3

SPECIFICATION OF REDUCTION STRATEGIES IN TERM REWRITING SYSTEMS

M.C.J.D. van Eekelen, M.J. Plasmeijer.

Nijmegen University, The Netherlands.

partially supported by the Dutch Parallel Reduction Machine Project.

Abstract.

There is a growing interest in Term Rewriting Systems (TRS's), which are used as a conceptual basis for new programming languages such as functional languages and algebraic specification languages. TRS's serve as a computational model for (parallel) implementations of these languages. They also form the foundation for a calculus for Graph Rewriting Systems (GRS's). In Rewriting Systems reduction strategies play an important role because they control the actual rewriting process. Strategies determine the order of the rewriting and the rules to apply. Hence they have a great influence on the efficiency and the amount of parallelism in the computation. In ambiguous or non-deterministic TRS's, they even influence the outcome of the computation. Some of the reduction strategies used in TRS's are extremely complex algorithms. Unfortunately, there is no common formal specification method for reduction strategies yet.

In this paper three formal methods for specifying reduction strategies in TRS's are presented. In the first method the reduction strategy is encoded in the TRS itself. The original TRS is transformed to a so called annotation TRS in which the strategy is encoded using functions. This annotation TRS itself may use any normalizing reduction strategy. Unfortunately, compared with the number of rules of the original TRS, the annotation TRS may contain an exponential number of additional rules. In the second method this drawback is prevented, simply by using a priority TRS as annotation TRS. The desire to specify a strategy uniformly for all TRS's leads to the third method. A new TRS system is introduced that uses two basic primitives for matching and rewriting and that is build out of three separate TRS's. The use of this abstract-interpretation TRS is shown to be the most promising method.

3.1 INTRODUCTION

A Term Rewriting System (TRS) consists of a *term* τ to rewrite and a set ρ of *rewrite rules*. A *reduction strategy* σ is an algorithm which determines in which order the rewritable subterms of the term (the so called redexes i.e. reducible expressions) have to be rewritten and which rules have to be applied. A TRS with a strategy will be called a term *reducer* or a term *rewriter*. A *redex* is a subterm which matches the left-hand-side (LHS) of a rewrite rule. TRS's consist of *variables* and *symbols*. Symbols start with an upper case character. All symbols that occur as left-most symbol of a rewrite rule are *functions*, other symbols are *constructors*.

For instance when we have the following well-known set of rewrite rules:

```
Ap (Ap (Ap S x) y) z   → Ap (Ap x z) (Ap y z)                    (S)
Ap (Ap K x) y          → x                                       (K)
```

the term Ap (Ap (Ap K S) S) (Ap (Ap K K) K) can be rewritten to Ap S (Ap (Ap K K) K) and finally to Ap S K by applying two times the rule (**K**). In this example Ap is a function, S and K are constructors and x, y and z are variables.

A thorough introduction to TRS's is given in Klop (1985). TRS based languages like DACTL (Glauert *et al.* (1985)) and Lean (Barendregt *et al.* (1986b)) are used as a computational model for implementations of new languages such as Miranda (Turner (1985)) and OBJ (Futatsugi *et al.* (1985)). Currently various attempts (Barendregt *et al.* (1986a) based on Raoult (1984) and van den Broek & van der Hoeven (1986) based on Ehrig (1979)) are made to extend TRS's to a calculus for Graph Rewriting Systems (GRS's), in order to make them serve as a computational model of (parallel) graph reducing implementations.

A strategy in a TRS can be compared with the control flow in an ordinary imperative programming language. In the example above the strategy recursively takes the left-most redex in the term until the term after rewriting contains no redexes anymore and therefore it is in *normal form*. The outcome of the rewriting process may depend on the strategy followed. In particular this is the case if the TRS is ambiguous.

We distinguish the following forms of *ambiguity*:

- *Non-deterministic* type of rules, i.e. rules of which the LHS's match the same instance, such as

  ```
  Choose x y    → x
  Choose x y    → y
  ```

- *Strongly ambiguous* rules of which (an instance of) a subterm of a LHS is a redex, e.g.

  ```
  G (K n 1)     → 1
  K x y         → x
  ```

  or, more subtle

  ```
  F (F x)       → ...
  ```

  which is ambiguous with itself.

For TRS's like these the outcome is depending on the strategy that is followed. Non-deterministic rules such as Choose might be useful but, of course, the strategy must support it by making a non-deterministic choice between the rules. Another example of a useful ambiguous definition is the following TRS which still has the *Church-Rosser* property (i.e. the system has an unique normal form):

```
Or True x          → True
Or x True          → True
Or False False     → False
```

Although in the case that the TRS is Church-Rosser, the outcome of the computation will not depend on the order in which redexes are chosen, it can happen that depending on the strategy followed the rewrite process may or may not *terminate*. Take for instance Berry's example which has a unique normal form:

```
F x 0 1            → x
F 0 x 0            → x
F 1 1 x            → x
```

Let's assume that of the actual arguments of F in the term there are two arguments which reduce to the indicated normal form while the other argument has no normal form at all. The most efficient strategy would not touch the latter. Unfortunately, on forehand we do not know which argument that is. Forcing argument evaluation if a pattern is specified on the corresponding position in a rule will start a non terminating computation. The only safe way to reach the normal form is to reduce each argument of F once to see if the term has become a redex (the parallel outermost strategy). This strategy is not always the most efficient one.

Another example which shows the importance of strategies is the following program with a pretty familiar appearance:

```
Fac 0    → 1                                                    (1)
Fac n    → * n (Fac (- n 1))                                    (2)
```

It only has the obvious semantics if the right strategy is chosen. In general the rules are ambiguous because Fac 0 matches both rules. Fac (- 1 1) even matches the wrong rule. However a valid definition of Fac can be obtained if the argument of Fac is evaluated on forehand and the first rule has priority over the second rule. Now Fac (- 1 1) will be reduced to Fac 0 and the priority of the rules guarantees that Fac 0 now matches only the first rule. The reduction strategies in most functional languages are of this type.

Hence we must conclude that, unfortunately, there is no best strategy for all TRS's. Safe strategies are not always in all cases efficient. Efficient strategies are not always in all cases safe. Some algorithms can be expressed more conveniently in one strategy than another. Also one could prefer some strategies for specific reasons, e.g. for a parallel architecture one would like to reduce as much redexes as possible in parallel. One could also imagine, as is proposed in Lean, that several strategies are mixed in one and the same TRS in order to optimize performance and descriptive power. All this gives rise to complex strategies which cannot be explained anymore in terms like "take the left-most redex". Hence there is a need to specify strategies formally.

Before we will specify reduction strategies we must first ask ourselves the question what the properties must be of a good strategy specification. Of course it must specify which redexes are to be rewritten in which order and which of the matching rules have to be applied. If we see the specification as an algorithm, the execution of that algorithm must reduce exactly the same redexes in exactly the same order as the original strategy does. There is however a problem in the case of *parallel* reduction strategies which is caused by the concept time. The question arises if two redexes, which according to the original strategy have to be reduced in parallel, also have to be executed at the same time when we execute the corresponding specification.

This consideration leads to two views on parallel reduction: *synchronous* and *asynchronous*. According to the synchronous view the strategy recursively determines one or more redexes which have to be reduced in parallel after which these redexes are all rewritten in one step. This view is often used in a theoretical framework but in practice there are only very few machines for which this view is applicable (e.g. Magó (1980)). In most distributed environments the strategy algorithm is also implemented distributed and the asynchronous view is therefore more appropriate. In the asynchronous view redexes which are reduced in parallel are actually reduced in any (*possibly* parallel) order while the strategy may already have determined new redexes although not all previously assigned redexes were rewritten. In the asynchronous view we cannot state that the execution of the specification must rewrite *all* redexes in the same order as the original strategy would do. A specification can only model the dynamic behaviour. Though in this paper the specifications are used mainly in a sequential context, they can be easily extended to a parallel environment with the synchronous view as well as using the asynchronous view.

There are many ways in which strategies can be specified. In this paper three methods are introduced and a comparison between them is made. All methods create TRS's which can be reduced with any normalizing strategy.

1. Specification of the strategy by transforming the TRS τ to another one annotation-τ; reducing the latter induces the wanted strategy on τ.
2. Same as 1., but now in annotation-τ there is a priority in the rewrite rules (specificity).
3. The strategy is specified in a separate TRS on a different level.

For each method we will work out as a simple example the TRS consisting of the Curry combinators K and S (notated with explicit application functions). The strategy to be specified will be head reduction, i.e. left-most reduction to head normal form. A term is in head normal form if no reduction can possibly lead to a rewrite of the head of the term. We will refer to the example as Curry's example. For every method Curry's example will be proven correct. These proofs will be using O'Donnell's definition of a TRS simulating another TRS (O'Donnell (1985)). The methods are compared in terms of ease of correctness proving, ease of developing, ease of expressing efficient strategies etcetera.

3.2 TRANSFORMING A TRS TO AN ANNOTATION TRS

3.2.1 DESCRIPTION OF THE METHOD

The method is inspired by the use of strictness annotations in functional programming languages (Miranda e.g.) which serve as compiler directives. The general idea is that we use such annotations, for instance a shriek "!", for labeling those redex(es) in the term that should be rewritten. Instead of directives we use functions. These functions have ordinary semantics which enables us to use the full TRS semantics to reason about their meaning. Therefore the original TRS is transformed to an annotation TRS. All rules in this annotation TRS start with these functions and therefore only those redexes of the original TRS which are "annotated", are also redexes in the annotation TRS. The annotation TRS itself can be reduced with any normalizing reduction strategy.

A small example: if the rules of the original TRS are:

```
Ap (Ap (Ap S x) y) z        → Ap (Ap x z) (Ap y z)        (S)
Ap (Ap K x) y               → x                           (K)
```

Then these rules are transformed to:

```
! (Ap (Ap (Ap S x) y) z)    → Ap (Ap x z) (Ap y z)        (1)
! (Ap (Ap K x) y)           → x                           (2)
```

This TRS has only one function; the function !. The transformation has as a consequence that functions in the original TRS (Ap in the example) now become constructors in the annotation TRS. Of course, also the term to reduce must be marked. For instance if Ap ((Ap (Ap K S) S) (Ap (Ap K K) K) is transformed to Ap (! (Ap (Ap K S) S)) (Ap (Ap K K) K) it has only one redex in the annotation TRS. Due to the applicative order reduction this redex will be reduced giving Ap S (Ap (Ap K K) K) as result.

Hence only one of the redexes that is present in the original TRS, is reduced. This is the one which was marked.

In order to continue this process one also has to insert new markings for those redexes which have to be reduced next. When we specify a strategy in this way quite a lot of additional rules have to be added to deal with markings in the right way. Also the outcome of the computation must be the same as in the original TRS, i.e. all marking must disappear at the end. Let's try to make clear what kind of additional rules are needed by looking at our example.

3.2.2 EXPRESSING HEAD REDUCTION FOR CURRY'S EXAMPLE

We start off with the rules of the ordinary TRS:

```
Ap (Ap (Ap S x) y) z        → Ap (Ap x z) (Ap y z)        (S)
Ap (Ap K x) y               → x                           (K)
```

We add exclamation marks to the left-hand-side of the rules of the original TRS in order to promote the original redexes to redexes in the new system.

```
! (Ap (Ap (Ap S x) y) z)    → Ap (Ap x z) (Ap y z)                     (1)
! (Ap (Ap K x) y)           → x                                        (2)
```

Note that the function ! has only one argument, hence the parentheses around the argument will be sometimes redundant. When there can be no confusion, we will leave them out.

We need a general propagation rule to achieve a left-most search for a redex.

```
! (Ap (Ap (Ap (Ap v w) x) y) z) → Ap (! (Ap (Ap (Ap v w) x) y)) z      (3)
```

The propagation rule takes care of the propagation of the ! for the case that we have a pattern with four Ap's on the left. We also have to specify what happens for all other cases, i.e. when we have less than four Ap's. They can be divided in two classes:

1) The ! may have an argument which contains no redex at all. In this case the !-function must reduce to its argument in order to get the same normal form as in the original TRS. For each non-redex we have to add a rule. This gives us five additional rules.

```
! S                → S                                                  (4)
! K                → K                                                  (5)
! (Ap K x)         → Ap K x                                             (6)
! (Ap S x)         → Ap S x                                             (7)
! (Ap (Ap S x) y)  → Ap (Ap S x) y                                      (8)
```

Note that we could not write a more general rule like

```
! (Ap x y)    → Ap x y
```

because this rule is ambiguous with the K rule. This would introduce the danger that ! (Ap (Ap K K) K) is reduced to Ap (Ap K K) K instead of being reduced to K.

2) The ! may have an argument of which a subterm is a redex. We call rules that are introduced to handle these cases *envelope-rules*. In this example only one envelope-rule is needed:

```
! (Ap (Ap (Ap K x) y) z)    → Ap x z                               (9)
```

The nine rules we have given so far, model exactly **one** step of the head reduction strategy. In order to model the complete reduction to head normal form, the exclamation mark has to be created over and over in a driver rule like

```
* term   → * (! term)
```

In order to let this reduction stop we must encode in the term whether or not a rewrite (according to the strategy on the original TRS) was done. We use a success constructor '+' to denote that a rewrite was done and a failure constructor '-' to denote that it wasn't. We add the following 'driver' rules

```
* (- term)    → term                                          (10)
* (+ term)    → * (! term)                                    (11)
```

The strategy stops if on the term no rewrite was done (10). Of course we must also change all the other rules to make them produce a result with the proper success or failure constructor (+/-). Furthermore we must also propagate these constructors back to the beginning of the term so that we can make the decision whether or not to stop the strategy. This is done by adding the following rules:

```
~ (Ap (- x) y)    → - (Ap x y)                                (12)
~ (Ap (+ x) y)    → + (Ap x y)                                (13)
```

and by changing the propagation rule to:

```
! (Ap (Ap (Ap (Ap v w) x) y) z) → ~ (Ap (! (Ap (Ap (Ap v w) x) y)) z)  (3)
```

Finally the complete set of rules is:

```
! (Ap (Ap (Ap S x) y) z)         → + (Ap (Ap x z) (Ap y z))             (1)
! (Ap (Ap K x) y)                → + x                                  (2)
! (Ap (Ap (Ap (Ap v w) x) y) z)  → ~ (Ap (! (Ap (Ap (Ap v w) x) y)) z)  (3)
! S                              → - S                                  (4)
! K                              → - K                                  (5)
! (Ap K x)                       → - (Ap K x)                           (6)
! (Ap S x)                       → - (Ap S x)                           (7)
! (Ap (Ap S x) y)                → - (Ap (Ap S x) y)                    (8)
! (Ap (Ap (Ap K x) y) z)         → + (Ap x z)                           (9)
* (- term)                       → term                                (10)
* (+ term)                       → * (! term)                          (11)
~ (Ap (- x) y)                   → - (Ap x y)                          (12)
~ (Ap (+ x) y)                   → + (Ap x y)                          (13)
```

The initial term is

```
* (! term)
```

Note that it was also possible to achieve the same result using only one annotation function in stead of three different functions. This was not done for reasons of clarity.

3.2.3 CORRECTNESS PROOF

In this section we first formally define what it means for a strategy to specify another strategy. This definition will also be used in the proofs of the other methods. Then we proof that Curry's example, as it was specified with this method, satisfies the corresponding conditions and we finish this section with some general remarks on proving specifications that are constructed with this method. We also introduce some new terminology for a special kind of TRS that we encounter.

Definition of Specification

In his excellent book (O'Donnell (1985)) Michael O'Donnell defines what it means for a TRS to simulate another TRS. When one considers a strategy of a TRS to be giving a restriction on the reduction relation, then restricting the reduction relation to what is specified by the strategy, his definition can be used directly as the definition for one term rewriter specifying another. This results in the following definition:

Let $<\tau_o, \rightarrow_\sigma>$ and $<\tau_s, \rightarrow_\pi>$ be reducers where $\rightarrow_\sigma$ and $\rightarrow_\pi$ are the reduction relations of τ_o (the original TRS) and τ_s (the specification TRS) restricted to the strategies σ and π, then $\tau_{s\pi}$ *specifies* $\tau_{o\sigma}$ if there exist

an *encoding set* E Ÿ τ_s,
a *decoding function* $d: \tau_s \rightarrow \tau_o$ ‹ {nil} with nil ‡ τ_o,
a *computation relation* $\rightarrow_c$ Ÿ $\rightarrow_s$,

such that

1. $d[E] = \tau_o$ & $d^{-1}[N\tau_{o\sigma}]$ ¤ E Ÿ $N\tau_{s\pi}$
   where $N\tau_{x\psi}$ is the set of normal forms of a TRS τ_x with respect to the strategy ψ,
2. $\forall\alpha, \beta$ ¶ τ_s $\alpha \rightarrow_c \beta \Rightarrow d(\alpha) \rightarrow_o d(\beta)$,
3. $\forall\alpha, \beta$ ¶ τ_s $\alpha (\rightarrow_{s\pi}\text{-}\rightarrow_c) \beta \Rightarrow d(\alpha) = d(\beta)$
   where $\rightarrow_{s\pi}\text{-}\rightarrow_c$ stands for any $s\pi$ reduction that is not a c reduction,
4. $\forall\alpha$ ¶ E, β ¶ τ_o $d(\alpha) \rightarrow_{o\sigma} \beta \Rightarrow \exists\delta$ ¶ E $\alpha (\rightarrow_{s\pi}\text{-}\rightarrow_c)^* \rightarrow_c (\rightarrow_{s\pi}\text{-}\rightarrow_c)^* \delta$ & $d(\delta) = \beta$,
5. $\forall\alpha$ ¶ $N\tau_{s\pi}$ $d(\alpha)$ ¶ $N\tau_{o\sigma}$ ‹ {nil},
6. There is no infinite $(\rightarrow_{s\pi}\text{-}\rightarrow_c)$ reduction path and moreover there exists a bounded function $b: \tau_o \rightarrow$ » such that $\forall\alpha, \beta$ ¶ E $\alpha (\rightarrow_{s\pi}\text{-}\rightarrow_c)^m \rightarrow_c (\rightarrow_{s\pi}\text{-}\rightarrow_c)^n \beta \Rightarrow m < b(d(\alpha))$ & $n < b(d(\alpha))$.

The specification is called *effective* if b, $\rightarrow_c$, d and E are all total computable.

Intuitively, this means that we encode terms into another TRS allowing multiple encodings. The first condition requires that decoding respect normal forms. The specifying TRS has computational reductions which mirror a rewrite in the original TRS and non-computational reductions which are internal book-keeping steps. Conditions 2,3 and 4 require that every original reduction is simulated by any number of book-keeping steps which do not change the encoded expression, followed by exactly one computational reduction to effect the change in the encoded expression. This reduction may again be followed by book-keeping steps. Condition 5 prevents dead ends in the specification. Condition 6 states that the number of book-keeping steps is not allowed to grow without bound, otherwise e.g. all possible original reduction sequences could be simulated before officially choosing one of them, which clearly is not what we want. All conditions together mean that we really simulate the steps of the original TRS and not only return the appropriate normal forms as a result.

For the function d we basically need to describe which term is encoded for all reducts of the elements of the encoding set, but we also have to define which terms are not reducts of elements of E and are therefore decoded to nil. Maybe it is possible to give a precise definition of a TRS simulating another one using a decoding function which is only defined on reducts of elements of E. In the future we would like to do research along this line to find out whether for our purposes it is possible to simplify this definition.

Proof of Curry's Example

We define the encoding set E as $N\tau_{o\sigma}$ ‹ { * (! t) | t ¶ τ_o-$N\tau_{o\sigma}$}, where τ_o-$N\tau_{o\sigma}$ stands for the set of all terms of τ_o that are not terms of $N\tau_{o\sigma}$. The set of constructors *,+,-,!,~ is called A. The decoding function d is defined as follows:

x ¶ $N\tau_{o\sigma}$	›	d(x) = x	
x matches one of the rules of τ_s and the subterms matching variables do not contain elements of A	›	d(x) = x	with all applications of elements of A skipped
in all other cases		d(x) = nil.	

The computation relation c is given by rules (1), (2) and (9). This leaves us to proof that all conditions are satisfied. When we are not interested in the strategy π of $\tau_{s\pi}$ (i.e. the statement holds for any normalizing strategy of τ_s), we will leave out the strategy suffix π.

1a. d[E] = τ_o
Clear from the definitions.

1b. $d^{-1}[N\tau_{o\sigma}]$ ¤ E Ÿ $N\tau_s$
Suppose we have α ¶ E and β ¶ $N\tau_{o\sigma}$ with d(α) = β, then α = β and because α does not contain any elements of A, α ¶ $N\tau_s$.

2. ∀α, β ¶ τ_s $\alpha \rightarrow_c \beta \Rightarrow d(\alpha) \rightarrow_o d(\beta)$
Reductions according to the rules (1), (2) and (9) decode into S and K reductions.

3. ∀α, β ¶ τ_s $\alpha\,(\rightarrow_s\text{-}\rightarrow_c)\,\beta \Rightarrow d(\alpha) = d(\beta)$
Follows directly from the definition of d.

4. ∀α ¶ E, β ¶ τ_o $d(\alpha) \rightarrow_{o\sigma} \beta \Rightarrow$ ∃δ ¶ E $\alpha\,(\rightarrow_s\text{-}\rightarrow_c)^* \rightarrow_c (\rightarrow_s\text{-}\rightarrow_c)^*\,\delta$ & d(δ) = β
Suppose we have α ¶ E, $d(\alpha) \rightarrow_{o\sigma} \beta$; then d(α) ‡ $N\tau_{o\sigma}$, hence α is of the form * (! t) with t ‡ $N\tau_{o\sigma}$. So d(α) = t and t reduces in $\tau_{o\sigma}$ to β. We have to find a δ ¶ τ_s such that d(δ) = β **and** α $(\rightarrow_s\text{-}\rightarrow_c)^* \rightarrow_c (\rightarrow_s\text{-}\rightarrow_c)^*$ δ; t ‡ $N\tau_{o\sigma}$, so t has a head-redex r in τ_o, hence t is of the form Ap^1 (...

Ap^n (Ap (Ap (Ap S x) y) z) arg_n) ... arg_1 or Ap^1 (... Ap^n (Ap (Ap K x) y) arg_n) ... arg_1. Consequently, the form of β is also known (apply S or K rule). Take δ = * (! β) then trivially d(δ) = β.

Let us first assume that r is a S redex. If n > 0 then the only rule that is applicable on α is rule (3) which is not a computational rule. After one step this reduces to ~ (Ap^1 (! (Ap^2 (... Ap^n (Ap (Ap (Ap S x) y) z) arg_n) ... arg_1) and again if n-1 > 0 only rule (3) is applicable. So after n steps the result is ~ (Ap^1 (~ (Ap^2 (... ~ (Ap^n (! (Ap (Ap (Ap S x) y) z)) arg_n) ... arg_1). These n steps are the internal steps of τ_S preceding the computational step which is applying rule (1), resulting into ~ (Ap^1 (~ (Ap^2 (... ~ (Ap^n (+ (Ap (Ap x y) (Ap x z)) arg_n) ... arg_1). Now the only possible reductions are n applications of rule (13), which gives us * (+ β), which can only reduce to * (! β) = δ.

The other case we look at, is r being a K redex. We now have to be careful because there can be two computational rules corresponding to a K reduction. If n ≥ 1, then we can follow the same reasoning as in the S case. We have n-1 applications of rule (3) on application of rule (9) and n-1 applications of rule (13), followed by one application of rule (11); if n=0, then we can only apply directly the computational rule (2) and we have only one internal step from * (+ β) to * (! β).

5. $\forall\alpha$ ¶ $N\tau_S$ d(α) ¶ $N\tau_{O\sigma}$ ‹ {nil}
Take α ¶ $N\tau_S$ then either α is also a term of $N\tau_{O\sigma}$ and then d(α) = α and d(α) ¶ $N\tau_{O\sigma}$ or α is not a term of $N\tau_{O\sigma}$ and then d(α) is nil because α is not a redex in τ_S.

6. There is no infinite ($\rightarrow_S$-$\rightarrow_C$) reduction path and moreover there exists a bounded function b: τ_O $\rightarrow$ » such that $\forall\alpha, \beta$ ¶ E α $(\rightarrow_S\text{-}\rightarrow_C)^m \rightarrow_C (\rightarrow_S\text{-}\rightarrow_C)^n$ β $\Rightarrow$ m < b(d(α)) & n < b(d(α)).
In the proof of 4 it is shown implicitly that there is no infinite internal reduction path. We define b as b(t) = 3 * (the number of Ap's on the spine) + 1. Take α, β ¶ E then β can be an element of $N_{O\sigma}$ or not: β ‡ $N\tau_{O\sigma}$ $\Rightarrow$ β = * (! d(β)) and in the proof of 4 we saw that both m and n are less than the number of Ap's on the spine in respectively α and β. So certainly m < b(d(α)) and n < b(d(β)); analogously to the reasoning in 4 one can easily show that β ¶ $N\tau_{O\sigma}$ $\Rightarrow$ α $(\rightarrow_S\text{-}\rightarrow_C)^{m1} \rightarrow_C (\rightarrow_S\text{-}\rightarrow_C)^{n1}$ (* (+ β)) $(\rightarrow_S\text{-}\rightarrow_C)^{m2}$ γ $(\rightarrow_S\text{-}\rightarrow_C)^{n2}$ (* (- β)) where γ is a term not containing ! and n1, n2, m1 and m2 are all less or equal to the number of Ap's on the spine of α, β and γ respectively. Proving this we use the fact that rules (1), (2), (3), (4), (5), (6) and (9) cover all possible terms ! t with t ¶ $\tau_{O\sigma}$.

The specification is *effective* because b, $\rightarrow_C$, d and E are all total computable.

Remarks

Several parts of the proof heavily relied on the specific rules, the specific structure of the term and of course on the specific strategy. We see no way to easily extend these proofs to other reducers.

Note that starting with terms * (! t) every possible reduct had at most one redex. We call a term with such a property *linear*, because the reduction path is linear. A TRS where all terms are linear is called a *linear* TRS. For those who think this is confusing considering the concept of left-linearity, i.e.

having repeated variables on the left-hand-side, we suggest *left-comparing* as a possibly better name in stead of left-linear. It does not have the relation to polynomial terms (linear, quadratic etc.) and moreover it names the essential aspect of left-linearity which is that arguments have to be identical in order to be able to apply the rule. Linear TRS's have good prospects for efficient sequential execution, and left-comparing TRS's can cause trouble during execution when used with infinitely growing arguments.

3.2.4 EVALUATION OF THE METHOD

Although it is possible to convert the original TRS to an annotation TRS in which the strategy is explicitly encoded, the method has severe drawbacks.

First of all our annotation TRS has many more rules than the original TRS. Unfortunately the number of extra 'non-redex' rules per function can become exponential: the number of constants, that can occur, to the power of the width of the pattern. This results in too many rules. Some of these rules are very awkward such as the envelope rule. Furthermore the rules in the annotation TRS not only depend on the strategy but also on the original TRS. Consequently the correctness of the strategy can not be proven for all TRS's at once, but it must be proven for every TRS separately. Furthermore it is rather tedious work to give such a proof.

Concluding we can state that though it is very well possible to express a strategy in an annotation TRS, the number of rules and their complexity makes this method not very suitable for practical use.

3.3 TRANSFORMING THE TRS TO AN ANNOTATION TRS WITH PRIORITY RULES

3.3.1 DESCRIPTION OF THE METHOD

The method we will describe in this section is very closely related to the way strategies are expressed in Dactl (Glauert *et al.* (1985)). It differs from the previous method in that we will use a different kind of annotation TRS namely an annotation TRS with priority rules (Baeten *et al.* (1986)), a so called priority rewriting system (PRS). The difference with an ordinary TRS is that whenever a rule matches a given redex, it may only be chosen if none of the rules with higher priority can ever be applicable on the (internally rewritten) term.

3.3.2 EXPRESSING HEAD REDUCTION FOR CURRY'S EXAMPLE

We start again with the rules of the ordinary TRS

```
Ap (Ap (Ap S x) y) z        → Ap (Ap x z) (Ap y z)                    (A)
Ap (Ap K x) y               → x                                       (B)
```

and we add exclamation marks to the rules

```
! (Ap (Ap (Ap S x) y) z)    → Ap (Ap x z) (Ap y z)                    (1)
! (Ap (Ap K x) y)           → x                                       (2)
```

Our propagation rule is much simpler now:

```
! (Ap x y)                    → Ap (! x) y                         (3)
```

The reason for this is that if a redex matches the third rule **and** one of the other two rules, the topmost rule (**not** rule 3) is taken because It has higher priority. We indicate this by putting an arrow in front of rules with decreasing priority. Rules with the same priority are indicated by adding a bar in front of them. Rules without any indication are not affected by the priority mechanism.

The non-redex rules are extremely simple now:

```
! x                           → x                                  (4)
```

Anything that does not match one of the other rules, is not a redex. That's it! The somewhat strange envelope-rule of the previous section is also not necessary any more.

So we have elegantly modelled one step of this strategy. In order to model the strategy completely we still must encode whether or not a subterm was successfully rewritten. This is modelled by exactly the same changes and extra rules as in the previous section. Note that in general we must be careful with adding rules because our PRS could cause that they will never be applied. In this case we have no problems with that issue.

The complete set is now:

```
| ! (Ap (Ap (Ap S x) y) z)    → + (Ap (Ap x z) (Ap y z))     (1)
| ! (Ap (Ap K x) y)           → + x                          (2)
  ! (Ap x y)                  → ~ (Ap (! x) y)               (3)
  ! x                         → - x                          (4)
   * (- term)                  → term                         (5)
   * (+ term)                  → * (! term)                   (6)
   ~ (Ap (- x) y)              → - (Ap x y)                   (7)
   ~ (Ap (+ x) y)              → + (Ap x y)                   (8)
```

Again, the initial term must be of the form

```
* (! term)
```

Note that only the third and the fourth rule really rely on the order in which rules are matched.

3.3.3 CORRECTNESS PROOF

In order to prove this specification we first define what the semantics of a PRS is. Then we will show that in this case the semantics of the PRS is equivalent to the semantics of a TRS without priority. This TRS will turn out to be equivalent to the one we constructed using the previous method.

Semantics of a PRS

A PRS has a unique semantics (is *well-defined*) if it has a unique sound and complete rewrite set. Unfortunately our underlying TRS is not strongly normalizing nor bounded. So we have to use the stabilization lemma formulated in Baeten *et al.* (1986). We have to prove that starting with $R^{\underline{0}} = R_{max}$ (R_{max} being the set of all possible rewrites of τ_S i.e. all possible instances of all rules not considering the priority), there exists a $\underline{n}$ for which $R^{\underline{n}} = R^{\underline{n+1}}$, where $R^{\underline{n+1}}$ is defined as $(R^{\underline{n}})^c$ with $(R)^c$ being the set of all rewrites that is correct with respect to R. A rewrite $t \rightarrow_r s$ is *correct* if there is no internal R-reduction $t \rightarrow^* t'$ to an r'-rewrite $t' \rightarrow_{r'} s'$ ¶ R with r' > r (r' has higher priority than r). R is *sound* if all rewrites which are an element of R, are correct w.r.t. R. R is *complete* if it contains all possible rewrites of R_{max} which are correct w.r.t. R.

Proof

Because this method resembles the previous one, we can take all definitions the same except for the computation relation c which will be given by rules 1 and 2 only. But before we can even start to say something about this solution, we must prove that this PRS has a unique semantics.

If we restrict ourselves to those terms t for which $d(t) \neq nil$, then one can easily see that the term has at most one redex for which several rules with different priorities can be applied. Of course, the intended semantics is that the rule with the highest priority is applied. Since there is always at most one redex, there are no internal reductions of more than zero steps. Yet there are trivial zero-step reductions using other instantiations for the variables. So we start with the set R_{max} and compute $(R_{max})^c$. We prove that it is sound and complete, hence $((R_{max})^c)^c = (R_{max})^c$ and $(R_{max})^c$ is the semantics of our PRS.

We want to determine the set of rewrites that are correct w.r.t. R_{max}. As was already stated, the only internal redexes to be considered are other instantiations of variables. We know the structure of the terms, so all instantiations of the left-hand-side of rule (4) are ruled out by rule (3) except for the instantiations S and K. So the PRS is equivalent to another one where rule (4) is substituted by the rules:

```
! S            →  - S                                       (4')
! K            →  - K                                       (4'')
```

In the same way we can determine the patterns for which rule (3) is really applicable i.e. not ruled out by rule (1) or rule (2), giving the following rules replacing rule (3):

```
! (Ap S y)                          → ~ (Ap (! S) y)                         (3')
! (Ap K y)                          → ~ (Ap (! K) y)                        (3'')
! (Ap (Ap S a) y)                   → ~ (Ap (! (Ap S a)) y)                (3''')
! (Ap (Ap (Ap K a) b) y)            → ~ (Ap (! (Ap (Ap K a) b)) y)        (3'''')
! (Ap (Ap (Ap (Ap a b) c) d) y)     → ~ (Ap (! (Ap (Ap (Ap a b) c) d)) y)
                                                                        (3''''')
```

So the set $(R_{max})^c$ is the set determined by this new TRS without priority. This set is sound because evidently all rewrites in it are correct and it is complete because it also contains all rewrites which are correct with respect to it. One easily checks that the resulting rules that determine the semantics of the PRS, are equivalent to the rules of the previous example. So we have proven that our PRS model specifies Curry's example correctly.

Remarks

Note that this proof more or less included the proof for the previous method and that we needed special analysis to determine the semantics of this PRS. In general there is not even always a unique semantics and it is not known whether a semantics always exists (Baeten *et al.* (1986)).

3.3.4 EVALUATION OF THE METHOD

Clearly this method is a lot better than the previous one. There are more rules in the annotation TRS than in the original TRS, but the growth is no longer exponential because the non redex case can now be expressed with one rule (4). Also the funny envelope rules have disappeared.

Still this method has a severe disadvantage. The rules generated still heavily depend on the original TRS so if we want the same strategy for other TRS's we still have to change the rules of each one of them. The proof must be given for every TRS separately as was the case with the previous method. The complexity of the proof is even worse than the complexity of the previous proof because we also have to prove the well-definedness of the PRS. It would be better if we could define a strategy in such a way that the same description is valid for all TRS's.

3.4 DEFINING THE STRATEGY SEPARATELY IN AN ABSTRACT–INTERPRETATION TRS

3.4.1 DESCRIPTION OF THE METHOD

In order to be able to specify strategies uniformly for all TRS's we will consider three different conceptual levels of TRS's:

a) The first TRS is the *user-defined TRS*. This is the original TRS which now remains unchanged. An example of such a TRS is the S-K TRS.

b) The reduction strategy for the user-defined TRS is specified in a separate TRS called the *interpretation TRS*. Functions in the interpretation TRS may have a user-defined TRS or any subterm of such a TRS as argument. For instance one may write:

HeadReduce *S*	$\rightarrow S$	(I-a)
HeadReduce *K*	$\rightarrow K$	(I-b)

The arguments of HeadReduce are written in *italic* style to indicate that they are part of a TRS on another level. Instead of the rules above one can write a more general rule which is valid for *all* combinators:

HeadReduce *c : Combinator*	$\rightarrow c$	(I-c)

In this rule *c* is a variable which will be bound to a (sub)term of a TRS. The suffix *Combinator* restricts the number of matching expressions. Such a suffix is called an *abstraction* .

c) These abstractions which are used as patterns in the interpretation TRS's, are defined by a third TRS: the *abstraction TRS.* This abstraction TRS is used to make an abstract syntax of the user-defined TRS available to the interpretation TRS. An example of such an abstraction TRS is:

Combinator	$\rightarrow S$	(A-a)
Combinator	$\rightarrow K$	(A-b)

Now the extra restriction on *c* imposed by the suffix *Combinator* implies that the actual value of *c* must be a normal form of the term *Combinator* in the abstraction TRS. Hence only HeadReduce *S* and HeadReduce *K* will match rule (I-c). The abstraction level can be seen as a preprocessing level where things are done like syntactical categorizing, type checking, strictness analysis etcetera.

To make the specification of strategies really easy we introduce two primitive functions which can be used in the interpretation TRS:

Match *term rules*

which returns True if the *term* is a redex according to the *rules* and False otherwise.

Rewrite *term rules*

which returns the *term* after one rewrite according to the *rules* if the *term* matches the *rules* and the *term* itself otherwise.

The match function only checks whether or not the given term as a whole matches one of the given rules. The rewrite function will make a non-deterministic choice out of the matching rules. Although it is possible to define these primitive functions precisely in the interpretation TRS, the formal definition is rather tedious.

3.4.2 EXPRESSING HEAD REDUCTION FOR CURRY'S EXAMPLE

First we give the rules of the user-defined TRS:

```
Ap (Ap (Ap S x) y) z → Ap (Ap x z) (Ap y z)                    (U-S)
Ap (Ap K x) y        → x                                        (U-K)
```

Furthermore we have to define those abstractions of the TRS that we need at the interpretation level. In this example these abstractions are extremely trivial. The abstraction TRS is:

```
Combinator      → S                                             (A-1)
Combinator      → K                                             (A-2)
```

Finally we have to define the interpretation TRS. We will start with describing one step of the strategy: we will call that OneStepHead.

When the term is a redex we rewrite it and otherwise we search in the function part of apply for a redex (the propagation rule):

```
OneStepHead ( Ap f a ) rules          → Cond    ( Match ( Ap f a ) rules )
                                                ( Rewrite ( Ap f a ) rules )
                                                ( Ap ( OneStepHead f rules ) a )      (I-1)
```

In this rule Cond has the ordinary meaning. The result of OneStepHead applied to a function Ap with parameters is, if it matches as a whole, the rewrite of it, otherwise the result is the function Ap with as new parameters the result of OneStepHead applied to the function part. This surely terminates because of the next rule (I-2). Note that functions now also must have *rules* as parameter in order to be able to use the matching and rewriting primitives.

Again S and K cannot be rewritten, which we can now express in one rule:

```
OneStepHead c:Combinator rules        → c                                             (I-2)
```

The result of OneStepHead applied to a single combinator is that combinator itself.

We will now extend the one-step strategy to a complete strategy by adding another function:

```
Head term rules                       → Cond    ( ContainsAHeadRedex term rules )
                                                ( Head ( OneStepHead term rules ) rules )
                                                term                                  (I-3)
```

When our term contains a head-redex we do one left-most rewrite and continue, in the other case we stop and the result is the term itself.

The definition of ContainsAHeadRedex is very similar to the definition of OneStepHead. When there is a Rewrite in the definition of OneStepHead, we return True and when in OneStepHead we return the parameter as a result, we return False in ContainsAHeadRedex. Of course, recursive calls in OneStepHead are simply converted to recursive calls in ContainsAHeadRedex.

The definition of ContainsAHeadRedex is:

ContainsAHeadRedex *c:Combinator rules*	→ False		(I-4)
ContainsAHeadRedex (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) True (ContainsAHeadRedex *f rules*)	(I-5)

The complete set is now:

OneStepHead (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (Rewrite (*Ap f a*) *rules*) (*Ap* (OneStepHead *f rules*) *a*)	(I-1)
OneStepHead *c:Combinator rules*	→ *c*		(I-2)
Head *term rules*	→ Cond	(ContainsAHeadRedex *term rules*) (Head (OneStepHead *term rules*) *rules*) *term*	(I-3)
ContainsAHeadRedex *c:Combinator rules*	→ False		(I-4)
ContainsAHeadRedex (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) True (ContainsAHeadRedex *f rules*)	(I-5)

The initial term must be:

Head *Term SKRules*

3.4.3 CORRECTNESS PROOF

In order to prove this specification we first define what the semantics of the abstract-interpretation TRS is. We will proof that the conditions are satisfied after proving some simple lemmas. These lemmas state general facts on the functions that are defined. These facts are easy to find because they cover the intention with which we constructed the functions. It will be rather simple to prove those lemmas.

Semantics of the Abstract-interpretation TRS

We start with settling the semantics of *c:Combinator* in the rules for OneStepHead and ContainsAHeadRedex. Because of the structure of the terms OneStepHead is equivalent to:

OneStepHead (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (Rewrite (*Ap f a*) *rules*) (*Ap* (OneStepHead *f rules*) *a*)	(I-1)
OneStepHead *S rules*	→ *S*		(I-2$_a$)
OneStepHead *K rules*	→ *K*		(I-2$_b$)

For our semantics this eliminates the abstraction TRS. Furthermore the user-defined and the interpretation TRS can be seen as one and the same TRS with restrictions on the construction of terms, such as: the first argument of OneStepHead is an element of τ_o, the second argument are the rules, etcetera. During reduction reducts of terms that meet these restrictions also meet these restrictions. So we now have only one TRS with the ordinary meaning. However we do not deal with the full reduction relation but only with those reductions that are allowed by normalizing strategies. Strategies for this TRS are normalizing if they reduce the first argument of Cond and the Cond itself without reducing the other arguments and if they reduce the argument of Head to a term which is an element of τ_O, before applying the Head rule again (this restriction is a consequence of uniting the two levels). So we simply remove those reductions out of the reduction relation. The reducer we get this way is called τ_S again and we will prove the specification following again O'Donnell's definitions.

Proof

Before we proof that the specification satisfies O'Donnell's definitions we want to prove some lemmas about the functions we have defined. The formulation of these lemmas is simple because of the way the functions were constructed.

Lemma The normal form of ContainsAHeadRedex x SKRules with x ¶ τ_O is True if x has a head-redex in τ_O, False otherwise.

Proof: Recall the definitions:

Def: x contains a head-redex fi x matches a S or K rule or else
if x = Ap a b, a contains a head-redex.

```
ContainsAHeadRedex S rules          → False                                    (I-4a)
ContainsAHeadRedex K rules          → False                                    (I-4b)
ContainsAHeadRedex ( Ap f a ) rules → Cond   ( Match ( Ap f a ) rules )
                                             True
                                             ( ContainsAHeadRedex f rules )    (I-5)
```

Suppose x ¶ τ_O contains a head-redex then x ≠ S and x≠ K (S and K themselves do not match the S or K rule) so x = Ap a b. So ContainsAHeadRedex x SKRules reduces in τ_S to Cond (Match (Ap a b) SKRules) True (ContainsAHeadRedex f SKRules). If x matches a S or K rule then obviously the normal form is True because we restricted the reduction relation to safe strategies. If x does not match a S or K rule then the left part of the application contains a head-redex which by induction and the absence of terms with an infinite spine results in the correct normal form. Proceeding this way it is very simple to prove the other parts of the lemma.

The following lemma is just as simple to proof :

Lemma OneStepHead x SKRules with x ¶ τ_O has as its normal form: if x has a head-redex in τ_O, the reduct of x after reducing this head-redex and if x does not have a head-redex in τ_O, its normal form is x.

We can use these lemmas in our proof of the specification. It will make everything very easy. We define E as: x ¶ Nτ_O › E(x) = x ; x ‡ Nτ_O › E(x) = Head x SKRules. The computational relation is given by reductions of the primitive Rewrite. The definition of d is:

if x is an element of Nτ_O then d(x) = x;

if x can be reduced via non computational reductions to a x' which is an element of $N\tau_o$ then also d(x) = x;

if x is not an element of $N\tau_o$ then we define d a bit special: if x can be reduced to x' via non computational reductions until the only possible reduction is a computational one and x' is of the form : Ap^1 (..... Ap^n ((Rewrite (Ap f a)) .....), perhaps surrounded by Head and SKRules then the decoding of x is defined as x' with applications of Head, SKRules and Rewrite skipped.

In other cases the decoding of x is nil.

Because we are only interested in reducts of elements of E, we restrict τ_s to those reducts.

1a. d[E] = τ_o
Trivially true.

1b. $d^{-1}[N\tau_{o\sigma}]$ ¤ E Ÿ $N\tau_s$
Suppose α ¶ E and β ¶ $N\tau_o$ with d(α) = β, then clearly α = β, and since $N\tau_o$ Ÿ $N\tau_s$, α is an element of $N\tau_s$.

2. ∀α, β ¶ τ_s $\alpha \rightarrow_c \beta \Rightarrow d(\alpha) \rightarrow_o d(\beta)$
Suppose α reduces to β via a reduction of Rewrite then if α and β are reducts of elements of E then clearly the decodings also reduce to each other.

3. ∀α, β ¶ τ_s $\alpha\, (\rightarrow_s\text{-}\rightarrow_c)\, \beta \Rightarrow d(\alpha) = d(\beta)$
Analogously to 2.

4. ∀α ¶ E, β ¶ τ_o $d(\alpha) \rightarrow_{o\sigma} \beta \Rightarrow \exists\delta$ ¶ E $\alpha\, (\rightarrow_s\text{-}\rightarrow_c)^* \rightarrow_c (\rightarrow_s\text{-}\rightarrow_c)^*\, \delta\ \&\ d(\delta) = \beta$
Suppose α¶ E, $d(\alpha) \rightarrow_{o\sigma} \beta$ › α ‡ $N\tau_o$, so α = Head α' SKRules with α' ¶ τ_o. Furthermore $d(\alpha) \rightarrow_{o\sigma} \beta$, so α' has a head-redex. Then OneStepHead α' SKRules reduces to δ with d(δ) = β and Head α' SKRules reduces to Head δ SKRules.

5. ∀α ¶ $N\tau_s$ d(α) ¶ $N\tau_{o\sigma}$ ‹ {nil}
If α is a normal form of an element of E then d(α) ¶ $N\tau_{o\sigma}$, d(α) is nil otherwise.

6. There is no infinite $(\rightarrow_s\text{-}\rightarrow_c)$ reduction path and moreover there exists a bounded function b: τ_o → » such that ∀α, β ¶ E $\alpha\, (\rightarrow_s\text{-}\rightarrow_c)^m \rightarrow_c (\rightarrow_s\text{-}\rightarrow_c)^n\, \beta \Rightarrow m < b(d(\alpha))\ \&\ n < b(d(\alpha))$.
There is no infinite non computational path because we only use safe strategies. We define b to be: b(x) = 3*(the number of Ap's on the spine of x) + 3. We notate sp(x) for the number of Ap's on the spine of x. If x ¶ τ_o then ContainsAHeadRedex x SKRules has a number of non computational reductions which is bound by 3*sp(x). OneStepHead x SKRules is bound by 3*sp(x) + 1 because it has an extra Rewrite. So if β ‡ $N\tau_s$ then m= 3*sp(α) + 1 and n = 1; if β ¶ $N\tau_s$ then we may need 3*sp(β) + 2 non computational reductions to discover that we have a normal form.

The specification is *effective* because b, $\rightarrow_C$, d and E are all total computable.

Remarks

The proof was easy because the most essential aspects were already covered by the lemmas which were themselves simple and easy to proof. The use of the primitives Match and Rewrite made it possible to reason easily about the TRS system.

3.4.4 TRANSFORMATIONS OF THE SPECIFICATION

The description we have developed is a very nice one but operationally it is not the same as in the other two sections, because a call of ContainsAHeadRedex will induce an extra sweep through the term. If we want our description to mirror exactly the operations that were performed by the other methods, then we must change the rules and make them deliver a composite result consisting of the term and a boolean indicating whether the term was rewritten. The resulting rules are:

OneStepHead (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (Pair True (Rewrite (*Ap f a*) *rules*)) (Pair (First (OneStepHead *f rules*)) (*Ap* (Second (OneStepHead *f rules*)) *a*))	(I-1')
OneStepHead *c:Combinator rules*	→ Pair False *c*		(I-2')
Head (Pair False *term*) *rules*	→ *term*		(I-3')
Head (Pair True *term*) *rules*	→ Head (OneStepHead *term rules*) *rules*		(I-4')

with as initial term:

Head (OneStepHead *Term SKRules*) *SKRules.*

Probably it will not be very difficult to prove this specification to be equivalent to the one without the booleans. Though this specification can now be considered to be just as efficient as the specifications in the other sections, it does not exactly mirror the same actions. In the other sections annotation functions were used in stead of booleans. Of course it is also possible to give a similar description with the abstract-interpretation method. We will rename Head to * and OneStepHead to !. We will also introduce success and failure constructors +/- and a propagate function called ~. The set of corresponding rules is:

! (*Ap f a*) *rules*	→ Cond	(Match (*Ap f a*) *rules*) (+ (Rewrite (*Ap f a*) *rules*)) (~ (*Ap* (! *f rules*) *a*))	(I-1")
! *c:Combinator rules*	→ - *c*		(I-2")
* (- *term*) *rules*	→ *term*		(I-3")
* (+ *term*) *rules*	→ * (! *term rules*) *rules*		(I-4")
~ (*Ap* (+ *x*) *y*) *rules*	→ + (*Ap x y*)		(I-5")
~ (*Ap* (- *x*) *y*) *rules*	→ - (*Ap x y*)		(I-6")

If this specification is compared with that of section 3.2.2 we see that the redex rules and the propagation rule are now all covered by rule (I-1") thanks to the power of the match and rewrite primitives. The non-redex rules are covered by rule (I-2"). The last four rules are the same.

We have shown that it is relatively easy to express a reduction strategy using the abstract-interpretation TRS with several algorithms and that it is also simple to transform one specification to another.

3.4.5 EVALUATION OF THE METHOD

This method enables us to write elegant strategy specifications. The non-redex cases are easily handled using the Match and Rewrite primitives. The different actions for syntactically different terms are conveniently dealt with using the abstraction mechanism. The fact that we write our strategy description on another level helps us in specifying strategies more generally.

The abstract-interpretation TRS system makes it possible to specify strategies formally in such a way that the specification holds for a large class of TRS's. For instance the description given at the interpretation level in this section is a valid head reduction specification for all TRS's using explicit application functions. Only a very trivial adaptation to the abstraction TRS is necessary in order to summarize all combinators. The proof then also holds for this large class of TRS's. Besides that, the proof was less tedious because we were led immediately to some general lemmas which themselves were easy to proof using the properties of the primitives Match and Rewrite. This specification is very short and readable and it appeals to our intuition. It enables us to easily give transformations of one specification to other specifications which have specific properties.

Concluding we can state that we have a promising facility with a great expressive power for describing strategies independently of the TRS.

3.5 CONCLUSIONS AND FURTHER RESEARCH

All of the three methods introduced in this paper are suitable for the specification of reduction strategies. Using an ordinary TRS gives rise to an exponential number of rewrite rules. This drawback disappears when a PRS is used. However the proof of the specification using this PRS was even more difficult than the proof with an ordinary TRS.

The most readable and general specification can be obtained by using an abstract-interpretation TRS extended with special primitives for matching and rewriting. The structure of the specification made it possible to give a simple proof and to construct transformations of the specifications in order to get alternative specifications with special desirable properties. Although this new TRS system is specially designed for the specification of reduction strategies, we think that its descriptive power is suitable for the specification of abstract interpretations (Burn *et al.* (1985)) in general.

In the near future we will search for simplifications of the definition of specification and we will investigate the use of the abstract-interpretation TRS in its full strength for the specification of strategies in graph rewriting systems, for giving correctness proofs of more complex strategies and for the investigation of the descriptive power for other domains.

3.6 ACKNOWLEDGEMENTS

We would like to thank Ronan Sleep, John Glauert and Richard Kennaway of the University of East-Anglia for the many fruitful discussions about reduction strategies and we also thank Jan-Willem Klop of the Centre for Mathematics and Computer Science in Amsterdam for his explanations. Most of all we are grateful to Henk Barendregt of the University of Nijmegen for several important observations and valuable improvements.