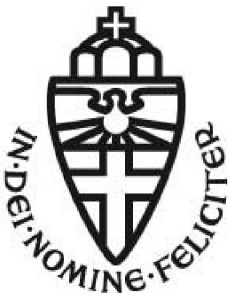


Radboud Universiteit Nijmegen

BACHELOR THESIS

FORMAL CHARACTERISTICS OF THE FUNCTIONAL STRATEGY

Rafael Alejandro Imamgiller

Supervisor/First assessor:
prof. dr. Erik Barendsen

Second assessor:
dr. Sjaak Smetsers

August 2014

Abstract

The functional method of evaluation, also known as the functional strategy, is an algorithm to perform calculations in functional languages. It is efficient and intuitively easy to understand, which makes it a great strategy for functional language interpreters. But, because the algorithm deals with priority rewriting, its semantics are complicated and difficult to formalize, which makes it hard to use as a formal strategy for term rewriting systems. In this paper we look at the formal characteristics of the functional strategy by comparing two different formalization approaches, and determine its normalizing behaviour.

Contents

Chapter 1

Introduction

Functional programming is a branch of programming that closely resembles actual mathematics. It is based on the formal construct of the Term Rewriting Systems or TRSes (as for instance described by J.W. Klop in *Term Rewriting Systems* (1992) [1]), a way of defining mathematical operations by *rewriting* them to simpler forms until the simplest form, or *normal form*, is reached. The normal form of an equation is the result of calculating that equation.

A computer algorithm trying to perform calculations in a TRS therefore constantly rewrites equations to simpler forms using these definitions, until it reaches this normal form and finishes the calculation. One such algorithm is the Functional Strategy. It approaches the normal form through simple rewriting steps and a concept called *lazy evaluation* that circumvents certain problems with undecidability.

The main problem of the Functional Strategy is that, while easy to understand intuitively, its semantics are hard to formalize. Toyama et al. did this in *The Functional Strategy and Transitive Term Rewriting Systems* [3] by designing an algorithm based on *index rewriting* and proving it was normalizing for the class of *left-incompatible term rewriting systems*.

The question we try to answer in this paper is as follows:

RQ: What are the formal characteristics of the Functional Strategy?

We break this question down into two subquestions:

SQ1: How can we formalize the Functional Strategy?

SQ2: How can we describe the normalizing behaviour of the Functional Strategy?

We answer the first subquestion in Chapter 2 and 3 by comparing a refinement of the definition from [2] to the definition from [3]. We answer

the second subquestion in Chapter 3 by exploring the concept of *transitive term rewriting systems* and *left-incompatibility*. Finally, we will show that the algorithm suggested by Toyama et al. [3] is an actual implementation of the Functional Strategy.

But before we can formalize the Functional Strategy, we need to do some preliminary work. In this chapter we introduce the concepts of Term Rewriting Systems, Reduction Strategies, Normal Form and Head-Normal Form. We also introduce a special type of tree diagram that we use extensively to illustrate examples throughout this paper.

1.1 Term Rewriting

In term rewriting such as described in *Term Rewriting Systems* [1], functions are defined recursively by distinguishing several different cases for different *patterns* of arguments.

In the example below, $s(x)$ indicates the *successor* of x: $s(x) = x + 1$. So every number $n \in \mathbb{N}$ can be written as $s^n(0)$.

1.1 Example: Addition defined recursively:

1. $Add(x, 0) \to x$
2. $Add(x, s(y)) \to Add(s(x), y)$

1.2 Example: Multiplication defined recursively:

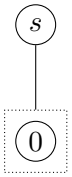
1. $Mult(x, 0) \to 0$
2. $Mult(x, s(y)) \to Add(Mult(x, y), x)$

Every single one of these cases is a *rewrite rule*. Together they define how a given expression or *term* can be *reduced* or *rewritten* to a different term. We use a right-headed arrow ($\to$) instead of an equality symbol to signify that a rewrite rule rewrites in only one direction. When no rewrite rule can rewrite a term, that term is in *normal form*.

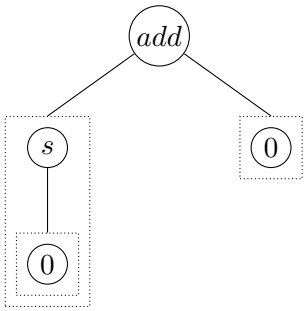
Terms have a tree-like structure: the arguments of a function symbol in a term are terms of their own. We can express this in a tree diagram. We will now show a few examples of this.

1.3 Example: The constant 0 without arguments or proper subterms.

1.4 Example: The term $s(0)$, called with 0 as its first and only argument. 0 is a proper subterm of $s(0)$ here.



1.5 Example: The term $add(s(0), 0)$, function add called with arguments $s(0)$ and 0. Note that 0 here is a proper subterm of both $s(0)$ and $add(h(0), 0)$, since 0 is in an argument space for both s and add. The term $s(0)$ is only a proper subterm of $add(s(0), 0)$.



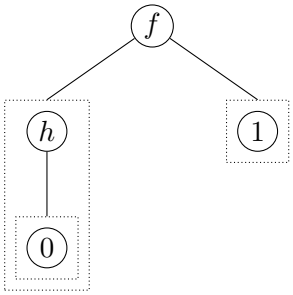
Note the use of dotted boxes in these diagrams. These express the *argument spaces* of a function symbol, the possible locations for another term to be inserted into it. Terms inside dotted boxes are inserted into the corresponding argument spaces of other terms; they are *proper subterms*.

We can generalize by using generic function letters such as f to look at a few interesting properties:

1.6 Example: Two generic functions defined recursively:

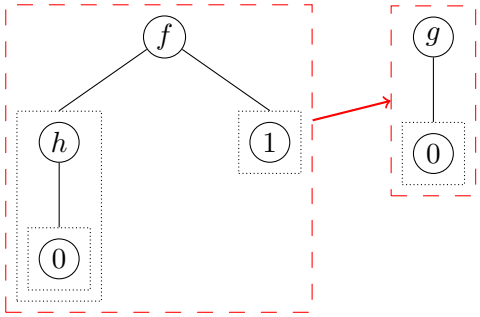
1. $f(h(x), y) \to g(x)$
2. $h(x) \to j(x)$

1.7 Example: The term $f(h(0), 1)$ is similar to $add(s(0), 0)$ in basic structure, but has different symbols in several places.

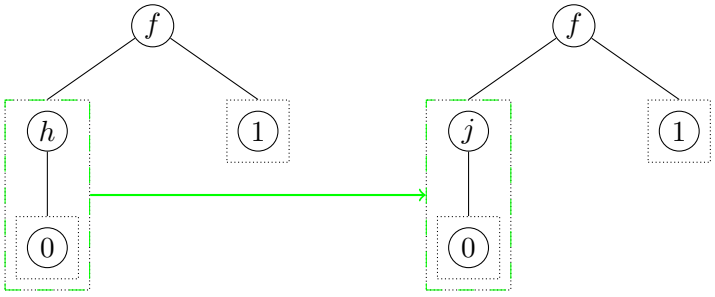


As long as a (sub)term matches the left-hand side of the rewrite rule, that (sub)term can be reduced using that rule. The result is simply inserted back at the same argument space.

Now say we want to know what we get if we rewrite $f(h(0), 1)$ using Example 1.6. To better illustrate what happens, we put colored boxes around the (sub)terms that are important in our examples. If we use the first rewrite rule, we get the following reduction: $\boxed{f(h(0), 1)} \rightarrow \boxed{g(0)}$.



But if we apply the second rule on $f(h(0), 1)$, instead this happens: $f(\boxed{h(0)}, 1) \rightarrow f(\boxed{j(0)}, 1)$.



Neither of these terms can be rewritten any further; they are both in *normal form*. But both were obtained by rewriting $f(h(0), 1)$, using different rewrite rules. So the rewrite rule we choose to rewrite the term with is of importance. We call a series of reductions a *reduction path*.

A function that takes a TRS and a term in that TRS as input, and gives another term as output such that a reduction path exists between the input and output terms, is called a *reduction strategy*.

If a strategy always yields normal forms for a certain set of TRSes, it is said to be *normalizing* on that set.

As we have seen, s does not need its own rewrite rules; in fact our definitions of Add and $Mult$ depend on every number being of the form $s^n(0)$. We call a function symbol without rewrite rules (such as s) a *constructor*.

A Term Rewriting System or TRS $\mathbf{R}$ is a tuple of an alphabet Σ of function symbols, constants and variables, and a set of rewrite rules R consisting

of terms constructed from that alphabet. $\mathbf{R} = (\Sigma, R)$. So each TRS is a set of rewrite rules over a set of terms. *A TRS is defined by its alphabet and its rewrite rules.*

In Example 1.1, the alphabet consists of the functions *Add* and s, the constant 0 and a countably infinite set of variables $x, y, z, ...$ The set of rewrite rules consists of the rules as listed in that example. By adding the function $Mult$ and the rewrite rules from Example 1.2 we get another TRS, one capable of both addition and multiplication.

1.2 Formal Definitions

In the previous section we saw Term Rewriting Systems, a method to define mathematical functions through simple rules, and then use those rules to perform calculations (using a strategy to rewrite a term to NF). Now we give very concise, formal definitions of TRSes and reduction strategies here:

1.2.1 Term Rewriting Systems

1.8 Definition (Term Rewriting System (TRS)): A TRS $\mathbf{R}$ is a pair (Σ, R) of an alphabet/signature Σ and a set of rewrite rules R.

- The set $T(\Sigma)$ is the set of terms over Σ, i.e. every term that is a variable, a constant or a function of which every argument is also a term.
- A *closed term* is a term that does not contain any variables.
- A rewrite rule $\mathbf{r} \in R$ is a pair (l, r) of terms $l, r \in T(\Sigma)$ such that l is not a variable, and all variables in r are contained in l. If rule $\mathbf{r} = (l, r)$, it is written as $\mathbf{r} : l \to r$.
- Let $t, s \in T(\Sigma)$. Then t *reduces* to s (notation $t \twoheadrightarrow s$) if t can rewrite to s in zero or more reduction steps. Thus, $\twoheadrightarrow$ is the transitive reflexive closure of $\to$. (So $t \twoheadrightarrow s$ is true even if $t = s$.)
- $t \to^+ s$ if t reduces to s in *one or more* reduction steps.

1.9 Example: Let $\mathbf{R} = (\Sigma, R)$ with $\Sigma = (\{f, g, 0, 1\}, V)$ (where V is a countably infinite set of variables) and $R = \{\mathbf{r}_1\}$ with $\mathbf{r}_1 = (f(x, y), g(x))$. Then $f(0, 1)$ is a term of $\mathbf{R}$.

1.2.2 Reduction Strategy

1.10 Definition (Reduction Strategy): Let $\mathbf{R} = (\Sigma, R)$ be a TRS and $t \in \mathbf{R}$ a term.

1. A reduction strategy $\mathbf{S}$ for $\mathbf{R}$ is a map from $T(\Sigma)$ to $T(\Sigma)$. The result of applying $\mathbf{S}$ to $\mathbf{R}$ and t is written as $\mathbf{S}_{\mathbf{R}}(t)$.

2. $\mathbf{S}$ is only a strategy if $t \twoheadrightarrow \mathbf{S}(\mathbf{R}, t)$ holds.

3. *Repeated application* of $\mathbf{S}$ on $\mathbf{R}$ and t is written as ${\mathbf{S}_{\mathbf{R}}}^{i}(t)$, where i is the number of repeats, and defined as follows:

   (a) ${\mathbf{S}_{\mathbf{R}}}^{0}(t) = t$.

   (b) ${\mathbf{S}_{\mathbf{R}}}^{n}(t) = \mathbf{S}_{\mathbf{R}}({\mathbf{S}_{\mathbf{R}}}^{n-1}(t))$.

4. For every t in $\mathbf{R}$, iff t has a NF, there exists an $i \in \mathbb{N}$ such that ${\mathbf{S}_{\mathbf{R}}}^{i}(t) = s$ with s in NF. If this holds, $\mathbf{S}$ is *normalizing* on $\mathbf{R}$.

5. $\mathbf{S}$ is normalizing on a *set* of TRSes if it is normalizing on every TRS in that set.

6. Let $\mathbf{S}$ and $\mathbf{Q}$ be two reduction strategies. If $\mathbf{S}_{\mathbf{R}}(t) \equiv \mathbf{Q}_{\mathbf{R}}(t)$ holds for every TRS $\mathbf{R} = (\Sigma, R)$ and term $t \in T(\Sigma)$ in a set, then $\mathbf{S}$ and $\mathbf{Q}$ are equivalent for that set (notation: $\mathbf{S} \equiv \mathbf{Q}$).

1.2.3 Formalized rewriting

A hole $\Box$ is a special constant that indicates an "open spot"in a term. A context $C[\,,\ldots,\,] = t$ lists all of these spots in t from left to right. If a spot in the context is empty, the location in t it corresponds with contains $\Box$. But by putting a term s at a spot in the context, it replaces that hole in t.

1.11 Example:

$$C[\ ,\ ] = f(\Box, \Box)$$

Then

$$C[\ , s] = f(\Box, s)$$

and

$$C[q, u] = f(q, u)$$

1.12 Definition (Holes, Contexts and Subterms): We introduce a new constant $\Box$ called a *hole*. Then $C \in T(\Sigma \cup \{\Box\})$ is a *context*. We use the notation $C[\,,\ldots,\,]$ for a context with $n \geq 1$ holes. If $t_1, \ldots, t_n \in T(\Sigma)$ then $C[t_1, \ldots, t_n]$ is the result of placing those terms in the holes of $C[\,,\ldots,\,]$ from left to right.

The variable occurrence z in $C[z]$ is *fresh* if $z \notin C[\,]$.

Now we can introduce a new definition for subterm: if $t \equiv C[s]$ then s is a *subterm* of t, written $s \subseteq t$. If also $t \not\equiv s$ then s is a *proper subterm* of t, written $s \subset t$. *Every term is a (non-proper) subterm of itself.*

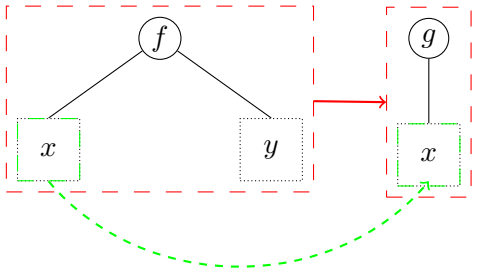
1.13 Definition (Substitution and Redex): A substitution σ is a map from $T(\Sigma)$ to $T(\Sigma)$ satisfying:

$$\sigma(f(t_1, ..., t_n)) \equiv f(\sigma(t_1), ..., \sigma(t_n))$$

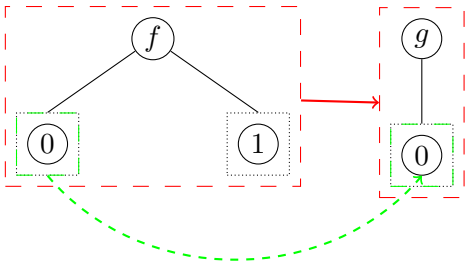
for every n-ary function symbol f. It is also possible to write t^σ instead of $\sigma(t)$. Note that the substitution skips function symbols; substitutions in term rewriting are determined entirely on variables.

Term t can be rewritten to term s ($t \to s$) if there exists a rule $\mathbf{r} : l \to r$ and a substitution σ such that $t = C[l^\sigma]$ and $s = C[r^\sigma]$. The term l^σ is called a *redex*.

1.14 Example: Rule $\mathbf{r}_1$ from Example 1.9. The green boxes show the variable that is carried over during the reduction.



Now if we want to use $\mathbf{r}_1$ to rewrite $f(0, 1)$, we substitute 0 for x (as indicated by the green boxes), and 1 for y.



Thus $f(0, 1)$ rewrites to $g(0)$ with $\mathbf{r}_1$.

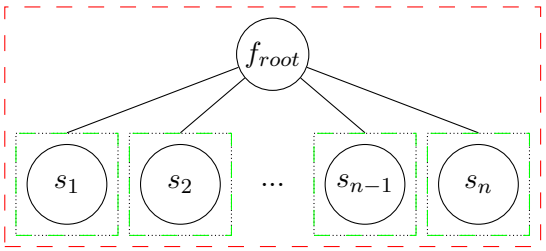
1.2.4 Normal Form and Head-Normal Form

1.15 Definition (Root): The root of a (non-variable) term is the function symbol at the head of the term. $root(f(p_1, ..., p_n)) = f$

1.16 Definition (Normal Form and Head-Normal Form): A term t in a TRS T is said to be in Normal Form (NF) if it can not be rewritten any further. In that case there exists no $s \in T(\Sigma)$ such that $t \to s$.. A term t in a TRS T is said to be in Head-Normal Form (HNF) if there exists no

redex that it can be rewritten to. In that case there exists no $s \in T(\Sigma)$ such that $t \twoheadrightarrow s$ and s is a redex. This means that its root can no longer be rewritten; proper subterms may still be able to be rewritten, making it a weaker constraint than NF.

1.17 Example: A generic term $t = f_{root}(s_1, s_2, ..., s_{n-1}, s_n)$ such that t is in HNF but $s_1, ..., s_n$ are not.



The red box, containing the whole term, is in HNF and therefore no rewrite rule will ever fit it again. The green boxes, containing the proper subterms, do not have to be in HNF and may still be rewritten.

Chapter 2

The Functional Strategy

In this chapter we describe the Functional Strategy. We start by proving a lemma that lets us reach NF step-by-step and use it to construct the Naive Strategy. We expand on this through a concept called *lazy evaluation* to obtain the Functional Strategy. Finally, we give a broad definition of the **FS** in natural language and a more specific one in pseudo-code called **PC-FS**.

2.1 Solving TRSes

2.1.1 Reaching NF through the Head-Normal Form

Head-Normal Form is an interesting property. We saw in Definition 1.13 that a redex is a term that matches the left-hand side of a rewrite rule (with a substitution for the variables). This in conjunction with Definition 1.16 means that a term in HNF is not a redex, and its root will no longer change. What if we know that every subterm is in Head-Normal Form?

2.1 Lemma: If every $s \subseteq t$ is in HNF, t is in NF.

Proof. With induction to the depth of the subterms of t.

Base: If t is a constant, it has no proper subterms and t is its only subterm. Because t is in HNF, none of its subterms are redexes.

Step: The proper subterms of t are in NF and therefore not redexes (IH), and because t is in HNF it isn't a redex either.

Conclusion: t is in NF. □

By determining if (sub)terms are in HNF, we can work toward reaching NF. Unfortunately, it's not as simple as just trying all rewrite rules and calling HNF when none fit. If we evaluate the proper subterms, it's still possible that the term rewrites to a redex, which means that it was not in HNF. Look at Example 1.17 again. Say that, by rewriting s_1 with a certain

rewrite rule, we get a s_1' such that the red box suddenly fits another rewrite rule. That means the red box was never in HNF to begin with.

This means we cannot determine whether a term is in HNF until we look at these problematic redexes that "obstruct"us. If a proper subterm is a redex that has to be evaluated before we can determine if a term is in (H)NF, we also call that subterm an *index*. We will look more in-depth at this in Definition 3.11 of Chapter 3.

As such we *force evaluation* of the redex first until it reaches its NF. The problem is determining what are the redexes. By forcing the evaluation of *every* proper subterm we can solve this, but then we run into the same problem again by having to evaluate the redexes of this proper subterm first. That means we have to force evaluation of every proper subterm contained in it, et cetera, until we reach terms without proper subterms. Thus we get a recursive call of our strategy on every proper subterm.

We also have to pick an order in which to evaluate them since we are designing a computer algorithm. For our strategy, we will choose *leftmost-outermost* order: we start with the leftmost-outermost proper subterm, and force its evaluation to NF. We call this strategy the *leftmost-outermost strategy* or **LOS**.

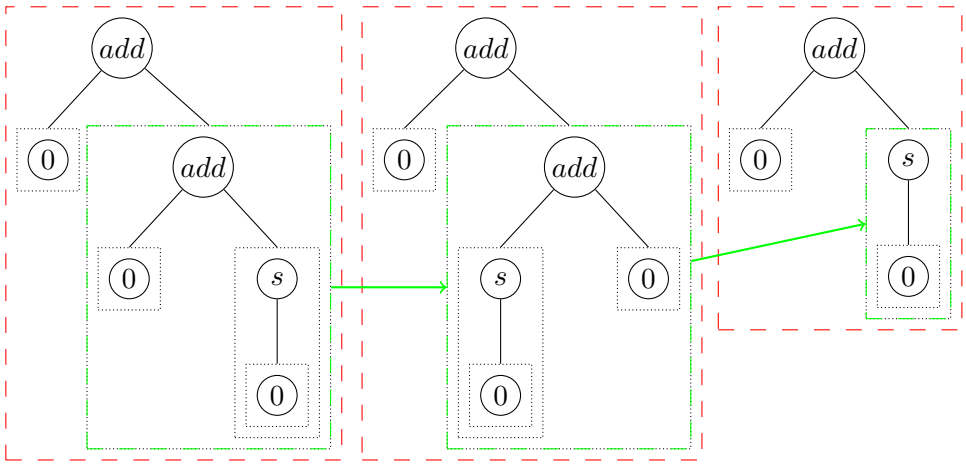
2.2 Example: Recall the TRS from Example 1.1. It had the following rewrite rules:

$$\mathbf{r}_1 : add(x, 0) \to x$$

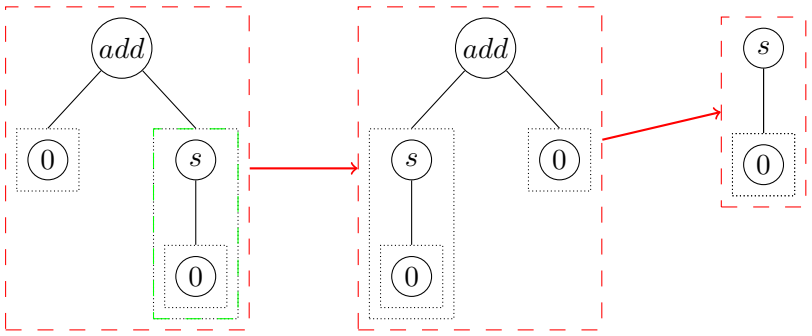
$$\mathbf{r}_2 : add(x, s(y)) \to add(s(x), y)$$

We want to rewrite the term $add(0, add(0, s(0)))$ to its normal form. We try to apply $\mathbf{r}_1$ on the red frame, which means we have to evaluate its proper subterms first. 0 is already in NF and evaluation will just return 0 again, but $add(0, s(0))$, the green frame, is a redex. We again try to apply $\mathbf{r}_1$, this time to the green frame. Both its subterms, 0 and $s(0)$ are in NF, but $\mathbf{r}_1$ does not fit, so we try $\mathbf{r}_2$ instead. This rewrite rule does fit, and rewrites $add(0, s(0))$ to $add(s(0), 0)$.

We start over, trying to apply $\mathbf{r}_1$ to the green frame. Again the subterms are in NF and the rule fits this time, rewriting the term to $s(0)$. The green frame is now in NF, so we finish rewriting it. The reduction of the green frame looks like this in a diagram:



Now we go back to the red frame. Unfortunately $\mathbf{r}_1$ does not fit it. This means we have to try the next rewrite rule, $\mathbf{r}_2$. $\mathbf{r}_2$ does fit now and rewrites the red frame to $add(s(0), 0)$. Finally we apply $\mathbf{r}_1$ one last time to reach $s(0)$, the NF of the full term. The reduction of the red frame after reducing the green frame looks like this in a diagram:



The only problem is that this assumes that the term has a (H)NF to begin with. But it is possible for a TRS to have a reduction path that never terminates, such as a *cycle*.

2.3 Example: Let $\mathbf{R}$ be a TRS with the rewrite rules $\mathbf{r}_1 : f(x) \to g$ and $\mathbf{r}_\omega : \omega \to \omega$. If we try to rewrite the term $t = f(\omega)$ in this TRS, we can only use $\mathbf{r}_1$. But under our old system, this means we have to evaluate the redex ω first. $\omega \to \omega$ and therefore $f(\omega) \to f(\omega)$. Because ω is not in NF yet, we have to evaluate it again and again, endlessly, resulting in non-termination, even though we could have used the first rewrite rule to rewrite directly to g.

This presents us with a challenge. It is clear that Lemma 2.1 could be used to construct a reduction strategy that is normalizing on some TRSes. However, due to the existence of non-terminating reduction paths, HNF is

undecidable. The only way to determine if a term is in HNF is by exhausting every possible rewrite rule, which involves evaluating the term and its subterms, which in turn may get stuck in a non-terminating reduction path. As a result **LOS** and any other strategy based on Lemma 2.1 would only an *approximation*, and **LOS** is a rather poor one, being completely incapable of dealing with non-termination.

Fortunately there is a solution to this problem that'll let us avoid these non-terminating subterms if they disappear from the term in a rewrite step.

2.1.2 Lazy Evaluation

In **NS** we always force the evaluation of subterms. We saw that this causes some problems in Example 2.3, where **LOS** is not normalizing despite the term having a normal form. The reason for this is that the subterm that causes nontermination is not carried over in the rewrite step. It has to match a variable that disappears in the right hand side of the rewrite rule.

We can solve this problem by not immediately forcing evaluation if a subterm has to match a variable. We call this *lazy evaluation*. By performing lazy evaluation in **LOS** we end up with the *functional strategy* or **FS**.

2.4 Example: We take another look at $\mathbf{R}$ from Example 2.3 with the rewrite rules $\mathbf{r}_1 : f(x) \to g$ and $\mathbf{r}_\omega : \omega \to \omega$. Again we try to rewrite the term $f(\omega)$ in this TRS with $\mathbf{r}_1$. But because we use *lazy evaluation* this time and ω has to match a variable, we can apply $\mathbf{r}_1$ directly. This rule rewrites $f(\omega)$ to g, and g is in NF.

But we encounter a new problem. When a variable does appear in the right-hand side of a rewrite rule, the subterm that we skipped evaluation on will remain in the term after rewriting, without being evaluated. Thus we may end up with a term that is not in NF.

2.5 Example: Let $\mathbf{R}$ be a TRS (Σ, R) with the rewrite rules $\mathbf{r}_1 : f(x) \to g(x)$ and $\mathbf{r}_2 : a \to b$. We want to rewrite $f(a)$. Using lazy evaluation, we use $\mathbf{r}_1$ to rewrite $f(a)$ to $g(a)$. Our strategy terminates here.

Of course, such a subterm may be a redex or contain redex. Solving the problem is as simple as just repeating **FS** again on these subterms.

2.6 Example: We continue from Example 2.5 by applying our strategy on the redex a. We use $\mathbf{r}_2$ to rewrite $g(a)$ to $g(b)$. Our strategy terminates again, and this time the whole term is in NF.

Remember that, even after rewriting, the **FS** has to run all rewrite rules on a term again before it can decide that term is in (H)NF. This means that, even if a proper subterm had to match a variable and was ignored at first, if it has to match a non-variable at any point during this evaluation it'll get evaluated regardless. This also means that, if the **FS** terminates now, evaluating the proper subterms did not rewrite the term to a redex. Else the term would have gotten rewritten further. *When we skip evaluation of a*

proper subterm in Lazy Evaluation and that proper subterm remains behind in the term after **FS** *terminates, evaluating that proper subterm will never rewrite the term containing it to a redex.*

2.7 LEMMA: Let $\mathbf{R}$ be a TRS and $t \in \mathbf{R}$. Say that $\mathbf{FS_R}(t) = t'$ such that $t' = C[s_1, ..., s_n]$ with $s_1, ..., s_n \notin NF$. Then $t' \in$ HNF.

Proof. Say that $t' \notin$ HNF. Then there exists a reduction path such that $t' \twoheadrightarrow t''$ with t'' a redex. This means evaluation of the proper subterms $s_1, ..., s_n$ of t' rewrites it to a redex. But **FS** evaluated the proper subterms when it tried to match t' with the rewrite rules. Therefore, if this was the case, $\mathbf{FS_R}(t) = t''$. We know that $\mathbf{FS_R}(t) = t'$, so $t' = t''$. But we also know that t' is not a redex, so we have a contradiction. Conclusion: $t' \in$ HNF. □

2.1.3 Rule Order and Overlap

As we saw in Definition 1.8, R is defined as a *set* of rewrite rules. A set has many possible orders. However, a computer algorithm has to try the rewrite rules one by one, so it will have to settle for one order. But does the chosen order really matter?

2.8 EXAMPLE: Let $\mathbf{R}$ be a TRS with the following rewrite rules:

$\mathbf{r}_1 : min(x, 0) \to 0$
$\mathbf{r}_2 : min(0, y) \to 0$
$\mathbf{r}_3 : min(s(x), s(y)) \to s(min(x, y))$
$\mathbf{r}_\omega : \omega \to \omega$

We want to rewrite $t_1 = min(\omega, 0)$. We apply **FS** on the term, and start by trying to match it with $\mathbf{r}_1$. The subterm ω has to match x, a variable, so its evaluation is skipped. The subterm 0 has to match 0, a non-variable, so its evaluation is forced. However, this evaluation quickly terminates, because 0 is already in NF. This concludes the match, and we use $\mathbf{r}_1$ to rewrite t_1 to 0.

Now we want to rewrite $t_2 = min(0, \omega)$. We apply **FS** on the term, and start by trying to match it with $\mathbf{r}_1$. The subterm 0 has to match x, a variable, so its evaluation is skipped. But ω has to match 0, a non-variable, so we force its evaluation. However, ω always rewrites to ω, so we get stuck in a non-terminating reduction path.

Let's switch the order in which we try the rewrite rules and start with $\mathbf{r}_2$ instead. First 0 has to match 0. 0 is already in NF and it does match, so we move on to the next subterm, ω. This time ω has to match y, a variable, so we skip its evaluation. This concludes the matching, and we use the rewrite rule to rewrite t_2 to 0.

But what if we had tried to match t_1 with $\mathbf{r}_2$? The first subterm, ω, has to match the non-variable 0 and its evaluation is forced. Thus we get stuck in a non-terminating reduction path again.

We saw two different terms in the same TRS in the above example. The order in which the rewrite rules were listed determined on which one **FS** was normalizing. But what exactly causes this problem? It's because of *overlap*.

2.9 Definition (Overlap): Two terms t and s are *overlapping* if there exist substitutions σ_1 and σ_2 such that $t^{\sigma_1} \equiv s^{\sigma_2}$.

In Example 2.8, $\mathbf{r}_1$ and $\mathbf{r}_2$ overlap, thus a term may fit both of them. Of course, in the end $min(\omega, 0)$ and $min(0, \omega)$ will both only fit one of them, but the algorithm cannot figure this out because ω is non-terminating. There are two ways to circumvent this problem. One way is to completely avoid overlap and assume that every TRS we work with is *orthogonal*:

2.10 Definition (Orthogonality): A term is *linear* if every variable in it occurs only once. A TRS R is *orthogonal* if:

1. For all rewrite rules $\mathbf{r} : l \to r \in R$, l is linear.
2. For any two rewrite rules $\mathbf{r}_1 : l_1 \to r_1$ and $\mathbf{r}_2 : l_2 \to r_2 \in R$:
   (a) If $\mathbf{r}_1$ and $\mathbf{r}_2$ are different, then l_1 and l_2 are non-overlapping.
   (b) For all $s \subset l_2$ such that s is not a single variable, l_1 and s are non-overlapping.

Unfortunately, orthogonality is a very strict property and severely limits the TRSes we can use.

2.11 Example: A generic function with overlap.
$\mathbf{r}_1 : f(x, y) \to g$
$\mathbf{r}_2 : f(x, 0) \to h$
$\mathbf{r}_\omega : \omega \to \omega$
We want to solve $f(\omega, \omega)$. When we apply **FS** and try $\mathbf{r}_2$ first, we get stuck on the subterm ω, which is evaluated because it has to match 0. If we try $\mathbf{r}_1$ first, ω has to match the variable y instead and evaluation is skipped, reducing the term to its normal form h.

Again the order of the rewrite rules matters. But this time, by trying $\mathbf{r}_1$ first, we try to match all terms with variables before non-variables, unlike in Example 2.8, where there is no order such that a subterm is always matched with a variable before a non-variable. The TRS in Example 2.11 is not orthogonal, but **FS** is normalizing on it if the rewrite rules are tried in the right order.

2.12 Definition (Order): Let $\mathbf{R} = (\Sigma, R)$ be a TRS with $R = \{\mathbf{r}_1, ..., \mathbf{r}_n\}$. Let $\mathcal{R} : 1, ..., n \to R$ be a bijection on R. Then $\mathcal{R}$ is an *order* of $\mathbf{R}$, and $\mathcal{R}(i)$ is the ith rule in that order.

But the left-hand side of the first rewrite rule is $f(x, y)$. Two variables mean that regardless of the subterms, terms will always fit this rewrite rule before the next one, $f(x, 0)$. This form of overlap makes the second rewrite

rule completely useless if the first rewrite rule is tried first, which means that as far as **FS** is concerned, the second rule does not exist. This would also eliminate the overlap issue. That's why we will stick with orthogonal TRSes for the scope of this paper.

Even in an orthogonal TRS rule order can be important. Look at this TRS:

2.13 Example: An orthogonal TRS in which the rule order determines the normalizing behaviour:

$\mathbf{r}_1 : f(x, 0) \to g$
$\mathbf{r}_2 : f(1, 1) \to h$
$\mathbf{r}_\omega : \omega \to \omega$

If we want to solve $f(\omega, 0)$ and we try $\mathbf{r}_2$ first, we get stuck in a cycle. But if we try $\mathbf{r}_1$ first, we immediately rewrite to the normal form of the term, g. In Chapter 3 we show that this TRS is *left-incompatible*, and that the **FS** is normalizing on the class of *left-incompatible TRSes*, a subclass of the orthogonal TRSes. For now we will stick with just the assumption that all our TRSes are orthogonal.

2.1.4 Describing the FS in natural language

How do we use the properties we have specified so far to design the Functional Strategy? Let's look at the description given in *Functional Programming and Parallel Graph Rewriting* [2].

2.14 Definition (**FS**): The Functional Strategy defined intuitively in natural language:

1. If there are several rewrite rules for a particular function, the rules are tried in *textual order*.

2. Patterns are tested from left to right.

3. Evaluation of an actual argument is always forced when this argument must match a non-variable in the corresponding pattern (even in overlapping cases).

The **FS** as it is given here is an algorithm that tries to match a term with rewrite rules by going over the subterms one by one, and force the evaluation of those subterms when they have to match a non-variable. This 'forced evaluation' is actually a recursive call of the **FS** with this subterm. This means that the end result of the forced evaluation should be a term in HNF. We proved in Lemma 2.1 that, by repeating this until the **FS** cannot rewrite this subterm any further, it is in NF. Thus the **FS** recursively rewrites the term to NF. This definition also implements lazy evaluation to be able to find the HNF if one exists.

In addition, every run of the **FS** is a multi-step strategy. As we have seen before, the **FS** will rewrite orthogonal systems to HNF in a single run, and therefore by Lemma 2.1 is normalizing if repeated on every subterm.

The problem with Definition 2.14 is that it is a bit too vague and concise to be useful in formalization. A lot of the actual procedures are left to the reader's imagination. Therefore we will give a more precise definition in pseudo-code that describes an actual algorithm and call it **PC-FS**.

2.2 Formalizing the FS as an algorithm

Definition 2.14 is a bit too vague to really be considered formal. We have to expand this definition, clarifying what happens at each step and interpreting some of the words used

2.2.1 The FS defined in pseudo code (PC-FS)

2.15 Definition (**PC-FS)**: The Functional Strategy defined as an algorithm in pseudo-code, taking a TRS $\mathbf{R} = (R, \Sigma)$ with textual order $\mathcal{R}$ and a term $t \in T(\Sigma)$ as input.

1. Make a subset R' of all rules $\mathbf{r} : l \to r$ of which the root of l matches the root of t'.

2. For $i \geq 1$ and $i \leq \#R$ in leftmost-outermost order:

   (a) If $\mathcal{R}(i) \in R'$ take $\mathbf{r} : l \to r = \mathcal{R}(i)$ as our next rewrite rule. Else skip to the next i.

   (b) We know that $root(t) = root(l)$. Let this root be f. Then $t = f(s_1, ..., s_n)$ and $l = f(l_1, ..., l_n)$. For $j \geq 1$ and $j \leq n$:

      i. If l_j is a variable, let $s'_j = s_j$ and skip to the next j.

      ii. If l_j is not a variable, let $s'_j = \mathbf{PC\text{-}FS_R}(s_j)$ with order $\mathcal{R}$.

      iii. If $root(s'_j) = root(l_j)$, recursively repeat (b) with s'_j as t and l_j for l before continuing with the next j, and don't go to (c) if this run is succesful but instead continue here. This lets us verify that all proper subterms also match in a depth-first way. If at any point in these recursive calls the roots do not match, completely stop the evaluation of the term with this rule, go back to (a) and skip to the next rule.

      iv. Go back to (i) for the next j. If there is no next j, take $t = f(s'_1, ..., s'_n)$ and go to (c).

   (c) Only advance to this step if the run of (b) that just finished was not a recursive call by (iii). Else go back to the iteration of (iii) that caused this run of (b). Rewrite t with $\mathbf{r}$ and go back to (1).

3. If there are no rules left, terminate with "t is in Head-Normal Form".

But is this really a good definition? Remember that HNF is undecidable, so we have to proof the following: *if* **PC-FS** returns a term, that term is in HNF.

2.16 Lemma: If $\mathbf{PC\text{-}FS}_{\mathbf{R}}(t) = s$, then s in HNF.

Proof. With induction to the size of t.

Base: Let t be in HNF. Recall Definition 1.16. If t is in HNF, there exists no redex s such that $t \twoheadrightarrow s$. No rewrite rule will ever match t, regardless of what its proper subterms rewrite to. So (iii) will run into a pair of subterms whose roots do not match for every rewrite rule. Thus t will not be rewritten, and the output of **PC-FS** is t itself. Therefore, if $\mathbf{PC\text{-}FS}(\mathbf{R}, t) = s$ for t in HNF, then $t = s$ and s in HNF.

Step: Let t not be in HNF. Say that $\mathbf{r}_i$ is the rewrite rule that will (eventually) fit. Evaluating the subterms in (b) causes the reduction path $t \twoheadrightarrow t'$ (if all subterms had to match variables or there were no subterms, this path has a length of 0 and $t' \equiv t$). By assumption we can now use $\mathbf{r}_i$ to rewrite t' to s. By our induction hypothesis, by now taking s as t and repeating evaluation from (1) will result in a s that is in HNF.

□

So **PC-FS** can rewrite a term to its HNF. According to Lemma 2.1, it is now very easy to make a strategy $\mathbf{PC\text{-}FS}^*$ that uses **PC-FS** to write a term to NF. **PC-FS** must simply be repeated on the redexes left behind by lazy evaluation and all subterms will be in HNF, which means the whole term is in NF. But how do we go about finding these redexes? One way is to modify the strategy such that, if a variable is carried over during reduction, the subterm it had to match is evaluated anyway. But this means that, if that subterm were to disappear in a later reduction step, we evaluated a part of the term that did not have to be evaluated and we violated the principle of lazy evaluation.

We don't know whether the redexes that remained behind actually have to be considered until we know that the term containing them is in HNF. But because of the undecidable nature of HNF, we don't know if the term is in HNF until the algorithm terminates (which it may not if it encounters a cycle). So the only time at which we can start considering these redexes is after **PC-FS** terminates.

One way is to repeat the strategy in leftmost-outermost order on every $s \subset t$ after **PC-FS** terminates on t. Because t is already in HNF, the s will not disappear after evaluating it and we can safely assume that it is necessary to reduce it.

2.17 Lemma: Applying **PC-FS** in leftmost-outermost order on every subterm of t is normalizing.

Proof. We know that $\mathbf{PC\text{-}FS}(\mathbf{R}, t)$ is in HNF. Thus if we repeat it on every proper subterm, then by Lemma 2.1 we will achieve NF. □

But we already have a recursive call of **PC-FS** in the algorithm itself, so we may end up evaluating terms that we already know are in HNF. So although this approach works, it is not very efficient.

In Chapter 3 we explore a better way to write a term to NF. We use a decidable version of HNF to determine these important redexes while we are still rewriting the term, so we can evaluate them as we encounter them, without the need for the algorithm to terminate first and having to repeat it on every subterm.

Chapter 3

Omega-Reduction

In *The Functional Strategy and Transitive Term Rewriting Systems* [3], the Functional Strategy is defined through Omega-Reduction, or Ω-Reduction. This is a special type of reduction, introduced in *Sequentiality in Orthogonal Term Rewriting Systems* [4], that only reduces to a special constant called Ω. As we will see, this allows Ω-Reduction to focus on how a term reduces, rather than what it reduces to.

The main purpose of this chapter is to explain the concepts and variation of the **FS** introduced in [3], and to see how they pertain to **PC-FS**. We start by explaining Ω-reduction in depth and defining all the relevant aspects. Then we give the definition of the Functional Strategy from [3], which we call **OR-FS**. We conclude by proving that **OR-FS** really is a definition of the Functional Strategy by proving equivalence between **PC-FS** and **OR-FS** for the class of *left-incompatible TRSes*.

3.1 Omega-Reduction

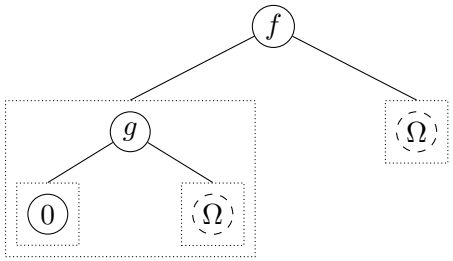
In Chapter 2, we saw that, in order to make an algorithm that can solve TRSes, we have to keep in mind that such an algorithm has to move through a term step by step. We tried to use Lemma 2.1 to make such an algorithm, but had to work around HNF being undecidable. We partially solved this by performing lazy evaluation and not considering the entire term at once. The algorithm could now at least find the HNF if it existed. Sadly, we had no good way to formalize "leaving part of the term out of consideration", having to resort to pseudocode to describe the way the algorithm moves through a term.

Sequentiality in Orthogonal Term Rewriting Systems [4] introduced a new constant: Ω. This constant represents the part of the term outside consideration; it is an abstraction of every possible term that can exist within the given TRS. This constant allows us to reason about a term while leaving parts of it out of consideration.

3.1.1 Omega-Terms

We introduce a new constant Ω, and the new set T_Ω of Ω-terms ($T(\Sigma \cup \{\Omega\})$). Because Ω is so important for the definitions and examples in the chapter, we use a special symbol for term nodes in our diagrams that contain Ω: a dashed circle.

3.1 Example (Diagram with Ω): $f(g(0, \Omega), \Omega)$



These terms have a partial ordering $\succeq$ that tells us something about their basic shape.

3.2 Definition (Ω-terms): We define Ω-terms as follows:

1. We introduce a new constant Ω and the new set $T_\Omega = T(\Sigma \cup \{\Omega\})$, the set of Ω-terms. We also define the preordering $\succeq$ on T_Ω as follows:
   (a) $\forall_{t \in T_\Omega}, t \succeq \Omega$.
   (b) $f(t_1, \ldots, t_n) \succeq f(s_1, \ldots, s_n)$ $(n \geq 0)$ if $t_i \succeq s_i$ for $i = 1, \ldots, n$.
   (c) If $t \succeq s$ and $t \not\equiv s$, then $t \succ s$.
   (d) Let $S \subset T_\Omega$. If there exists an $s \in S$ such that $t \succeq s$, then $t \succeq S$. Otherwise, $t \not\succeq s$.

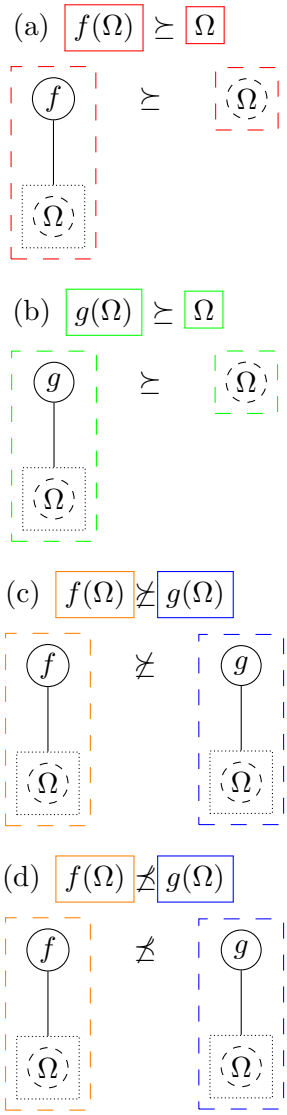
If $t \succ s$, we left more of s out of consideration than of t. So by leaving more of t out of consideration (by replacing certain subterms with Ω), we can get s.

3.3 Example: Let $t = f(g(\Omega))$ and $s = f(\Omega)$. Then $t \succ s$. But if we leave $g(\Omega)$ out of consideration in t we end up with $f(\Omega)$, which is the same as s.

If neither $t \succeq s$ nor $s \succeq t$, it is not possible to get one term by leaving more of the other out of consideration. Then the two terms are not ordered with respect to each other, hence why this ordering is only partial.

In Examples 3.4 and 3.5 we use coloured boxes to show the critical parts of the terms that determine whether they can be ordered or not. Matching colours indicate that two subterms can be ordered, conflicting colours that they cannot.

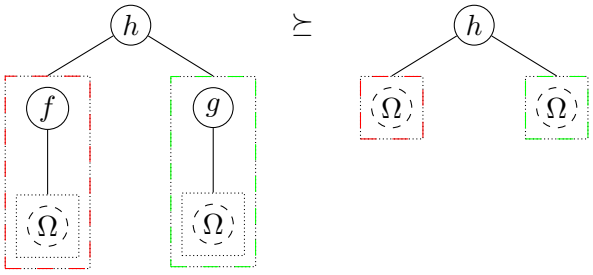
3.4 Example: Respective ordering of Ω, $f(\Omega)$ and $g(\Omega)$.



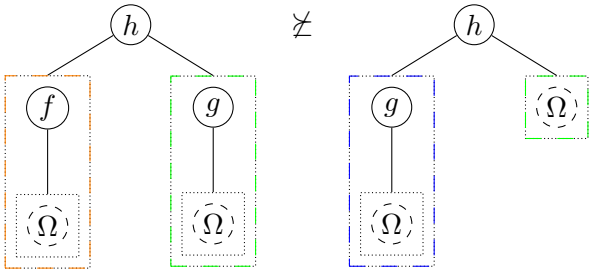
In Example 3.5, pay extra attention to the framed terms at the left side, and the Ωs in their place at the right side. If the colours of the frames are the same on both sides, the right side has an Ω there. If they are not, the right side has another term there. This ties into how, in order for the left side to be of higher order than the right side, the left side has to be able to turn into the right side by leaving more of it out of consideration. This is not possible if the two sides have different terms at the same location.

3.5 EXAMPLE: Respective ordering of $h(f(\Omega), g(\Omega))$ with $h(\Omega, \Omega)$ and $h(g(\Omega), \Omega)$

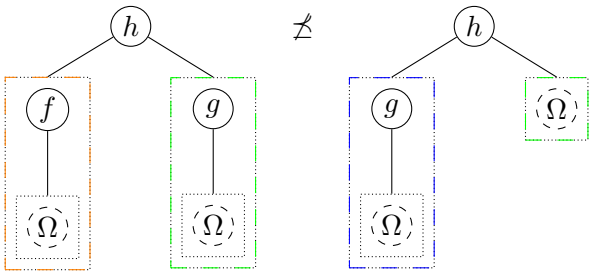
(a) $h(f(\Omega), g(\Omega)) \succeq h(\Omega, \Omega)$



(b) $h(\boxed{f(\Omega)}, \boxed{g(\Omega)}) \not\succcurlyeq h(\boxed{g(\Omega)}, \boxed{\Omega})$



(c) $h(\boxed{f(\Omega)}, \boxed{g(\Omega)}) \not\preccurlyeq h(\boxed{g(\Omega)}, \boxed{\Omega})$



3.1.2 Compatibility

Compatibility expresses another kind of similarity. Let r be a term in a TRS. By leaving certain subterms of r out of consideration you may end up with a term t, and by leaving certain other subterms out of consideration you get a term s: $r \succeq t$ and $r \succeq s$. If there exists such a *common ancestor* r for two terms t and s, they are *compatible*, written as $t \uparrow s$. The inverse is *incompatibility*, written $t\#s$, in which such a common ancestor does not exist.

3.6 Definition (Compatibility): 1. If there exists an $r \in T_\Omega$ such that $r \succeq t$ and $r \succeq s$, then $t \uparrow s$ (t and s are compatible). Otherwise, $t\#s$ (t and s are incompatible).

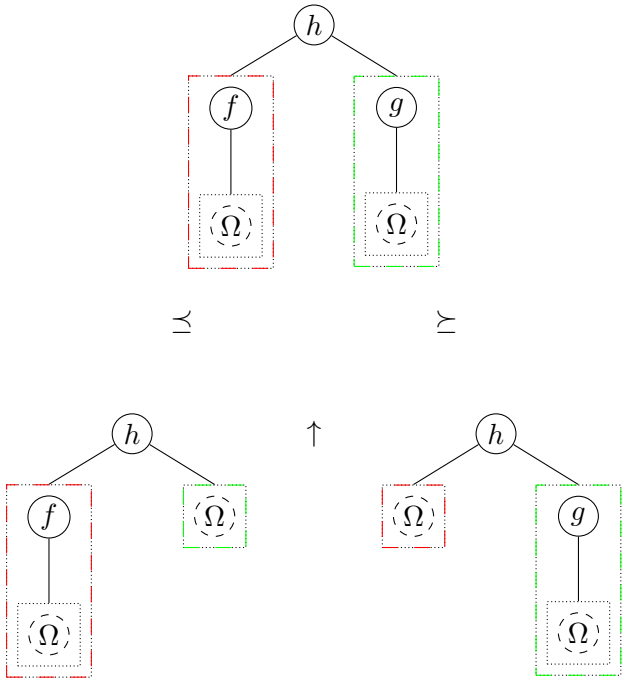
2. Let $S \subset T_\Omega$. If there exists an $s \in S$ such that $t \uparrow s$, then $t \uparrow S$. Otherwise, $t\#S$.

3.7 Example: Compatibility of $h(f(\Omega), \Omega)$ and $h(\Omega, g(\Omega))$.

(a) $h(f(\Omega), \boxed{g(\Omega)}) \succeq h(f(\Omega), \boxed{\Omega})$

(b) $h(\boxed{f(\Omega)}, g(\Omega)) \succeq h(\boxed{\Omega}, g(\Omega))$

(c) $h(\boxed{\Omega}, \boxed{g(\Omega)}) \uparrow h(\boxed{f(\Omega)}, \boxed{\Omega})$



Two terms are compatible if they leave different parts out of consideration but are otherwise the same. Also, note these special cases: for all Ω-terms t and s, $t \uparrow t$ and $t \succeq s \implies t \uparrow s$. Why? Because $t \succeq t$ is always true, which means that every Ω-term can be its own ancestor.

3.1.3 Omega-systems and Strong Head-Normal Form

We have defined several relations for Ω-terms now, but how are they useful for term rewriting? For this we introduce a new class of TRSes called Ω-systems.

3.8 Definition (Ω-systems): Let $\mathbf{R}$ be a TRS.

1. $Red = \{l_\Omega | l \to r \in \mathbf{R}\}$ is the set of *redex schemata* of R.
2. Ω-reduction ($\to_\Omega$) is defined on T_Ω as $C[s] \to_\Omega C[\Omega]$ where $s \uparrow Red$ and $s \not\equiv \Omega$.
3. The Ω-system $\mathbf{R}_\Omega$ corresponding to $\mathbf{R}$ is defined as a reduction system on T_Ω with $\to_\Omega$ as reduction relation.

4. $\omega(t)$ denotes the normal form of t with respect to $\to_\Omega$. NF_Ω is the set of Ω-normal forms.

We are not interested in *what* a term reduces to, but rather *which parts* have the *potential* to reduce. Remember that Ω is a formalization of terms left out of consideration. So by rewriting to Ω in Ω-reduction, we literally take terms out of consideration by Ω-reducing them.

In addition, a term merely has to be *compatible* with any rewrite rule to rewrite, in other words if the term shares a common ancestor with the rewrite rule. In the rewrite rule, the Ωs took the place of variables and as such it does not matter what the term had in those locations; in the term, the Ωs represent the parts that we left out of consideration, and as such have the *potential* to fit the rewrite rule. To find out if the rule would really fit in actual rewriting, we'd have to take the Ωs in the term into consideration. This offers us the following advantage: if a term doesn't Ω-reduce, it will NEVER rewrite regardless of what is in the locations that were left out of consideration.

If we look at it more formally, Ω-reduction is extremely linear. It only rewrites to Ω, and Ω cannot be rewritten any further. As a result it always terminates. This means we can use it to define a decidable variant of the Head-Normal Form property called Strong Head-Normal Form:

3.9 Definition (Strong Head-Normal Form): A term t is in *Strong Head-Normal Form* (SHNF) if $\omega(t) \not\equiv \Omega$.

We just explained that, if a term is *not* compatible with any rewrite rule, we can be sure that no rewrite rule will ever fit it, regardless of what subterms were in the locations we left out of consideration. Remember that the biggest problem of HNF was that we could not decide it without possibly ending up in a non-terminating reduction path. SHNF completely circumvents this problem by being based on a form of reduction that only acknowledges whether a part of the term has any chance at all of getting rewritten.

Lemma 3.8 from *The Functional Strategy and Transitive Term Rewriting Systems* [3] shows us why SHNF is useful. We only give the lemma here:

3.10 Lemma: If t is in SHNF then t is in HNF.

Proof. See Lemma 3.8 in Toyama et al. [3] □

This in conjunction with Lemma 2.1 means that we can achieve NF by rewriting every subterm to SHNF.

When deciding SHNF, we run into none of the non-termination issues that plagued HNF. So we should be able to use SHNF to explore on which TRSes the Functional Strategy is normalizing.

3.2 Solving TRSes with Omega-Reduction

3.2.1 Indexes and Strong Sequentiality

Now we have a form of rewriting that focuses solely on the rewriting, we want to be able to identify the redexes that remain unaffected through lazy evaluation. If a redex remains behind in a term after it is written to its HNF, we call such a redex an *index*. We say the same for an Ω that remains behind:

3.11 Definition (Index): Let $C[\ ]$ be a context such that $z \in \omega(C[z])$ where z is a fresh variable. Then the displayed occurence of Ω in $C[\Omega]$ is called an *index* and we write $C[\Omega_I]$. Let $C[\Omega_I]$ and Δ be a redex occurrence in $C[\Delta]$. This redex occurrence is also called an index and we write $C[\Delta_I]$.

The benefit of holes and contexts here is that they allow us to talk about (fresh) variables even though all variables in the system itself are replaced with Ω.

3.12 Definition (Strong Sequentiality): Let $\mathbf{R}$ be a TRS.

1. $\mathbf{R}$ is *strongly sequential* if for each term t that is not in NF, t has an index.
2. If Δ is an index of t then $t \xrightarrow{\Delta} s$ is the index reduction.

By searching for indexes in terms from a strongly sequential TRS and rewriting them, those terms will eventually reach NF. This is shown in Proposition 3.11 from *The Functional Strategy and Transitive Term Rewriting Systems* [3], who in turn refer to the proof from *Call by need computations in non-ambiguous linear term rewriting systems* [5]. We give the proposition here:

3.13 Proposition: Let $\mathbf{R}$ be strongly sequential. Then index reduction is normalizing.

Proof. Proven by Huet and Levy in *Call by need computations in non-ambiguous linear term rewriting systems* (1979) [5]. □

3.2.2 Transitivity in term rewriting

There is a problem with searching for these indexes. Remember that, according to lazy evaluation, if a term has to match a variable, it is not evaluated, even if that variable carries over in the rewrite step. This is because we don't know if that term has to match another variable in another rewrite step that does not carry over until we are sure the term is in HNF. Deciding whether a term is an index depends on deciding whether the term that contains it is in HNF.

Let $C_1[\Omega_I]$ and $C_2[\Omega_I]$ be two contexts. We cannot assume that also $C_1[C_2[\Omega_I]]$ until we know that C_1 is in HNF because it is possible that the

combination of C_1 and C_2 created a term that did match a pattern, such that $z \notin \omega(C_1[C_2[z]])$ (whatever is in the hole will not carry over). In this case the Ω_I in $C_2[\Omega_I]$ is no longer an index if C_2 is inserted at the index in C_1. Even if $C_1[C_2[z]]$ doesn't immediately rewrite, it is possible that the HNF of C_2 is a C_2' such that $C_1[C_2'[z]]$ does rewrite. In this case too an Ω or redex at the location of z is not an index.

Because this can happen after every index reduction, every time we find an index and rewrite it, we have to reconsider the entire term when searching for the next index. This is similar to lazy evaluation: we never evaluate a subterm that has to match a variable because it is entirely possible that the subterm doesn't disappear after only a single rewrite step but it does after a few more. If the subterm had to match a variable every time, lazy evaluation ensure that it is never evaluated and will disappear from the term without having ever been considered. But if the subterm didn't disappear it remained behind in the term even after **PC-FS** terminated, which meant that it could only write to HNF.

If we can determine which indexes *do* remain behind, we would know exactly which subterms we have to repeat the strategy on *while we are still executing the strategy* and we could continue searching for the next index at the point we left off. We could even base our entire strategy on just searching these points and rewriting them. Not only would it be much more efficient than rewriting to HNF and then repeating the strategy, it also means our strategy rewrites to NF instead.

If an index stays an index even if it's behind another index we call it a *transitive index*:

3.14 Definition (Transitive Index): The displayed index in $C_1[\Omega_I]$ is *transitive* if for any Ω-term $C_2[\Omega_I], C_2[C_1[\Omega_I]]$. We indicate the transitive index with $C_1[\Omega_{TI}]$. We also call the redex occurrence Δ in $C_1[\Delta]$ a transitive index and indicate it with $C_1[\Delta_{TI}]$.

3.15 Definition (Transitive Term Rewriting Systems): Let $\mathbf{R}$ be a TRS. $\mathbf{R}$ is *transitive* if for each term t that is not in SHNF, t has a transitive index.

But this only tells us something about SHNF and transitive TRSes. In order to actually use it, we need a connection between transitive and strongly sequential TRSes. For that we use the following proposition, based on Proposition 4.6 from *The Functional Strategy and Transitive Term Rewriting Systems* [3]. We only give the proposition:

3.16 Proposition: Let $\mathbf{R}$ be a TRS. If $\mathbf{R}$ is transitive then $\mathbf{R}$ is strongly sequential.

Proof. See Proposition 4.6 in Toyama et al. [3] □

Combining Proposition 3.13 and Proposition 3.16 shows that index re-

duction is normalizing for transitive TRSes.

3.2.3 Transitive directions

Now we know that index reduction is normalizing for a transitive TRS, we have to actually find them. For this we must identify parts of the term that will never fit a rewrite rule regardless of the shape of the term above them. We call these *directions*.

3.17 Definition (Direction): 1. Let $Q \subseteq T_\Omega$. The displayed Ω in $C[\Omega]$ is a *direction* for Q if $C[z] \# Q$. We indicate a direction for Q with $C[\Omega_Q]$.

2. Let $Red^* = \{p | \Omega \prec p \subseteq r \text{ for some } r \in Red\}$. A *transitive direction* is defined as a direction for Red^*. We denote a transitive direction with $C[\Omega_{TD}]$.

A transitive direction of a term is a subterm in SHNF. Thus, if it contains a redex, that redex is an index (SHNF implies the direction cannot be rewritten, therefore no terms it contains can get lost during rewriting). With transitive directions, we can track down indexes step by step as shown by this lemma, based on Lemma 4.9 from *The Functional Strategy and Transitive Term Rewriting Systems* [3]:

3.18 Lemma: Let $C[\Omega_{TD}]$ and $C[z]$ in NF_Ω. Then the displayed Ω is an index.

Proof. See Lemma 4.9 in Toyama et al. [3] □

We also have to show that transitive directions are actually a good way to find these indexes. In *The Functional Strategy and Transitive Term Rewriting Systems* [3], Toyama et al. use 3 lemmas to proof this. We only give the final and most important lemma here, but first we need a new definition:

3.19 Definition ($Red^\prec$): $Red^\prec = \{p \mid \Omega \prec p \prec r \; for \; some \; r \in Red\}$

This set contains all possible terms that can be created by leaving more of the elements of Red out of consideration. Now say that $t \in Red^\prec$ was made by leaving more of element $s \in Red$ out of consideration. Then $t \prec s$. Then also $t \preceq s$. Then s can act as common ancestor, and $t \uparrow s$. This means that $t \uparrow Red$, which means Ω-reduction can rewrite any element from $Red^\prec$ to Ω. So $\omega(t) \equiv \Omega$.

This makes $Red^\prec$ the terms that are relevant when determining the next index, because these are the terms that have the potential to fit a rewrite rule, and therefore determine which of their subterms are indexes (as it is here where subterms may disappear after a rewrite step). So these are the terms that we want to look at to find the next direction.

3.20 Lemma: A TRS $\mathbf{R}$ is transitive iff every $t \in Red^\prec$ has a transitive direction.

Proof. See Lemma 4.13 in Toyama et al. [3] □

If our TRS is transitive, we can always find a transitive direction, and thus we can always locate a transitive index. In addition we can use this lemma to check if a TRS is transitive.

3.2.4 Left-Incompatibility

In Example 2.13 we saw an orthogonal TRS that the **FS** was only normalizing on if the rules were tried in the right order. This order appeared to be related to the order in which the **FS** tries to match patterns and forces evaluation. This seems to make sense, since if we try the rewrite rules that leave a subterm out of consideration before the rules that don't, we won't consider subterms until we are sure that they have to be considered to find the right rewrite rule.

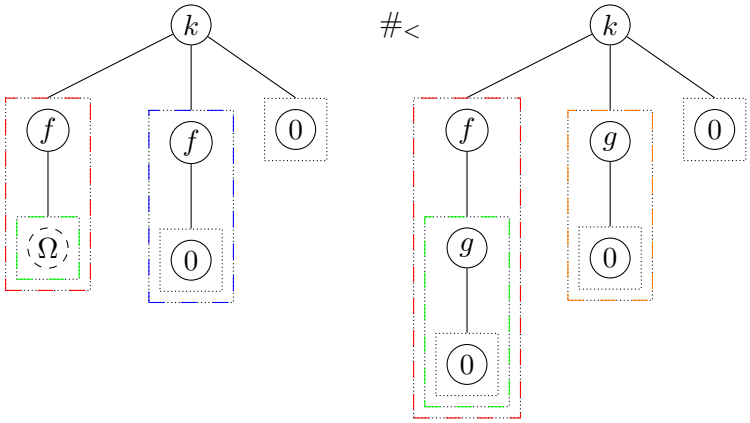
Using Ω-reduction and compatibility, we can define this property as *left-incompatibility*. If two terms are left-incompatible, they are only similar in structure from left to right, up to a certain subterm, the *left-incompatible point*. All subterms left of this point are ordered in a certain way, the point itself is incompatible, and the terms on the right side can be anything.

3.21 Definition (Left-Incompatibility): Terms $s, t \in T_\Omega$ are *left-incompatible* ($t\#_< s$) iff:

1. $t \not\equiv s$, $t \not\equiv \Omega$, $s \not\equiv \Omega$, and
2. Let $t \equiv f(t_1, ..., t_n)$ and $s \equiv g(s_1, ..., s_m)$. Then:
   (a) If $f = g$, then $\exists i[(\forall j < i, t_j \preceq s_j) \wedge t_i\#_< s_i]$.
   (b) If $f \neq g$ the terms are left-incompatible regardless of the arguments.

The above i is called the *left-incompatible point*.

3.22 Example: Let $t = k(\boxed{f(\boxed{\Omega})}, \boxed{f(0)}, 0)$ and $s = k(\boxed{f(\boxed{g(0)})}, \boxed{g(0)}, 0)$. Then $t\#_< s$, because for the first arguments property (a) holds($\boxed{f(\boxed{\Omega})} \preceq \boxed{f(\boxed{g(0)})}$) and the 2nd argument of k, $\boxed{f(0)}$, can be picked as the left-incompatible point. But *not* $s\#_< t$ because property (a) does not hold when the first arguments are swapped around: $\boxed{f(\boxed{g(0)})} \not\preceq \boxed{f(\boxed{\Omega})}$.



Left-incompatibility allows us to define a special type of order, the *left-incompatible order*:

3.23 Definition (Left-Incompatible Order): Let $\mathbf{R} = (\Sigma, R)$ be a TRS with $R = \mathbf{r}_1, ..., \mathbf{r}_n$ and let $\mathcal{R}$ be an order of this TRS. If $i < j \implies \mathcal{R}(i)^{\Omega} \#_{<} \mathcal{R}(j)^{\Omega}$, then $\mathcal{R}$ is a *left-incompatible order*.

We can also define a special class of TRSes, the *left-incompatible TRSes*:

3.24 Definition (Left-Incompatible Term Rewriting System): An orthogonal TRS is *left-incompatible* if it satisfies the following two conditions:

(i) *Red* can be expressed as a list $[p_1, ..., p_n]$ with $p_i \#_{<} p_j$ if $i < j$.

(ii) $\forall p_i \in Red$, $q \in Red^{+}[p_i \#_{<} q]$, where $Red^{+} = Red^{*} - Red$.

(iii) If, for every p_i in $Red = [p_1, ..., p_n]$, $\mathcal{R}(i) = \mathbf{r}_i : p_i \to r_i$ then $\mathcal{R}$ is the left-incompatible order that belongs with this *Red*.

Note that (i) merely requires *Red* to be able to be expressed as such a list. Usually, there are several ways to make this list, so we need (iii) to specify that a left-incompatible order belongs to a specific list of *Red*. This allows us to use the position of an element in the list to determine the rule we need to rewrite the index we find in our strategy.

So far we can construct a normalizing strategy for *transitive* TRSes by searching for transitive directions and rewriting transitive indexes. But Example 2.13 and the left-incompatibility property suggest that the Functional Strategy is normalizing on specifically the class of left-incompatible TRSes. If we look at Lemma 6.5 from *The Functional Strategy and Transitive Term Rewriting Systems* [3], we see an interesting property:

3.25 Lemma: Let $\mathbf{R}$ be a left-incompatible TRS with $Red = [p_1, ..., p_n]$. Let $C[\,]$ be a context such that $C[\Omega] \uparrow p_d$, $C[\Omega] \# p_i (1 \leq i < d)$ and let $C[\Omega_{\{p_d\}}]$ display the leftmost direction for $\{p_d\}$. Then $C[\Omega_{TD}]$.

Proof. See Lemma 6.5 in Toyama et al. [3] □

This lemma shows that the leftmost direction for a rewrite rule is a transitive direction if the context it is in is compatible with that rule, but incompatible with every rule before it (so every rule $\#_<$ that rule). Remember that, in Ω-reduction, a term rewrites if it is compatible with a rewrite rule. Because our algorithm has to try the rewrite rules in order, if a term is compatible with this rule, it already has been matched with all rules before this one, and was found incompatible with them. *We know that the context is incompatible with every rule prior to this one because we have to match it to this rule in the first place.* So as soon as we find the leftmost direction we also know it is a transitive direction.

Lemma 3.20 tells us that, iff every $t \in Red^{\prec}$ has a transitive direction, the TRS is transitive. If we can prove that every $t \in Red^{\prec}$ also has a transitive direction in a left-incompatible TRS, we have therefore shown that left-incompatible TRSes are transitive. Then all our earlier lemmas will also hold for left-incompatible TRSes.

The following corollary is based on Corollary 6.6 in *The Functional Strategy and Transitive Term Rewriting Systems* [3]. Because of its importance, we will include the proof this time:

3.26 Corollary: Every left-incompatible TRS is transitive.

Proof. According to Lemma 3.20 it is sufficient to prove that each $t \in Red^{\prec}$ has a transitive direction. Let $t \in Red^{\prec}$. Then there exists some $p_d \in Red$ such that $t \# p_i$ $(i < d)$ and $t \uparrow p_d$. Since $t \not\geq p_d$, t must have a direction for $\{p_d\}$. By Lemma 3.25 (Lemma 6.5 in Toyama et al. [3]), the leftmost direction of t for $\{p_d\}$ is a transitive direction. □

3.3 Formalizing the FS with Omega-Reduction

3.3.1 Marking subterms

If we want to achieve NF by rewriting all subterms to SHNF, there is one final thing we need: a way to keep track of which subterms are already in SHNF. For this we introduce the concept of *marking*.

3.27 Definition (Marking): Let (Σ, R) be a TRS.

1. Let $D = \{root(l) | l \to r \in R\}$ be the set of *defined funcion symbols*. $D^* = \{f^* | f \in D\}$ is the set of *marked function symbols* assumed that $D^* \cap \Sigma = \emptyset$ and f^* has the arity of f. It is clear that $f^* \in D^*$ is not a defined function symbol. $T^* = T(\Sigma \cap D^*)$ is the set of *marked terms*.

2. Let t be a marked term. $e(t)$ denotes the term obtained from t by erasing all marks. $\delta(t)$ denotes the Ω-term obtained from t by replacing all the maximal subterms with defined function symbols (Definition 3.27) at the roots with Ω. $\bar{\delta}(f(t_1, ..., t_n)) \equiv f(\delta(t_1), ..., \delta(t_n))$ for $f \in \Sigma \cap D^*$.

3.28 Definition: $t \in T^*$ is *well-marked* if $\forall s \subseteq t[root(s) \in D^* \implies e(\delta(s)) \in \mathrm{NF}_\Omega]$.

A term is well-marked if the marks clearly show which part of that term will never rewrite again. The following lemma, based on Lemma 5.5 from *The Functional Strategy and Transitive Term Rewriting Systems* [3], shows us how we can use this concept in our strategy:

3.29 Lemma: Let t be well-marked and let $e(\bar{\delta}(t)) = C[\Omega_{TD}]$. Then $C[z] \in \mathrm{NF}_\Omega$.

Proof. See Lemma 5.5 in Toyama et al. [3] □

So if a term is well-marked up and has a transitive direction, the context around that transitive direction is in SHNF.

There is one more thing we need: a way to refer to whatever subterm is at that transitive direction.

3.30 Definition: Let

$$t \equiv C[t_1, ..., t_p, ..., t_n] \in T^*$$

and

$$t' \equiv e(C)[\Omega, ..., \Omega_{TD}, ..., \Omega].$$

Then we say that t_p is a *directed subterm of t with respect to t'*.

A directed subterm is the subterm in a transitive direction. Combined with everything we established about marks and well-marked terms, this allows us to use marks to establish the next subterm to look at in our strategy.

3.3.2 The FS defined with Omega-Reduction (OR-FS)

The Functional Strategy and Transitive Term Rewriting Systems [3] defines the Functional Strategy as an algorithm that only returns an index. We expand on this algorithm by including our notion of *order* and rewriting the index the moment it is found. Thus the output should be the normal form of the term we want to rewrite.

3.31 Definition: (**OR-FS**) The Functional Strategy defined with Ω-reduction, taking a left-incompatible TRS $\mathbf{R} = (R, \Sigma)$ with left-incompatible order $\mathcal{R}$ belonging with $Red = [p_1, ..., p_n]$ and a term $t \in T(\Sigma)$ as input.

1. If t has no defined function symbol, return $e(t)$ as the normal form.
2. Take the leftmost-outermost subterm of t having a defined function at the root as s.

3. Find the first compatible element p_d to $e(\bar{\delta}(s))$ in the list *Red* if it exists; otherwise, mark the root of s and go to (1).

4. If $e(\bar{\delta}(s)) \succeq p_d$, $e(s)$ is our next index. Let $t = C[e(s)]$. Use $\mathcal{R}(d)$ to rewrite $e(s)$ to s', take $t = C[s']$ (to make the rewritten subterm part of the term) and go back to (1).

5. Take as s the leftmost directed subterm of s with respect to $e(\bar{\delta}(s))$ and p_d, and go to (3).

Does this strategy really return the normal form?

3.32 LEMMA: If $\mathbf{OR\text{-}FS}(\mathbf{R}, t) = s$, then s is the normal form of t.

Proof. We distinguish two cases: either t is already in NF, or t is not in NF.

$t \in$ NF: No subterms of t will ever be compatible with *Red*. So regardless of what else the algorithm does, (3) will never find a compatible element and always mark the root and go to (1). Eventually there will be no defined function symbols left and **OR-FS** terminates by returning $e(t)$, the original term without markings. Because we already know t was in NF, it therefore returns the NF of t.

$t \notin$ NF: If $e(t)$ is not in normal form, the algorithm will find a subterm s for which (4) holds. If s is really an index, $e(t) \equiv e(C)[e(s)_I]$. If this s was obtained from (2), it's the leftmost-outermost subterm of t and $C[\Omega]$ is in SHNF, meaning that this Ω (and therefore this s) is an index. If this s was obtained from (5), it's the leftmost directed subterm of an $s' \prec p_d$ with respect to $e(\delta(s'))$ and p_d in which case it's a transitive index. In both cases, s is really an index, which we immediately rewrite. By Proposition 3.13 and Corollary 3.26, index rewriting is normalizing, so as long as we keep rewriting indexes we will reach a point where $e(t)$ is in normal form. We have already proven that, once this is the case, the algorithm will just iterate until there are no defined function symbols left and terminates with $e(t)$.

□

3.3.3 Equivalence between PC-FS and OR-FS

We have to show that **OR-FS** is actually a definition of the Functional Strategy. We introduced the concept of marking to help us keep track of which part of the term was already in SHNF, and used directions to determine the next transitive index we had to rewrite. This is similar to how **PC-FS** attempts to rewrite a term to its HNF and has to be repeated on the redexes that remained behind to find the actual NF.

According to Definition 1.10, two reduction strategies are equivalent if their output is always the same. We cannot apply this here directly because **PC-FS** returns the HNF rather than the NF. But remember that we called the strategy that uses **PC-FS** to find the NF of a term **PC-FS***. Lemma 2.17 even showed us one way to make this **PC-FS***. Then **PC-FS*** $\equiv$ **FS**.

The goal of Toyama et al. [3] was to find a more efficient implementation of the **FS** than this approach, and the paper provided an algorithm to locate indexes. We based **OR-FS** on this algorithm, extending it by not returning the index but rewriting that index. This means that, if **OR-FS** really is an implementation of the **FS**, **OR-FS** $\equiv$ **FS** for any left-incompatible TRS. For this to be true, **OR-FS** $\equiv$ **PC-FS*** has to hold for the class of left-incompatible TRSes.

First we have to proof that what we just established about normalizing behaviour also holds for **PC-FS***:

3.33 Lemma: If **R** is a left-incompatible TRS with left-incompatible order $\mathcal{R}$, $\mathbf{PC\text{-}FS}^*_{\mathbf{R}}$ is normalizing on **R**.

Proof. Say that we want to evaluate $t = C[s]$ with **PC-FS***. The strategy won't evaluate s unless s has to match a non-variable, or **PC-FS** already terminated on C. In the first case, s has to be considered to determine NF, so it is an index. In the second case, $z \in \omega(C[z])$ so s is an index too. In both cases, s is only evaluated if it is an index.

From Corollary 3.26, Proposition 3.16 and Proposition 3.13 we know that index reduction is normalizing on left-incompatible TRSes. Since we just showed that **PC-FS*** only rewrites indexes, we have proven it is normalizing on left-incompatible TRSes (and in fact on all strongly sequential TRSes). □

Now that we know that both strategies are normalizing on left-incompatible TRSes, the equivalence proof is simple.

3.34 Corollary: **OR-FS** $\equiv$ **PC-FS*** for left-incompatible TRSes.

Proof. By Definition 1.10, two strategies are equivalent for a class of TRSes if they have the same output for every TRS in that class. By Lemma 3.33 and Lemma 3.32 we have proven that both **PC-FS*** and **OR-FS** are normalizing for the class of *left-incompatible TRSes*. Then from the orthogonality of left-incompatible TRSes and the confluence of orthogonal TRSes, if a term in a left-incompatible TRS normalizes it has only one normal form. Therefore **OR-FS** $\equiv$ **PC-FS*** for left-incompatible TRSes. □

From this we can conclude that, for left-incompatible TRSes, **OR-FS** is indeed an implementation of **FS**, and that the Functional Strategy is

normalizing on left-incompatible TRSes. However, we also found that **PC-FS** and therefore the Functional Strategy is normalizing on all strongly sequential TRSes, which we cannot say about **OR-FS**.

Chapter 4

Conclusion

4.1 Answers to the Research Questions

4.1.1 SQ1: How can we formalize the Functional Strategy?

The Functional Strategy can broadly be characterized as a rewriting algorithm with the following features:

1. The Functional Strategy rewrites all subterms to HNF. By Lemma 2.1 this rewrites the term to NF.
2. Rule Order: Rewrite rules are tried in a given order.
3. Pattern Order: Patterns are matched in leftmost-outermost order.
4. Lazy Evaluation: Subterms are only evaluated if they have to be considered to determine if a term is in NF.

We saw in Lemma 3.33 that lazy evaluation is the same as index reduction. Therefore, the Functional Strategy is essentially just an algorithm for finding and rewriting indexes.

4.1.2 SQ2: How can we describe the normalizing behaviour of the Functional Strategy?

Because the Functional Strategy performs leftmost-outermost pattern matching, **the order in which it tries the rewrite rules, determines whether a subterm will be matched with a variable before a non-variable**. When tried in the wrong order, this means the strategy may get stuck in a non-terminating reduction path that it would have avoided when trying the rules in a different order. **The property that determines in which order rewrite rules have to be listed to ensure that subterms are matched with variables before non-variables is called left-incompatibility**, and we found that the Functional Strategy is **normalizing on**

the class of left-incompatible TRSes. Furthermore, we also proved that the Functional Strategy is **normalizing on the entire class of strongly sequential TRSes**.

4.1.3 RQ: What are the formal characteristics of the Functional Strategy?

The Functional Strategy is a reduction strategy that **tries rewrite rules in a given order**, performs **leftmost-outermost pattern matching** with those rewrite rules, and **only evaluates those subterms that absolutely have to be considered** to determine whether a term is in normal form. This rewrites every subterm to HNF, which rewrites the term to NF. The Functional Strategy is **normalizing on the class of strongly sequential TRSes, of which the left-incompatible TRSes are a subclass**.

4.2 Future Work

At the end of *The Functional Strategy and Transitive Term Rewriting Systems* [3], Toyama et al. mention two major problems that have to be solved:

1. The Functional Strategy was initially intended as a strategy for Priority Term Rewriting Systems (PTRS). The adequacy of **OR-FS** for PTRSes therefore has to be investigated.

2. Additional, implementations of (lazy) functional languages that use the Functional Strategy appear to be efficient. Whether this efficiency can be founded theoretically has to be investigated.

In addition, there have been other approaches to formalizing the Functional Strategy that can be compared with the characteristics and normalizing behaviour that we specified here.

For example, in *Graph Rewriting Aspects of Functional Programming* [6], Barendsen et al. use the Functional Strategy to evaluate Graph Rewriting Systems (GRSes) and use special rewrite rules to rewrite TRSes (and GRSes) to another functional language that leaves no doubt about the way rewrite rules and patterns should be matched. It would be interesting to explore if this is the same as index rewriting, and therefore normalizing on strongly sequential and/or left-incompatible TRSes.

Bibliography

[1] Jan Willen Klop, *Term Rewriting Systems*, 1992.

[2] Rinus Plasmeijer, Marko van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison-Wesley Publishers Ltd., 1993.

[3] Yoshihito Toyama, Sjaak Smetsers, Marko van Eekelen, Rinus Plasmeijer, The Functional Strategy and Transitive Term Rewriting Systems, *Term Graph Rewriting: Theory and Practice*, Pages 61 - 75, John Wiley and Sons Ltd. Chichester, Uk, 1993.

[4] Jan Willem Klop, Aart Middeldorp, *Sequentiality in Orthogonal Term Rewriting Systems*, Academic Press Limited, 1991.

[5] Grard Huet, Jean-Jacques Lvy, *Call by need computations in non-ambiguous linear term rewriting systems*, Institut de recherche d'informatique et d'automatique, 1979.

[6] Erik Barendsen, Sjaak Smetsers, Graph Rewriting Aspects of Functional Programming, *Handbook of Graph Grammars and Computing by Graph Transformation*, Pages 63 - 102, World Scientific Publishing Co. Pte. Ltd. River Edge, NJ, 1999.