_2 : g(f(x,y),h(y,z)) \to g(z,x)$, which is a slight variation of an example taken from [16]. In that paper, there was no typing, so we introduce it as follows:

$$\begin{aligned} f &:: b \to c \to a \\ g &:: a \to b \to b \\ h &:: c \to a \to b \end{aligned}$$

As before, we start with the left and right sides as the two input terms. After introducing the substitutions and pushing down, we get:

$$g(f(\rho(s_x),\rho(s_y)),h(\rho(s_y),\rho(s_z))) = g(s_z,s_x)$$

After applying cancellation, we get two equations: $f(\rho(s_x),\rho(s_y)) = s_z$ and $h(\rho(s_y),\rho(s_z)) = s_x$; rewriting these into rules gives us the following two rules:

$$\begin{aligned} r_{2,1}: \ s_z &\to f(\rho(s_x),\rho(s_y)) \\ r_{2,2}: \ s_x &\to h(\rho(s_y),\rho(s_z)) \end{aligned}$$

We can rewrite $r_{2,1}$ using $r_{2,2}$ and obtain $s_z = f(\rho(h(\rho(s_y),\rho(s_z))),\rho(s_y))$. Now note that after applying distributivity rules, we get:

$$s_z = f(h(\rho(\rho(s_y)),\rho(\rho(s_z))),\rho(s_y))$$

This gives us a problem as the "occurs" check is positive: s_z is contained within the right side. This means that the semi-unification algorithm has failed.

5.2 Algorithm 2

Even though the first algorithm described above suffices to detect semi-unification, we will still implement the second algorithm to extract the substitutions from the resulting rules. We will do this because, even though we are certain the algorithm is correct, it is still a rather complex algorithm in which small mistakes could easily be made. By implementing the second algorithm, we have a double check to ensure that the result is actually correct. If, after applying the second algorithm, the terms do not semi-unify with the substitutions, we can deny semi-unifiability.

5.2.1 Implementation

The algorithm can be found in Algorithm 4 on page 27. Note that here as well, not many changes need to be made in order to make the algorithm work for many-sorted terms. The most important points are in lines 4 and 17, where new fresh variables are created. These variables need to have the correct typing. On line 4, this means that the variable u should have the same type as variable x. On line 17, this means that all the fresh variables need to have the same type as term t. This algorithm is also implemented in Mara as an analyser called `SemiUnificationAnalyser`.

5.2.2 Intuition

The algorithm will return substitutions ρ and σ on an input system returned from the first algorithm. The algorithm has three general steps to it. In the first step (the first for loop), ρ symbols on the right hand sides are replaced by new variables. This replacement is reflected in all the rules. After this step there are no ρ symbols left in any right side of any rule.

The second step takes care of the rules where the left side does not have a ρ symbol. In this case the rule basically describes a substitution of σ: the left side is in the form of s_x, which means $\sigma(x)$. So after we replace the instances of s_y with y on the right side using θ^{-1} we can create a substitution step.

Algorithm 4 Semi-Unification (Algorithm 2)

Input: rules resulting from algorithm 1
Initialise $\rho = \emptyset$, $\sigma = \emptyset$ and $\mathcal{V} = V$
for all Subterms of the form $\rho(s_x)$ in all right hand sides **do**
    Create fresh variable u with $type(u) = type(x)$ and special symbol s_u
    Extend θ with $\theta(u) = s_u$
    $\mathcal{V} = \mathcal{V} \cup \{u\}$
    Extend ρ with $\rho(x) = u$
    Replace $\rho(s_x)$ with s_u in all rules
end for
for all Rules $s_x \to r$ where ρ does not occur in r **do**
    if Right hand side does not contain s_x **then**
        Extend σ with $\sigma(x) = \theta^{-1}(r)$
    end if
end for
for all Rules in which ρ *does* occur **do**
    Rule is in the form $\rho^i(s_x) \to t$ with $i \in \mathsf{Z}^+$
    Introduce fresh variables $u_1, \ldots, u_{i-1}$ all with the type equal to $type(x)$
    Extend ρ with: $\rho(x) = u_1$, $\rho(u_1) = u_2$, $\ldots$, $\rho(u_{i-1}) = \theta^{-1}(t)$
end for
return ρ and σ

The last step of the algorithm takes care of the left sides that *do* contain ρ symbols. Note that this left hand side may contain any number of ρ symbols. For this reason we create a new fresh variable for each symbol. This may lead to unnecessary variables. In the end the idea is the same as in step 2 but instead of extending σ, we extend ρ.

5.2.3 Examples

We will now look at two examples for the second algorithm. The first example is a continuation on the first example of the first algorithm:

Example 1

Because the first algorithm succeeded we can now use the second algorithm to extract the solution from the resulting rules.

The rules we got from the first algorithm are the following two:

$$\begin{aligned} r_{1,1}: \quad \rho(s_y) &\rightarrow h(s_y) \\ r_{1,2}: \quad s_x &\rightarrow 1 \end{aligned}$$

In the first step we have to deal with all the $\rho(s_x)$ terms on the right hand sides, but in our rules there are none.

In the second step we have to look at all the rules in which ρ does not occur, this is $r_{1,2}$. The left hand side is s_x, since the right hand side does not contain s_x, we extend σ with $\sigma(x) = \theta^{-1}(1) = 1$.

In the last step we look at all the rules that *do* contain ρ, which is $r_{1,1}$. The rule is in the form $\rho^i(s_y) \rightarrow t$ with $i = 1$ and $t = h(s_y)$, so we do not need any fresh variables and set $\rho(y) = \theta^{-1}(t) = h(y)$.

That concludes the second algorithm, the resulting substitutions are $\rho = \{y \leftarrow h(y)\}$ and $\sigma = \{x \leftarrow 1\}$. It is clear that these substitutions lead to equal terms, namely to $f(h(1), g(1, 1))$.

Example 2

To show the workings of all steps of the second algorithm, we take a hypothetical result that we could get from the first algorithm and extract the

solution from it. Suppose we got the following two rules from the first algorithm (note that these rules cannot be an actual outcome, as $r_{2,2}$ could be rewritten with $r_{2,1}$):

$$\begin{aligned} r_{2,1}: \quad & \rho(s_x) \rightarrow g(s_y) \\ r_{2,2}: \quad & \rho(\rho(s_y)) \rightarrow g(\rho(s_x)) \end{aligned}$$

In the first step, we have to deal with the $\rho(s_x)$ on the right side of rule $r_{2,2}$. We create a fresh variable u with the corresponding special symbol s_u where $\theta(u) = s_u$. We also extend ρ with $\rho(x) = u$. Finally we have to replace $\rho(s_x)$ with s_u in all rules, resulting in:

$$\begin{aligned} r_{2,1}: \quad & s_u \rightarrow g(s_y) \\ r_{2,2}: \quad & \rho(\rho(s_y)) \rightarrow g(s_u) \end{aligned}$$

Now we can go on to the next step: processing all rules in which ρ does not occur. For us this means that we have to process $r_{2,1}$. Since s_u is not a subterm of the right side, we have to extend σ with $\sigma(u) = \theta^{-1}(g(s_y)) = g(y)$.

Lastly, we have to process the rules in which ρ *does* occur. In our case this is only $r_{2,2}$. The rule is in the form $\rho^i(s_y) \rightarrow t$ where $i = 2$ and $t = g(s_u)$. Since we have $i = 2$, we should create one fresh variable v. We then extend ρ with $\rho(y) = v$ and $\rho(v) = \theta^{-1}(t) = g(u)$.

The two resulting substitutions are $\rho = \{x \leftarrow u, y \leftarrow v, v \leftarrow g(u)\}$ and $\sigma = \{u \leftarrow g(y)\}$.

Chapter 6

Unfolding

Unfolding is yet another method to detect non-termination. However, this time the method does not detect non-termination by itself. Rather it makes use of our previously defined method semi-unification. The unfolding technique, as it name suggests, unfolds a term rewriting system in such a way that we can use semi-unification on the resulting rules. Intuitively, we can see this as creating existing rules where we apply rules on the right hand sides. We will follow Payet's paper[27], and adapt the technique where needed for many-sorted terms.

The paper defines two main techniques: firstly a "normal" unfolding (which we will refer to as concrete unfolding), then an improved version upon that called abstract unfolding. We will look at both of these and also implement them into analysers.

6.1 Concrete Unfolding

The concrete unfolding technique has one main operator: the unfolding operator. The definition as in [27] is as follows:

Definition 6.1.

$$T_R(X) = \left\{ \left(l \to r\left[r'\right]_p \right)\theta \;\middle|\; \begin{array}{l} l \to r \in X, \\ p \in NPos(r), \\ l' \to r' \in R \text{ renamed with fresh variables}, \\ \theta \in mgu(r|_p, l') \end{array} \right\}$$

Where $NPos(r)$ gives back all the non variable positions and $r\left[r'\right]_p$ means replacing position p in r with r'.

From this an unfolding sequence can be created. The idea is to feed the result of an unfolding into the next step as follows.

Definition 6.2. *The unfolding sequence is inductively defined as follows:*

$$T_R \uparrow 0 = R$$

$$T_R \uparrow (n+1) = T_R(T_R \uparrow n) \text{ where } n \in \mathsf{N}$$

This operator by itself works as a non-termination analyser. Obviously a limit has to be set on the maximum number of unfoldings. To use this as an analyser, we can first do an unfolding step, then do the same thing as we did with semi-unification: for each non-variable subterm on the right hand side, check if it semi-unifies with the left hand side.

We can use this operator and do our analysis on the result because it retains the non-termination property of a rewriting system. Intuitively, all we are doing is applying rules on some subterms of right hand sides. And applying rules does retain the non-termination property.

Example 6.1 *We will omit types in this example.* Take the following rewriting system:

$$\begin{aligned} r_1: \quad f(x) &\to g(x) \\ r_2: \quad g(x) &\to f(x) \end{aligned}$$

Clearly, this system is non-terminating: we can indefinitely apply the rules given a term that contains either of the functions f or g. Note that the semi-unification analyser will not be able to detect this, as there exist no non-variable subterms on right hand sides that semi-unify with the left hand side. Now let us see what happens when we apply a step of unfolding. We take as $(l \to r) = r_1$, the position $p = \epsilon$, $l' \to r'$ with fresh variables $= g(x') \to f(x')$. Now we have a most general unifier $mgu(g(x), g(x')) = \{x \leftarrow x'\}$. So our resulting rule is $f(x) \to f(x)$. We do not have to look any further as the left hand sides semi-unifies with the right hand side: we can take empty substitutions for both ρ and σ.

Example 6.2 (The same example as in [27], from [32], types are omitted). Take the following rewrite system:

$$\begin{aligned} r_1: \quad f(0,1,x) &\to f(x,x,x) \\ r_2: \quad g(x,y) &\to x \\ r_3: \quad g(x,y) &\to y \end{aligned}$$

We know this system is non-terminating (take the term $f(0,1,g(0,1))$), but as we will see, our current definition of unfolding is not strong enough to

prove non-termination for this example. We start with the first level of unfolding, $T_R \uparrow 0 = R$. At this level we cannot prove non-termination using semi-unification as there is no non-variable position on a right hand side that semi-unifies with the left hand side of that rule. When we try to unfold to the next level, we see that it is not possible to unfold rules r_2 and r_3, since they do not have a non-variable subterm on the right hand side. r_1 also cannot be unfolded, the only non-variable subterm on the right is $f(x, x, x)$. And there is no left hand side of a rule that semi-unifies with that term. So at the first level we already have no terms left.

As we have just seen, we need some way to pre-process the rules in order to get useful unfoldings. For this, we can use the Augmented TRS (also from [27]).

6.2 Augmented Term Rewriting System

An augmented trs, denoted R^+, is a extension of an existing TRS R. The idea is to substitute variables in rules with left hand sides from other rules. This way, we know that we will be able to apply rules when unfolding. More precisely:

Definition 6.3. *For all rules $l \to r \in R$, create rules $(l \to r)\theta$ in R^+. Here θ is a substitution in the form of $\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$, where $n \in \mathsf{N}$, $\{x_1, \ldots, x_n\} \in Var(l)$ and t_i is a left hand side in R that is variable disjoint from $l \to r$ and all other t_j in θ.*

Again we need to make an adaptation to make this work for many-sorted terms. In particular, we only want to replace a variable with some left-hand side if they have the same type. So a substitution $\{x_i \leftarrow t_i\}$ can only be made if $type(x_i) = type(t_i)$. Also, note that this augmented system retains the non-termination property of the original TRS. We are basically just creating extra rules where variables are already replaced with terms on which we know we can apply rules.

To use this augmented TRS in an analyser, we only have to change our initial unfolding step. We defined this as $T_R \uparrow 0 = R$, now we adapt this to R^+, such that $T_R \uparrow 0 = R^+$. The other step stays the same: $T_R \uparrow (n + 1) = T_R(T_R \uparrow n)$.

Example 6.3 (Example 6.2 continued): To create the augmented rewrite system using a rewrite system with the rules r_1, r_2 and r_3, we consider the rules one by one. If we look at the first rule, there is only one variable: x. We can either not change this variable (leaving us with an empty substitu-

tion), replace it with the left hand side of r_1, or replace it with the left hand side of r_2 (we can ignore r_3 as its left hand side is equal to that of r_2). So for r_1 we have:

$$\theta_{r_1,0} = \emptyset, \theta_{r_1,1} = \{f(0,1,x') \leftarrow x\}, \theta_{r_1,2} = \{g(x',y') \leftarrow x\}.$$

The substitutions for the other two rules are going to be equal as their left sides are equal:

$$\begin{array}{lcllcl}
\theta_{r_2,0} & = & \emptyset & \theta_{r_2,5} & = & \{f(0,1,x') \leftarrow y, f(0,1,x'') \leftarrow x\} \\
\theta_{r_2,1} & = & \{f(0,1,x') \leftarrow x\} & \theta_{r_2,6} & = & \{g(x',y') \leftarrow y, f(0,1,x'') \leftarrow x\} \\
\theta_{r_2,2} & = & \{g(x',y') \leftarrow x\} & \theta_{r_2,7} & = & \{f(0,1,x') \leftarrow y, g(x'',y') \leftarrow x\} \\
\theta_{r_2,3} & = & \{f(0,1,x') \leftarrow y\} & \theta_{r_2,8} & = & \{g(x',y') \leftarrow y, g(x'',y'') \leftarrow x\} \\
\theta_{r_2,4} & = & \{g(x',y') \leftarrow y\} & & &
\end{array}$$

The resulting augmented rewrite system then consists of all the substitutions applied on the rules.

Example 6.4 (Example 6.3 continued): We can now prove using the augmented rewriting system combined with the unfolding operator, that the rewriting system is non-terminating. Take the rule resulting from the substitution $\theta_{r_1,2}$ on r_1:

$$f(0,1,g(x',y')) \rightarrow f(g(x',y'),g(x',y'),g(x',y'))$$

At the first unfolding level we can take position $1.\epsilon$ of the right hand side and the rule $g(x'',y'') \rightarrow x''$ to get to the rule:

$$f(0,1,g(x',y') \rightarrow f(x',g(x',y'),g(x',y'))$$

Then, on the next unfolding level, we can do the same using positon $2.\epsilon$ and the rule $g(x'',y'') \rightarrow y''$. This leads to the rule:

$$f(0,1,g(x',y')) \rightarrow f(x',y',g(x',y'))$$

Now note that we can semi-unify the two sides by taking $\rho = \emptyset$ and $\sigma = \{0 \leftarrow x', 1 \leftarrow y'\}$. Therefore, the TRS does not terminate.

Example 6.4 (Example 6.2 continued): Let us now take a look at the same rules *with* typing. We will type the functions as follows:

$$\begin{array}{lcl}
f & :: & \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha \\
0 & :: & \alpha \\
1 & :: & \alpha \\
g & :: & \beta \rightarrow \beta \rightarrow \beta
\end{array}$$

Note that the previous example which was non-terminating does not exist anymore, as we cannot use g in an argument of f. Furthermore, the rewriting system with these rules *is* terminating. Let us take a look at the augmented system from the rules. It is clear that the new augmented system will have a subset of the rules of the previous example. We just need to cross out all of the substitutions where g is used in f and vice versa:

$$\begin{array}{lcllcl} \theta_{r_1,0} & = & \emptyset & \theta_{r_1,1} & = & \{f(0,1,x') \leftarrow x\} \\ \theta_{r_2,0} & = & \emptyset & \theta_{r_2,1} & = & \{g(x',y') \leftarrow x\} \\ \theta_{r_2,2} & = & \{g(x',y') \leftarrow y\} & \theta_{r_2,3} & = & \{g(x',y') \leftarrow y, g(x'',y'') \leftarrow x\} \end{array}$$

These augmented rules will result in 11 rules after the first unfolding, then 6 after the second and third, to finally result in 0 resulting rules after the fourth. None of these resulting rules are semi-unifiable.

6.3 Implementation

The unfolding analyser is implemented in the `ConcreteUnfoldingAnalyser`. This implementation also includes pre-processing the given TRS into an augmented TRS.

Chapter 7

Abstract Unfolding

The abstract unfolding analyser is an improvement upon the concrete variant. The abstract unfolding tries to prevent useless rules from persisting through the unfolding operator. It also prevents as many calls to the semi-unification check as possible by first processing all the rules $l \to r$ into rules $l \to r'$, where r' is some subterm of r. The calls to semi-unification are reduced by only applying the semi-unification algorithm on the root positions of r'.

7.1 Useful Pairs

Definition 7.1. *We call a pair of terms (l, r) useful when either l and r semi-unify, or when the pair can be unfolded to a pair of terms (l', r') such that l', r' semi-unify.*

Notice that when we find a useful pair, we have exactly what we need. Because then we know we can unfold to a pair of terms that semi-unify, thus non-terminate. We know that this is a property that is undecidable. For this reason, we create the notion of *"is probably useful"*. The set of *probably useful* terms is an overestimation of the actual useful terms. To increase clarity, we sometimes use the term "definitely useful" to describe useful terms. We will integrate this notion into an abstract domain:

7.1.1 Abstract Domain

The abstract domain is much like a normal domain for a TRS, consisting of pairs of terms (l, r). Except the abstract domain also knows two special elements, namely `true` and `false`. Here, `true` denotes pairs of terms that semi-unify and `false` denotes non-useful pairs of terms.

7.1.2 Abstraction

The abstraction function is a function that creates an abstract TRS using a concrete TRS. Note how it applies the flattening step as described before:

Definition 7.2.

$$\alpha(R) = \bigcup_{l \to r \in R} \{\alpha_R(l, r|_p) \mid p \in Pos(r) \wedge type(l) = type(r|_p)\}$$

$$\alpha_R(l, r) = \begin{cases} \texttt{true} & \textit{if } (l, r) \textit{ semi-unify} \\ (l, r) & \textit{if } (l, r) \textit{ is probably useful for } R \\ \texttt{false} & \textit{otherwise} \end{cases}$$

7.1.3 Unfolding operator

The unfolding operator is a lot like the operator we defined for the concrete unfolding, only this time we will apply the abstraction function on the resulting pairs of terms:

Definition 7.3.

$$T_R^{\#}(X^{\#}) = \left\{ \alpha_R\left(l\theta, r\left[r'\right]_p \theta\right) \;\middle|\; \begin{array}{l} l \to r \in X^{\#}, \\ p \in NPos(r), \\ l' \to r' \in R \textit{ renamed with fresh variables}, \\ \theta \in mgu(r|_p, l') \end{array} \right\}$$

Note that this takes as input an abstract domain, so also the special elements `true` and `false`. The element `false` can always be ignored, as it only represents non-useful terms. The element `true` needs its own case, however. If $\texttt{true} \in X^{\#}$, then $T_R^{\#}(X^{\#}) = \{\texttt{true}\}$.

With this operator we can define the unfolding sequence just as before. We do this by replacing the previous operator T_R with $T^{\#}$ and by applying the abstraction function on the augmented TRS in the first level.

Definition 7.4. *The abstract unfolding sequence is inductively defined as follows:*

$$T_R^{\#} \uparrow 0 = \alpha(R^+)$$

$$T_R^{\#} \uparrow (n+1) = T_R^{\#}(T_R^{\#} \uparrow n), \text{ with } n \in \mathsf{N}$$

With this definition we can prove non-termination: if there exists an unfolding such that `true` is an element of that unfolding, the TRS does not terminate.

7.2 Probably Useful Pairs

As mentioned before, the definition of useful terms is exactly what we want to find. Thus we created the notion of *probably useful terms*, for us to use in our analyser. In this section we will look at how we can define the *probably useful terms*. We will follow Payet in his steps and again adapt it where needed. We start off with a lemma about definitely useful terms. This lemma will lead us to a definition for *probably useful pairs* that is good enough for us to use in our analyser.

Lemma 7.1 If (l, r) is a pair of definitely useful terms, then either l and r semi-unify, or (l, r) can be unfolded to a pair of terms (l', r'), such that $\mathit{root}(l) = \mathit{root}(l')$ and $\mathit{root}(r') \in \{\mathit{type}(l), \mathit{root}(l)\}$. Where $\mathit{root}(r)$ gives us either the function symbol of r if it is a function, or $\mathit{type}(r)$ if it is a variable.

This notion of the root of a term is different from the one specified by Payet. We need a different notion to be able to compare types of terms. Payet's definition gave back $\perp$ when applying the root function to a variable. If we would do the same, we lose the information of the type of the variable.

We will now split up the lemma into two parts and analyse how we can use them for a definition of probably useful terms.

For a given (definitely) useful pair of terms (l, r), there are two possibilities:

1. l and r already semi-unify,
2. the pair can be unfolded to a pair of terms (l', r') that do semi-unify.

The first case uses semi-unification, we already defined this so we will be able to use this as a criterion.

In case 2, we know that r cannot be variable, since it needs at least one step of unfolding. So we say that r is in the form of $f(t_1, \ldots, t_n)$. Then we have two possible ways to go from r to r':

2.1 There is no unfolding step at the root position, so r' is in the form of $f(t'_1, \ldots, t'_n)$ and each t_i is unfolded to t'_i.

2.2 There is an unfolding step at the root position. Here, we first unfold each t_i to a term t'_i to get $f(t'_1, \ldots, t'_n)$. Then we apply a rule $f(s_1, \ldots, s_n) \to r_1$ where r_1 will further lead to r'.

We will first look at case 2.1 where l is not a variable. We know that r' is not a variable. So by lemma 7.1 we know that $root(r') = root(l)$ and thus l must have the form $f(s_1, \ldots, s_n)$. Again by lemma 7.1, we know that l' is in the form $f(s'_1, \ldots, s'_n)$. Now notice that t_i unfolds to t'_i and s'_i semi-unifies with t'_i. We thus know that (s_i, t_i) is a useful pair.

Now we take a look at the next case: case 2.2, also where l is not a variable. We notice the following:

Let $f(t_1, \ldots, t_n)$ be a term where each t_i can be unfolded to t'_i in any number of steps. Now suppose we have a term $f(s_1, \ldots, s_n)$ such that:

$$mgu(f(t'_1, \ldots, t'_n), f(s_1, \ldots, s_n) \text{ with fresh variables}) \neq \emptyset$$

Then we know that for each argument t_i either:

2.2.1 t_i unifies with any variable disjoint variant of s_i,

2.2.2 or t_i can be unfolded in at least one step to some term whose root is equal to the root of s_i or $type(s_i)$.

We also know that the right hand side of the rule we apply to $f(t'_1, \ldots, t'_n)$, r_1 (from case 2.2), has to lead to r'. So using lemma 7.1 we can say that the root symbol of r_1 should be equal to the root symbol of l or $type(l)$. We define this "leading to" as a path in the graph of functional dependencies as criterion 2.3. The graph will be used to create a definition in which cases 2.2.1 and 2.2.2 are handled. The graph of functional dependencies is defined as follows:

7.2.1 Graph of Functional Dependencies

The graph of functional dependencies is a three-tuple consisting of the edges E, vertices V and initial vertices I $\langle E, V, I \rangle$. The initial vertices are a subset

of the vertices and will be used in a later definition. We denote a functional dependency graph as G_R, where R is the TRS from which the graph is generated.

The algorithm to compute the graph is divided into two steps. The first step generates part of the edges from the input TRS R and all the (initial) vertices. The second step possibly generates more edges.

Definition 7.5.

$$\frac{l \to r \in R}{\langle E, V, I\rangle \mapsto \langle E \cup \{l \to \mathit{root}(r)\}, V \cup \{l, \mathit{root}(r)\}, I \cup \{l\}\rangle}$$

$$\frac{\begin{array}{c} l \to f \in E \wedge l' \to g \in E \wedge \\ l \in I \wedge l' \in I \wedge \\ (\mathit{root}(l') = f \vee f = \mathit{type}(l')) \end{array}}{\langle E, V, I\rangle \mapsto \langle E \cup \{f \to l'\}, V, I\rangle}$$

Note that this definition differs from the one given in [27]. Our definition of the *root* function gives back the type of the term if that term is a variable. In the case where the term is not a variable it still gives back the function symbol of the term. Also, in the second rule, we check "$\mathit{root}(l') = f \vee f = \mathit{type}(l')$". The second part of this check differs from the existing definition. The definition by Payet for this was "$\mathit{root}(l') = f \vee f = \bot$", but we cannot just check for some $\bot$ (which was used to define the root of a variable), as the variable that created that $\bot$ vertex in the graph might have a different type. This way we can check if edges do not break typing rules.

Example 7.5 *(Example 6.2 continued).* This example also comes from [27]. We will first cover this example and then do the same but with a typed system, as in example 6.4. When we create the graph of functional dependencies from the system, we get:

$$f \longleftrightarrow \underline{f(0,1,x)} \longleftarrow \alpha \longleftrightarrow \underline{g(x,y)}$$

Note that the initial vertices are $f(0,1,x)$ and $g(x,y)$ (underlined for clarity). The edge from $f(0,1,x)$ to f is there because of the first rule. The same holds for the edge from $g(x,y)$ to α. The other edges are created with the second rule as follows. The edge from f to $f(0,1,x)$ can be created in the case of $l \to f \in E = l' \to g \in E = f(0,1,x) \to f$, then $\mathit{root}(l') = f$. The edge α to $g(x,y)$ can be created in the case of $l \to f \in E = l' \to g \in E = g(x,y) \to \alpha$, then $f = \mathit{type}(l') = \alpha$. The last edge from α to $f(0,1,x)$ can be created in the case $l \to f \in E = g(x,y) \to \alpha$, $l' \to g \in E = f(0,1,x) \to f$, then $f = \mathit{type}(l') = \alpha$.

Example 7.6 *(Example 7.5 continued).* We again type the system created as follows:

$$\begin{aligned} f &:: \alpha \to \alpha \to \alpha \to \alpha \\ 0 &:: \alpha \\ 1 &:: \alpha \\ g &:: \beta \to \beta \to \beta \end{aligned}$$

After applying the rules, we get the following graph:

$$f \longleftarrow \underline{f(0,1,x)}$$

$$\beta \longleftarrow \underline{g(x,y)}$$

Note how, compared with the previous example, we miss the edge from α to $f(0,1,x)$. This is because the type of g is now β, which means that the criterion for creating the edge does not hold anymore. This results in a disconnected graph.

7.2.2 The Relation $\xrightarrow[G_R]{+}$

The initial vertices in the graph are exactly the left hand sides of the rules in the input TRS. So a path in G_R from some initial vertex s to a function symbol (or type) f means that for any term s' for which holds that $mgu(s, s') \neq \emptyset$, we *may be able to* rewrite s', using the rules from the TRS, to a term whose root equals f. The first step of the unfolding is done at the root position of s'. Note that we say that we *may* be able to. This is where the overestimation comes into place, as this is not always the case. We can now create the following definition for case 2.2.

Definition 7.6. *We create a relation $\xrightarrow[G_R]{+}$. We take as input the graph G_R of a TRS R, a term $f(t_1, \ldots, t_n)$ and a function symbol (or type) g. We can write $f(t_1, \ldots, t_n) \xrightarrow[G_R]{+} g$ if there is a path in G_R from an initial vertex in the form of $f(s_1, \ldots, s_n)$ to g (Criterion 2.3) and for each $i \in [1, n]$, either of the following three statements holds:*

- *$mgu(t_i, s_i$ renamed with fresh variables$) \neq \emptyset$ (Case 2.2.1),*
- *$t_i \xrightarrow[G_R]{+} root(s_i)$ (Case 2.2.2 part 1),*
- *$t_i \xrightarrow[G_R]{+} type(t_i)$ (Case 2.2.2 part 2)*

We again have adapted Payet's definition in the third case. The original definition is as follows: $t_i \xrightarrow[G_R]{+} \bot$. Since we used the type of the variable instead of $\bot$ when looking at the root, we have changed this.

Example 7.7 (*Example 7.5 continued*). Recall the TRS and functional dependency graph:

$$\begin{aligned} r_1: \quad f(0,1,x) &\to f(x,x,x) \\ r_2: \quad g(x,y) &\to x \\ r_3: \quad g(x,y) &\to y \end{aligned}$$

$$f \longleftrightarrow \underline{f(0,1,x)} \longleftarrow \alpha \longleftrightarrow \underline{g(x,y)}$$

We will look at a number of examples whether or not the relationship $\xrightarrow[+]{G_R}$ holds.

- $g(v,w) \xrightarrow[G_R]{+} \alpha$: this holds, we can take the initial vertex $g(x,y)$, there exists a path from $g(x,y)$ to α, and both $mgu(v,x)$ and $mgu(w,y)$ are non-empty.
- $g(v,w) \xrightarrow[G_R]{+} f$: this also holds, we again take the initial vertex $g(x,y)$, then there exists a path from $g(x,y)$ to f, and both the arguments can unify with x and y respectively.
- $f(0,1,x) \xrightarrow[G_R]{+} \alpha$: this does not hold, there does not exist an initial vertex in the form of $f(s_1,s_2,s_3)$ such that there is a path from $f(s_1,s_2,s_3)$ to α.
- $f(0,x,0) \xrightarrow[G_R]{+} f$: this does hold, we take initial vertex $f(0,1,x)$, then there exists a path from $f(0,1,x)$ to f. Then 0 unifies with 0 (the first argument), 1 unifies with x (the second argument) and lastly 0 unifies with x as well (the third argument).

Example 7.8 (*Example 7.6 continued*). If we were to use the typed system and do the same checks as in Example 7.7, the second statement ($g(v,w) \xrightarrow[G_R]{+} f$) does not hold anymore. This is because the path from $g(x,y)$ to f is no longer there. The third statement then also does not hold anymore, since the vertex α we used now is replaced with β (the type of g).

7.2.3 The relation $useful_R$

We can now capture all of our cases (labelled for clarity) in one definition.

Definition 7.7. *We create a relation called useful$_R$. For any given TRS R and two input terms l and r, this relation overestimates whether they are useful for R. One of the following conditions has to hold:*

- *l semi-unifies with r (Case 1),*
- *l is in the form $f(s_1, \ldots, s_n)$, r is in the form $f(t_1, \ldots, t_n)$ and for each $i \in [1, n]$, useful$_R(s_i, t_i)$ (Case 2.1),*
- *l is in the form $f(s_1, \ldots, s_n)$, r is in the form $g(t_1, \ldots, t_m)$ and $r \xrightarrow[G_R]{+} g$ or $r \xrightarrow[G_R]{+} type(r)$ (Case 2.2 the sub-cases and the criterion are defined within the $\xrightarrow[G_R]{+}$ definition)*

We can use this definition of useful in our abstraction function to be used in the analyser.

Example 7.9 *(Example 6.4 continued).* Let us take a look at an example of the abstraction function now that we have a notation for the usefulness of two terms for a TRS. First, recall the TRS, the types and the resulting graph of functional dependencies:

$$\begin{array}{llcl} r_1: & f(0,1,x) & \to & f(x,x,x) \\ r_2: & g(x,y) & \to & x \\ r_3: & g(x,y) & \to & y \end{array} \qquad \begin{array}{lcl} f & :: & \alpha \to \alpha \to \alpha \to \alpha \\ 0 & :: & \alpha \\ 1 & :: & \alpha \\ g & :: & \beta \to \beta \to \beta \end{array}$$

$$f \longleftrightarrow \underline{f(0,1,x)}$$
$$\beta \longleftrightarrow \underline{g(x,y)}$$

Also recall the abstraction function:

$$\alpha(R) = \bigcup_{l \to r \in R} \{\alpha_R(l, r|_p) \mid p \in Pos(r) \wedge type(l) = type(r|_p)\}$$

$$\alpha_R(l,r) = \begin{cases} \texttt{true} & \text{if } (l,r) \text{ semi-unify} \\ (l,r) & \text{if } (l,r) \text{ is useful for } R \\ \texttt{false} & \text{otherwise} \end{cases}$$

We will apply the abstraction function to the TRS with rules r_1 through r_3. First take the rule $r_1 : f(0,1,x) \to f(x,x,x)$. The right side of this rule has two unique positions, namely ϵ and $1.\epsilon$. For these two positions we will apply the $\alpha_R(l, r|_p)$ function.

- ϵ: First we check if $(f(0,1,x), f(x,x,x)$ semi-unify, which is not the case. Then we have to check if this tuple is useful for R, then either the terms need to semi-unify, which they do not, or each subterm needs to be useful for R. This does hold, since 0 semi-unifies with x, 1 semi-unifies with x and x semi-unifies with x. We get $\alpha_R(f(0,1,x), f(x,x,x)) = (f(0,1,x), f(x,x,x))$.
- $1.\epsilon$: Again, we first check if $(f(0,1,x), x)$ semi-unify, which is not the case. Then in our check for usefulness, we note that the we can only do the first check, since x is not in the form of $f(s_1, \ldots, s_n)$. This first check is semi-unification, which we already checked and resulted negatively. So we get $\alpha_R(f(0,1,x), x) = \texttt{false}$.

Now we will look at the second rule r_2, this rule only has one position on the right hand side: ϵ.

- ϵ: We first check if $(g(x,y), x)$ semi-unify, which fails. Then we go to the second case and check if the terms are useful for R. The first check of semi-unification obviously fails. The second and third check again require the terms to not be variables, which is not the case, so we get $\alpha_R(g(x,y), x) = \texttt{false}$.

The third rule r_3 is much like the second rule, again we have one position on the right hand side: ϵ. The checks for this rule in the abstraction function are the same and get the same result. So we also get $\alpha_R(g(x,y), y) = \texttt{false}$.

The only useful rule we got is $f(0,1,x) \to f(x,x,x)$. If we were to analyse this further, we first have to look if there already is a non-variable position on the right hand side that semi-unifies with the left hand side. This is not the case as the only non-variable position on the right hand side is $f(x,x,x)$. In the unfolding operator we take this rule, then choose a non-variable right hand side position (so $f(x,x,x)$). Then we have to find a rule in the original TRS $l' \to r'$ (with fresh variables) for which holds that $mgu(f(x,x,x), l') \neq \emptyset$. The only rule that could be used keeping types in account, is $f(0,1,x') \to f(x',x',x')$. And this does not work as we cannot unify $f(x,x,x)$ and $f(x',x',x')$. This means that we have no rules left and the analyser failed in detecting non-termination.

Note that we did not use an augmented TRS as input as per the definition for simplicity.

7.3 Adaptations made for types

Due to the length of this (and the previous) chapter, we list the adaptations we made once more.

First off, for both the unfolding operators, we assume that the most general unifier calculated respects the types in the terms. Otherwise there can be replacements of positions by terms which are invalid when looking at the types (definitions 6.1 and 7.3).

Then we adapted the notion for the Augmented TRS, this meant that we had to add an extra typing check to prevent illegal typing when creating extra rules 6.3.

In the abstract unfolding technique, the first adaptation was in definition 7.2, where we added an extra typing check to prevent illegal pairs from being created.

The most important change in the abstract unfolding was the one to the $root(t)$ function. The corresponding definition in [27] returns $\perp$ when given a variable, now it gives back the type of the variable when given one. This basically meant that all previous instances of $\perp$ had to be replaced with a form of $type(t)$. It is important that these replacements are given the correct term as an argument. These changes can be found in Lemma 7.1, Definition 7.5, Definition 7.6 and Definition 7.7.

Chapter 8

Experiments

8.1 Competitions

To benchmark tools that analyse term rewriting systems, there exist competitions. Tools can be submitted to such competitions and will be tested on a (usually large) number of examples. The results are open to everyone and allow us to see which tools perform best.

These competitions are useful for us too, since the set of benchmark TRSs is freely available. We can use these to test our analyser and then also compare results.

We will look at benchmark sets from three different competitions. Firstly we will run our analyser on the TPDB (Termination Problem Data Base)[10]. This database contains all the problems used in the (non-)termination competition. In this competition the idea is to classify as many input TRSs as either terminating or non-terminating as possible.

Another competition we will take benchmarks from is the confluence competition[1]. This competition focuses on another property of term rewriting systems: confluence. The main reason we will also test our tool on this set is because this set contains many-sorted TRSs, where the termination competition does not contain many-sorted examples.

The last set of benchmark TRSs come from the higher order competition. The tool WANDA[19] was adapted to isolate the first order parts from the original higher order TRSs so that they could be used for our experiments. This is possible because the higher order competition does have types.

8.2 Experiments

The testing was mostly limited to the unfolding technique, because it encompasses all the other techniques before. We did however, test the examples on both the concrete and abstract analyser. All the experiments were run with a time limit of ten seconds and at most ten unfolding steps.

These are not the only tests we can run. Recall that the unfolding technique uses the semi-unification algorithm. We can also swap in the unification in its place. This will not give us incorrect results as we have seen that unification and semi-unification are both capable of detecting non-termination. The unification algorithm is a faster algorithm than semi-unification, though. It is interesting to find out if using semi-unification over unification is significantly better at finding solutions. It could for example be the case that for a TRS, unfolding with unification is enough to prove non-termination. If this is combined with the fact that the TRS is very large and unfolding with semi-unification results in a timeout, it might actually be useful to run the analyser with unification too. Note that semi-unification is still a technique that is theoretically stronger, unification will most definitely find less results.

Another test is to run the analyser without augmenting the TRS as a pre-processing step. Even though the augmenting step might increase the chance of finding non-termination, it also can increase the number of rules significantly, which in its place increases the running time of the algorithm.

To run the tests and visualise the results, a small framework was created in Python to which the link can be found in the appendices. The tests were run on a Windows machine with an i7-8565U CPU and 16GB of RAM.

8.3 Results

8.3.1 TPDB

The first results (Table 8.1 and Table 8.2) are the results of running our analyser on the Termination Problem Data Base[10] with a timeout of ten seconds. Note that these examples are all untyped and thus not what our tool is created for. These tables show concrete and abstract unfolding respectively. Both of these tables show the results of augmenting versus not augmenting and using semi-unification versus using unification.

We also ran the examples on pure semi-unification (so no unfolding

steps). We can do this by taking the concrete unfolder and setting the maximum unfoldings to zero. Doing this, we found 64 out of the 1259 examples to be non-terminating.

	Augmented		Non Augmented	
	Semi-Unification	Unification	Semi-Unification	Unification
Non Terminating	118	79	120	76
Maybe	159	169	441	464
Timeout	982	1011	698	719
Total Time (HH:MM:SS)	2:51:55	2:52:13	2:06:44	2:09:17
Avg. without timeouts (sec)	1.79	0.65	1.11	1.05

Table 8.1: TPDB: Concrete unfolding analyser results

	Augmented		Non Augmented	
	Semi-Unification	Unification	Semi-Unification	Unification
Non Terminating	123	85	117	73
Maybe	235	259	561	610
Timeout	901	915	581	576
Total Time (HH:MM:SS)	2:35:29	2:38:13	1:46:02	1:45:46
Avg. without timeouts (sec)	0.89	1.00	0.81	0.86

Table 8.2: TPDB: Abstract unfolding analyser results

Results unfolding

The results from the experiments are consistent. It is clear that choosing semi-unification instead of unification is always a good idea. Semi-unification finds about 50% more systems that are non-terminating in about the same time. However, the comparison on the total time is not completely fair. Since unification finds fewer non-terminating systems, it more often needs to go on for the entire timeout limit (or until the analysis is done). This can be seen in the last row of each result table. Note that this difference is not seen when the difference in timeouts is small.

The most important observation we can make is that augmenting the term rewriting system as a pre-processing step is expensive. However, it is still worthwhile: we have already seen an example where not augmenting would result in a `MAYBE` result but with augmenting we were able to find

a loop. This occurs many more times in systems we have analysed. In the results of the abstract analyser we find more non-terminating TRSs when augmenting (123 instead of 117). But there are also systems where augmenting would result in a timeout and not augmenting *would* find the non-termination. There were 19 cases where augmenting found a result where not augmenting would not and 15 cases where not augmenting would find a result where augmenting would not.

Comparing the abstract unfolding and concrete unfolding matches expectations. There is no example where the concrete unfolder *was* able to find non-termination and where the abstract unfolder returned `MAYBE`. There are cases, however, where the abstract unfolder timed out and the concrete unfolder was able to find non-termination. Overall, the abstract unfolder detected more cases of non-termination.

There is one special example, using the concrete unfolder and by not augmenting the system as a pre-processing step, we were able to detect non-termination in an example where the other tools in the most recent termination competition were not. It is `Zantema_15\s.xml` from the Termination Problem Data Base. It is the following TRS:

$$r_1 : \ a(a(a(S,x),y),z) \ \rightarrow \ a(a(x,z),a(y,z))$$

Non-termination is proven by unfolding on the positions in the following order: ϵ, $2.\epsilon$, $2.2.\epsilon$, $2.2.2.\epsilon$ and finally $2.2.2.2.\epsilon$. Then semi-unification is possible between the left hand side and position $1.\epsilon$ on the right hand side.

Results pure semi-unification

Pure semi-unification is not able to find any new results on top of what we already knew. This is because the setup is embedded into concrete unfolding with more than zero unfoldings.

8.3.2 Confluence Problem Database

We applied our analyser to a number of problems from the confluence database. Since these examples are labelled, we can extract all rewriting systems labelled as non-terminating or terminating. These labels were mainly a way to test our analyser. Note that these systems labelled as (non-)terminating are all untyped.

We tested the 418 term rewriting systems labelled as `non-terminating` with the abstract unfolding technique using semi-unification. Even without

augmenting all of the 418 systems come out as non-terminating in 12 seconds. When augmenting is used as a pre-processing step, a few timeouts (at ten seconds) occurred. When we use unification instead of semi-unification we are not able to find all the results (both with and without augmenting).

For 95 rewriting systems labelled as `terminating`, we found none of them being non-terminating.

There are also some many-sorted benchmarks in this database. For these examples, we ran the same tests as we did with the TPDB. This means that we tried both the concrete and abstract analyser on augmenting versus not augmenting and semi-unification versus unification. The results from these tests can be found in tables 8.3 and 8.4.

	Augmented		Non Augmented	
	Semi-Unification	Unification	Semi-Unification	Unification
Non Terminating	34	33	34	33
Maybe	5	5	8	9
Timeout	16	17	13	13
Total Time (HH:MM:SS)	0:03:00	0:03:11	0:02:15	0:02:13

Table 8.3: COPS: Concrete unfolding analyser results

	Augmented		Non Augmented	
	Semi-Unification	Unification	Semi-Unification	Unification
Non Terminating	34	33	34	33
Maybe	8	9	10	11
Timeout	13	13	11	11
Total Time (HH:MM:SS)	0:02:39	0:02:30	0:01:53	0:01:52

Table 8.4: COPS: Abstract unfolding analyser results

The results are all very much the same, there is only one example which could not be solved using unification. Augmenting the TRS as a pre-processing step did not have much impact besides taking slightly longer to analyse.

8.4 Transformed from Higher Order Problems

Lastly, we have the benchmark set that WANDA[19] could extract from the higher order database. We will again run these benchmarks with the same

configurations as before: both concrete- and abstract unfolding both with augmenting versus without and with semi-unification versus unification. The results of these tests can be found in table 8.5 for the concrete unfolding tests and in table 8.6 for the abstract unfolding tests. Note that most of these examples are terminating.

	Augmented		Non Augmented	
	Semi-Unification	Unification	Semi-Unification	Unification
Non Terminating	7	7	4	4
Maybe	43	44	75	75
Timeout	59	58	30	30
Total Time (HH:MM:SS)	0:11:28	0:10:57	0:05:31	0:05:25

Table 8.5: Transformed: Concrete unfolding analyser results

	Augmented		Non Augmented	
	Semi-Unification	Unification	Semi-Unification	Unification
Non Terminating	7	7	4	4
Maybe	51	52	87	88
Timeout	51	50	18	17
Total Time (HH:MM:SS)	0:09:08	0:08:54	0:03:47	0:03:34

Table 8.6: Transformed: Abstract unfolding analyser results

In this case, there was no difference between using semi-unification or unification. There was, however, a difference between not augmenting and augmenting. In particular, not augmenting lead to not detecting non-termination in three examples. There are no examples where not augmenting was successful and augmenting was not.

Chapter 9

Related Work

A lot of related work in the analysis of term rewriting systems is done specifically on the termination side. It should be noted that this work is also relevant to the non-termination analysis. Non-termination is disproven by proving termination and the other way around. Many tools use both termination and non-termination techniques to get as complete an answer as possible. The techniques implemented in this paper might determine when a TRS is non-terminating, but they cannot decide if TRS is terminating, i.e. Mara never answers `TERMINATING`. In this chapter a number of relevant techniques and tools are discussed.

9.1 Dependency Pair Framework

One of the most popular techniques for proving both termination and non-termination is the Dependency Pair Framework[29]. Traditionally it was created for unsorted term rewriting systems. But there are already variants including higher-order TRSs, context-sensitive TRSs and constrained TRSs. The higher-order and constrained TRSs both contain types so a definition for sorted TRSs should also be clear. The dependency pair framework is built upon the dependency pair approach[2]. In the dependency pair framework a set $DP(R)$ is generated from "rules" that identify the function calls in R. For example, from a rule $f(x) \to g(f(x), 0)$, the set $DP(R)$ becomes $DP(R) = \{F(x) \to G(f(x), 0), F(x) \to F(x)\}$. This is because in the rule, there is a call to f and one to g on the right-hand side.

Now the idea is to let both non-termination and termination techniques operate on the dependency pairs instead of on the TRS itself. This way, the problems are smaller and thus easier to solve. Techniques that operate

on dependency pairs are called dependency pair processors or *Proc*. The processor can either return a set of dependency pair problems, or `no`.

The unfolding technique could be implemented as a DP processor and that way be used in the dependency pair framework.

9.2 Finite Automata

A very recent technique for proving non-termination is by using finite automata [7]. Here, the goal is not to directly look for an infinite reduction in a TRS. Instead, a regular language is sought with properties from which non-termination follow. Then these properties are defined in a propositional formula. This formula can be analysed by a SAT solver, if the formula can be satisfied, the original TRS is non-terminating. A tree automaton is used to represent the language that is sought after, in a run of the algorithm, it has a fixed number of states.

9.3 Non-looping Non-Termination

The most common way to detect non-termination in a TRS is to find a loop. This is done by finding a finite sequence of rewriting steps such that the first term is a subterm of the last term. Loops are not the only way to detect non-termination, though. Another way is to find non-looping non-termination, as for example in [6]. Non-looping, as its name suggests, does not have a loop, but is able to create an infinite rewriting sequence. We use an example from [6]:

$$\begin{aligned} r_1: \quad & isNat(0) && \to && true \\ r_2: \quad & isNat(s(x)) && \to && isNat(x) \\ r_3: \quad & f(true, x) && \to && f(isNat(x), s(x)) \end{aligned}$$

This rewriting system does not loop, but also is non-terminating:

$$\begin{aligned} f(true, s^n(0)) &\to && f(isNat(s^n(0)), s^{n+1}(0)) \\ &\to^{n+1} && f(true, s^{n+1}(0)) \\ &\to && f(isNat(s^{n+1}(0)), s^{n+2}(0)) \\ &\to^{n+2} && f(true, s^{n+2}(0)) \\ & && \dots \end{aligned}$$

Here, $\to^n$ means n steps of rewriting and $s^n(0)$ denotes n s symbols. It is clear that the last two rewriting steps can be repeated indefinitely. But

it is not looping: the rewriting steps needed to translate the $isNat(s^n(0))$ terms into $true$ increases every iteration. In [6] a way to detect non-looping termination is presented.

9.4 Some Termination Techniques

In this section a few old but very powerful methods are discussed to detect termination, but these are certainly not all of the popular methods.

9.4.1 Polynomial Interpretations

Polynomial interpretations [3] can be used as a way to detect termination of term rewriting systems. Here, the goal is to give polynomial interpretations for each function symbol and then prove that for each left side is greater than the right side using the interpretations.

This technique has been extended and improved over time for example in [15], to extend the class of rewriting systems of which termination can be detected using negative coefficients. And in [9], to extend the technique to be able to work in higher order term rewriting systems. The latter can be useful because it is already typed, but for higher order systems.

An interesting find is also done in [30]. Here, there are no polynomial interpretations, just interpretations. The suggestion is to give different types different sets. For example, a type `nat` (the natural numbers) can be given the set of the natural numbers, `bool` (booleans) can be given the set $\{0, 1\}$. This way, the typing is saved.

9.4.2 Path Orderings

In Dershowitz [5] a number of orderings are given to prove termination. One of these orderings is called the recursive path ordering. This well known technique works by defining a ordering over the function symbols of a TRS, if the ordering respects the recursive path ordering, the TRS is terminating. In [8] a higher order variant is defined called Computability Path Ordering (CPO). Since CPO already uses types (higher order types), it might be useful in the first-order TRSs.

9.5 Tools

At the time of writing, there is no clear sign of any existing tool to detect non-termination in many-sorted term rewriting systems. This is not really surprising as there also were no techniques for this. There are, however, a lot of (non-)termination tools out there for different kinds of rewriting systems. Some examples are NTI, AProVE, NaTT, T$_{\text{T}}$T$_{2}$, WANDA and TORPA.

9.5.1 NTI

NTI[28] stands for Non Termination Inference, it is a tool to prove non-termination of term rewriting systems. The prover uses guided unfoldings, which is a reconsideration of the unfolding analyser we have adapted[27]. The first implementation of NTI is open source and written in C++, the newest version (which is currently participating in competitions), is not open source[26].

9.5.2 AProVE

AProVE[11] is a tool that can prove both termination and non-termination. AProVE is built upon the Dependency Pair Framework and contains at least 22 processors. These include processors for proving non-termination[12]. AProVE is not open-source.

9.5.3 NaTT

The Nagoya Termination Tool (NaTT) is a closed source termination prover[31]. It was the first tool to implement the weighted path order[31]. NaTT, like AProVE, is built upon the Dependency Pair Framework. The tool does have some non-termination techniques, but is specialised on termination.

9.5.4 T$_{\text{T}}$T$_{2}$

T$_{\text{T}}$T$_{2}$ is the Tyrolean Termination Tool 2[22]. It is the open-source successor of the non open-source T$_{\text{T}}$T[14]. The tool is a termination analyser for first-order term rewriting systems. Just like AProVE, it is based on the Dependency Pair Framework, with processors such as increasing interpretations, a modular match-bound technique, uncurrying and outermost loops[22]. As

with many other tools, it does have some non-termination techniques but is specialised on termination.

9.5.5 WANDA

WANDA is an open-source higher-order termination tool[19]. The tool features a number of techniques which include the Dependendy Pair Framework, polynomial interpretations and higher-order recursive path ordering. The theory of WANDA is based upon[20]. WANDA has very limited non-termination techniques. WANDA also uses a dedicated tool for the first order part of a higher-order system, currently AProVE. When AProVE answers `NO`, WANDA checks if this untyped answer (since it is the first-order part) can be correctly typed for the higher order system. Mara could be used to detect non-termination instead, because then WANDA does not have to do any type checks (since Mara can work with first-order types). This may also hold for other higher-order tools.

9.5.6 TORPA

TORPA is a termination analysis tool that focuses on string rewriting systems [33], TORPA is closed source. It includes a number of techniques, such as: polynomial interpretations, recursive path ordering, the Dependency Pair Framework, RFC-match-bounds and semantic labelling.

Chapter 10

Conclusions

We have to look at two separate points in our conclusions. First off, we adapted a number of techniques to work for many-sorted rewriting systems. The second point we can look at is how they perform on examples.

10.1 Adaptations

The main techniques we have adapted to work for many-sorted term rewriting systems are Concrete unfolding and Abstract unfolding. The impact that the sorts have on the techniques were small but certainly not negligible. Most of the adaptations made were extra checks for types, for example when substituting some position with another term.

Apart from the extra type checks, there have been more changes. A pertinent example in the graph of functional dependencies, where the definition had to be altered altogether to make it work for types.

10.2 Experiments

The results from all of the experiments match expectations. Semi-unification is often stronger than unification. It, together with the abstract unfolding technique, is the best at finding non-termination. What is not directly obvious, is if augmenting as a pre-processing step is always worth it. It usually leads to better results but it also takes considerably more time to calculate for problems that can be solved without augmenting, which may lead to timeouts.

We were also able to find non-termination in one example where no other tool could. This was not with the "strongest" setup; rather with the concrete unfolding with semi-unification and without augmenting beforehand. This might also be the reason the other tools are not able to find this as they probably do not use concrete unfolding combined with unification.

Chapter 11

Future Work

In future work, many other techniques for proving non-termination could be adapted for many-sorted term rewriting systems, most notably the Dependency Pair Framework. The work in this thesis could be translated into a DP processor. It could also be interesting to look at existing termination techniques and adapt them for many-sorted TRSs.

A second class of rewriting systems to look at, are the string rewriting systems. It could be interesting to look at how sorts impact the (non-)termination analysis on them.

Of course, the current implementations of the algorithms could also be improved. The implementation for semi-unification is worst case exponential, while polynomial time algorithms exist[16][24]. These have not been used here for the sake of complexity. Also, the abstract unfolding analysis has gotten an improvement in [26] which could be implemented alongside the existing concrete and abstract unfolders.

Bibliography

[1] Cops: Confluence problems database. http://cops.uibk.ac.at/. Accessed: 2019-12-11.

[2] T. Arts and J Giesl. Termination of term rewriting using dependency pairs. *Theoretical computer science*, 236(1-2):133–178, 2000.

[3] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137 – 159, 1987.

[4] S. R. Buss. Chapter i - an introduction to proof theory. In *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 1 – 78. 1998.

[5] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279 – 301, 1982.

[6] F. Emmes, T. Enger, and J. Giesl. Detecting non-looping non-termination. In *12th International Workshop on Termination (WST 2012)*, page 49, 2012.

[7] J. Endrullis and H. Zantema. Proving non-termination by finite automata. In *26th International Conference on Rewriting Techniques and Applications, RTA 2015*, volume 36, pages 160–176, 2015.

[8] Blanqui F., Jouannaud J.P., and Rubio A. The computability path ordering. *Logical Methods in Computer Science*, 11(4), 2015.

[9] C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. *Leibniz International Proceedings in Informatics, LIPIcs*, 15, 2012.

[10] J. Giesl. Termination portal. http://termination-portal.org/wiki/. Accessed: 2019-11-20.

[11] 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, *Automated Reasoning*, pages 281–286, 2006.

[12] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of the 5th International Conference on Frontiers of Combining Systems*, FroCoS'05, pages 216–231, 2005.

[13] F Henglein. Type inference and semi-unification. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 184–197, 1988.

[14] A. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474 – 511, 2007.

[15] N. Hirokawa and A. Middeldorp. Polynomial interpretations with negative coefficients. In B. Buchberger and J. Campbell, editors, *Artificial Intelligence and Symbolic Computation*, pages 185–198, 2004.

[16] D. Kapur, D. Musser, P. Narendran, and J. Stillman. Semi-unification. In K. V. Nori and S. Kumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, 1988.

[17] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Information and Computation*, 102(1):83 – 101, 1993.

[18] J. W. Klop. Handbook of logic in computer science (vol. 2). In *Handbook of Logic in Computer Science (Vol. 2)*, chapter Term Rewriting Systems, pages 1–116. 1992.

[19] C. Kop. Wanda: A higher-order termination tool. `http://wandahot.sourceforge.net/`. Accessed: 2019-11-20.

[20] C. Kop. *Higher Order Termination: Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2012.

[21] C. Kop. Cora. `https://github.com/hezzel/cora`, 2019.

[22] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In R. Treinen, editor, *Rewriting Techniques and Applications*, pages 295–304, 2009.

[23] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[24] A. Oliart and W. Snyder. Fast algorithms for uniform semi-unification. *Journal of Symbolic Computation*, 37(4):455 – 484, 2004.

[25] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[26] E. Payet. Non-terminatnion inference. http://lim.univ-reunion.fr/staff/epayet/Research/NTI/NTI.html. Accessed: 2019-11-20.

[27] E. Payet. Detecting non-termination of term rewriting systems using an unfolding operator. In *Logic-Based Program Synthesis and Transformation*, pages 194–209, 2006.

[28] E. Payet. Guided unfoldings for finding loops in standard term rewriting, 2018.

[29] R. Thiemann. *The DP framework for proving termination of term rewriting*. PhD thesis, RWTH, Department of Computer Science, 2007.

[30] Jaco van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, Universiteit Utrecht, 1996.

[31] A. Yamada, K. Kusakari, and T. Sakabe. Nagoya termination tool. In *RTA-TLCA*, 2014.

[32] T. Yoshihito. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141 – 143, 1987.

[33] H. Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34:105–139, 01 2005.

Appendix A

Test Framework

The test framework created and used is written in Python and can be found at `https://github.com/bornobob/MaraRunner`. It consists of two parts: the runner and the visualiser. The runner takes configurations and tests files, then outputs a JSON file with all the results. The visualiser creates an HTML page with all the results in a table, where the different configurations are split up in columns.

Appendix B

Results

The results from all the experiments were converted to HTML files using the previously mentioned test framework. There are four files: one for the experiments on the termination problem database, two for the confluence database and the last one is for the problems transformed from the higher order database.