Just-in-time: on Strategy Annotations

Jaco van de Pol

Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: `Jaco.van.de.Pol@cwi.nl`

Abstract. A simple kind of strategy annotations is investigated, giving rise to a class of strategies, including leftmost-innermost. It is shown that under certain restrictions, an interpreter can be written which computes a normal form of a term in a bottom-up traversal. The main contribution is a correctness proof of this interpreter. Furthermore, a default strategy is provided, called just-in-time, which satisfies the criteria for the interpreter. The just-in-time strategy has a better termination behaviour than innermost rewriting for many interesting examples.

1 Introduction

A term rewrite system (TRS) is a set of directed equations. A term is evaluated by repeatedly replacing a subterm that is an instance of the left-hand side of an equation (a redex) by the corresponding instance of the right-hand side (the contractum) until a term is reached which contains no redex (a normal form). Because a term can have many redexes, an implementation has to follow a certain *strategy*, that tells at any moment which redex should be chosen. A strategy is often chosen for its efficiency: following a good strategy results in short rewrite sequences. A smart strategy may even avoid infinite computations.

For an actual interpreter, one must also take into account the cost of finding the next redex. This is the reason that many systems implement leftmost-innermost rewriting, although this may produce relatively long reduction sequences, and has a bad termination behaviour [7]. We will use annotations as a simple way to specify strategies.

Strategy Annotations. Consider the following term rewrite system (TRS), where *if*, T and F are function symbols, b, x, y are variables, and with three rules named α, β and γ:

$$\boxed{\begin{array}{l} \alpha : \mathit{if}(T,x,y) \mapsto x \\ \beta : \mathit{if}(F,x,y) \mapsto y \\ \gamma : \ \mathit{if}(b,x,x) \mapsto x \end{array}}$$

A natural way of normalizing the term $\mathit{if}(p,q,r)$ is to first normalize p, and then try rule α or β. This procedure avoids unnecessary reductions inside q or r. In some cases, this could even prevent non-terminating computations. If the first argument doesn't reduce to T or F (for instance because p is an open term or because some rules are missing), the second and third argument must be normalized, and finally the last rule γ is tried. The sketched procedure can be very concisely represented by the following strategy annotation for *if*:

$$strat(if) = [1, \alpha, \beta, 2, 3, \gamma].$$

We say that rule α and β *need* the first argument, because they match on it. Rule γ needs the second and third arguments, because it compares them. We say that the annotation is *in-time* because the arguments of *if* are evaluated before the rules which need them are tried. We say that this annotation is *full* because all argument positions and rules for *if* occur in it.

Another full and in-time annotation for *if* would be $[1, 2, 3, \alpha, \beta, \gamma]$, denoting the left-most innermost strategy. We will define a default annotation, which evaluates all its arguments from left to right, and tries to apply a rule as soon as its needed arguments are evaluated. We call this default strategy the *just-in-time* strategy. Note that $strat(if)$ is the just-in-time strategy.

Contribution. Following [16], we define a normalizing function $norm(t)$, which normalizes a term according to a strategy annotation for all function symbols. It traverses the term once, and computes a normal form in a bottom-up fashion. If a redex is found at a certain position, it is replaced and the search proceeds *at the same position*. So for these strategy annotations, finding the next redex is as efficient as for innermost rewriting. The normalization function has been used as a design to build an actual interpreter in the programming language C.

Viewing normal forms as the correct answers, partial correctness of the normalizer means that if $norm(t) = s$, then s is a normal form of t. We call a strategy annotation *complete* if *norm* is partially correct. Our main result is that full and in-time are sufficient syntactic criteria for completeness. This result applies to any TRS, without restrictions. This generalizes [15], because our restrictions are more liberal, and their proof only works for left-linear TRSs.

The proof yields some extra information: although *norm* continues at the position where the previous redex was found, it is equivalent to a particular *memory-less* strategy. This means that the chosen redex depends on the term only, and not on previous reduction steps. Moreover, $norm(t)$ follows this strategy even for infinite reduction sequences.

A default strategy annotation (called just-in-time) that is full and in-time can be computed automatically, and is satisfactory for many function definitions, including if-then-else and the boolean connectives conjunction and disjunction.

Related Work. Many rewrite (logic) implementations allow the user to specify a strategy. ELAN [2] was the first rewrite implementation, where users can define their own strategies. Rewrite rules are viewed as basic strategies, that can be composed by sequential, alternative and conditional composition. Mechanisms to control non-determinism are also present. In Maude [5] strategies can be defined inside the logic, thanks to the reflection principle of rewrite logic. Stratego [17] incorporates recursive strategies and general traversal patterns. It was shown [12] how these can be defined inside ASF+SDF [4, 3].

All mentioned strategy languages are far richer than the annotations studied in our paper. Those systems advocate a separation between computations (rules) and control (strategies). By writing strategies the user can freely choose when the rules are applied. Important applications are the specification of program transformations. However, these strategies are not always complete, in the sense that a strategy might terminate in a term that still contains redexes. As far as we know, no analysis exists whether subclasses of these strategies are complete, which is an important issue if one is interested in finding normal forms.

Members of the OBJ-family [9, 14, 16] have strategy annotations that are similar to the ones discussed in our paper. In OBJ an annotation is a list of integers. Similar to us, $+i$ denotes the normalization of the i-th's argument. There are two differences. First, instead of mentioning rules individually (our α, β), OBJ uses 0 to denote a reduction at top level with any rule. Our more refined notion allows to assign a priority in applying the rules. Avoiding repetitions of 0 (which would lead to more calls to the matching procedure), in OBJ the only full and in-time annotation for the three *if*-rules mentioned before is the strategy $[1, 2, 3, 0]$, which corresponds to innermost rewriting. The second difference is that OBJ allows $-i$, denoting that argument i is only normalized on demand (i.e. if it is needed for matching with another rule). Such annotations specify lazy rewriting, which we have not studied.

The default strategy of CafeOBJ is similar to our just-in-time annotation. We cite from [14, p. 83]: *For each argument, evaluate the argument before the whole term, if there is a rewrite rule that defines the operator such that, in the place of the argument, a non-variable term appears.* We added to this: "or if in the place of the argument, a non-left-linear variable occurs". This extra condition is necessary for obtaining the completeness result of our paper. It is not clear from [9, 14] whether the OBJ-systems check the completeness of the user-provided annotation.

In [13] and [15] completeness of OBJ-annotations is studied. [13] only considers eager annotations, and proves (Thm. 6.1.12) that full annotations ending in a 0 are complete. This is generalized in [15, Cor. 3.8] by allowing argument positions after the last 0. In a separate correction, those authors indicate that their proof doesn't work for non-left-linear TRSs. Moreover, in [15] the criteria depend on all occurrences of function symbols in left hand sides, where our criteria only depend on head-occurrences. As a consequence, given the left hand sides $g(f(c))$ and $f(x)$, the annotation $f : [0, 1]$ is not allowed by the criteria of [15], where it would be allowed by our criteria. So we generalize the mentioned results, by having more liberal criteria on a larger class of TRSs. On the other hand, [15] also consider criteria for on-demand annotations, which we have not studied.

All normalization functions we found in the literature, e.g. [16, 13, 15] are presented with some memory (either by labeling or by using non-tail-recursive calls). Our correctness proof provides the extra information that the interpreter actually follows a certain memory-less strategy, even in case of divergence.

In [11] a survey of strategies in term rewriting is given. The focus is on *nor-*

malizing strategies for *orthogonal* systems. A strategy is normalizing if it finds a normal form whenever one exists. Orthogonality is a syntactic criterion which ensures confluence. For non-orthogonal systems only few results on normalizing strategies exist. We have not studied which class of annotations gives rise to normalizing strategies. On the other hand, our results apply to non-orthogonal term rewriting systems as well. In [13] decidable criteria on eager OBJ-annotations are provided which are sufficient to ensure normalizing strategies for orthogonal TRSs. These results can probably be adapted to our annotations.

Acknowledgements. The strategy annotations of this paper date back to ideas of Jasper Kamperman and Pum Walters around 1996. At that time we tried a correctness proof along the lines of [8, 6, 7], which failed. After a recent implementation of this strategy I tried a new proof of correctness. I thank Stefan Blom, Mark van den Brand, Jozef Hooman, Bas Luttik, Vincent van Oostrom and Hans Zantema for inspiring discussions and helpful hints. Finally, the referees were very helpful, by pointing out related work.

2 Basic Definitions and Result

2.1 Preliminaries

We take standard definitions from term rewriting [11, 1]. We presuppose a set of variables, and function symbols (f), each expecting a fixed number of arguments, denoted by $arity(f)$. Terms are either variables (x) or a function symbol f applied to n terms, denoted $f(t_1, \ldots, t_n)$, where n is the arity of f. With $head(t)$ we denote the topmost function symbol of t.

A *position* (p, q) is a string of integers. By ε we denote the empty string. With $t|_p$ we denote the subterm of t at position p, which is only well-defined if p is a position in t (see [1] for a formal definition). In that case, $t|_\varepsilon = t$ and $f(t_1, \ldots, t_n)|_{i.p} = t_i|_p$. With $t[s]_p$ we denote the term t in which $t|_p$ is replaced by s. We write $p \leq q$ if p is a prefix of q (i.e. $p.p' = q$ for some p').

A rewrite rule is a pair of terms $l \mapsto r$, where l is not a variable, and all variables occurring in r occur in l as well. A term rewrite system (TRS) is a set of rewrite rules. A substitution is a mapping from variables to terms, and with t^σ we denote the term t with all variables x replaced by $\sigma(x)$. A TRS R induces a rewrite relation on terms as follows: $t \rightarrow_R t[r^\sigma]_p$ if and only if $t|_p = l^\sigma$ for some rule $l \mapsto r \in R$. In this case l^σ is called the redex and r^σ the contractum, and the pair (l^σ, r^σ) is called a *rewrite* in [11]. A normal form is a term t which contains no redex. Note that a redex may have many occurrences in the same term, so in order to uniquely identify the rewrite step, we also need a position p. From the position p the redex l^σ can be reconstructed. For this reason, it is convenient to call the pair (p, r^σ) a *rewrite* of t.

2.2 Strategy Annotations and Strategies

A *strategy annotation* for a function symbol f in TRS R is a list whose elements can be either:

– a number i, with $1 \leq i \leq arity(f)$; or
– a rule $l \mapsto r \in R$, such that $head(l) = f$.

Without loss of generality, we will assume that an annotation *has no duplicates*, i.e. each i occurs at most once (after the first normalization the i'th argument is normal, so a second occurrence of i would not constitute an actual rewrite step). We write $[]$ for the empty annotation and $[x|L]$ for the annotation with head x and tail L. In the sequel i, j, k will range over argument positions, and α, β, γ over rewrite rules. So $[i|L]$ starts with a natural number and $[\alpha|L]$ with a rewrite rule.

An index i is *needed* for a left hand side $f(l_1, \ldots, l_n)$, if l_i is not a variable, or if it is a variable which occurs in l_j, for $i \neq j$. Index i is needed for rule $\alpha : l \mapsto r$ if i is needed for l. A strategy annotation L is *full* for f, if for each i with $1 \leq i \leq arity(f)$, $i \in L$ and for each rule $\alpha : l \mapsto r \in R$ with $head(l) = f$, $\alpha \in L$. A strategy annotation L is *in-time*, if for any α and i such that $L = L_1 \alpha L_2 i L_3$, i is not needed for α. In a full and in-time annotation all needed positions for α occur before α. The distinguishing feature of the notion of needed is as follows:

Lemma 1. *Let $l = f(l_1, \ldots, l_n)$ and let argument i be not needed for l. If $t = l^\sigma$ for some σ, then for any s, $t[s]_i = l^\rho$ for some ρ.*

Proof. For some σ, $t = l^\sigma = f(l_1^\sigma, \ldots, l_n^\sigma)$. Because i is not needed for l, l_i is a variable. Let $\rho = \sigma[l_i := s]$. As i is not needed, l_i doesn't occur in l_j (for $j \neq i$), so $l_j^\rho = l_j^\sigma$, and $l_i^\rho = s$. Hence $l^\rho = t[s]_i$. □

We now define the strategy associated to a strategy annotation. A strategy[1] can be viewed as a partial function that given a term t, yields some rewrite of t, i.e. a pair (q, s) such that $t|_q = l^\sigma$ and $s = r^\sigma$ for some rule $l \mapsto r$ and substitution σ. In this case $t \to_R t[s]_q$. Alternatively the function may yield $\perp$ (undefined – found no rewrite). A *complete strategy* yields $\perp$ on t only if t is a normal form.

In the sequel, a fixed TRS R is supposed, with a fixed strategy annotation *strat*. We write $strat(t)$ as an abbreviation of $strat(head(t))$, i.e. the strategy annotation of its head symbol, where $strat(x) = []$ for variables x. We say that *strat* is full (in-time) if $strat(f)$ is full (in-time) for all symbols f. Next we define $rewr_1(t, L)$, which computes the next rewrite in t according to annotation L. We allow a slight overloading as in $rewr_1(t)$ in case $L = strat(t)$.

Definition 2. *$rewr_1(t) = rewr_1(t, strat(t))$, where:*

$$
\begin{aligned}
rewr_1(t, []) &= \perp \\
rewr_1(t, [l \mapsto r|L]) &= \begin{cases} \textbf{if } t = l^\sigma \textit{ for some } \sigma \\ \quad \textbf{then } (\varepsilon, r^\sigma) \\ \quad \textbf{else } rewr_1(t, L) \end{cases} \\
rewr_1(t, [i|L]) &= \begin{cases} \textbf{if } rewr_1(t|_i) = (q, s) \textit{ for some } q,\ s \\ \quad \textbf{then } (i.q, s) \\ \quad \textbf{else } rewr_1(t, L) \end{cases}
\end{aligned}
$$

[1] This covers deterministic, one-step, memory-less strategies only.

The definition proceeds by induction on t and the strategy-annotation L. In each recursive call, either the term t gets smaller, or it remains equal and the list L gets smaller. Therefore this function terminates either in (q, s) or in $\perp$. We now show that for full annotations, the associated strategy is complete:

Proposition 3. *If strat is full and $rewr_1(t) = \perp$, then t is a normal form*

Proof. The proof is with induction on t. Assume that t is not in normal form. Then it contains a redex, either at top level, or in a proper subterm. We distinguish these two cases:

- Assume that $t = l^\sigma$ for some rule $\alpha : l \mapsto r$. Then, by induction on L one can show: "if $\alpha \in L$ then $rewr_1(t, L)$ is defined". By fullness, $\alpha \in strat(t)$, so $rewr_1(t) = rewr_1(t, strat(t))$ is defined.
- Assume that $t|_i$ contains a redex. Then using the induction hypothesis, one can show with induction on L: "if $i \in L$, then $rewr_1(t, L)$ is defined". By fullness, $i \in strat(t)$, so $rewr_1(t) = rewr_1(t, strat(t))$ is defined. $\square$

2.3 Problem Statement

Given the strategy, the associated reduction sequence can be defined as follows:

$$seq_1(t) = \begin{cases} \textbf{if } rewr_1(t) = (q, s) \text{ for some } q,\, s \\ \quad \textbf{then } t :: seq_1(t[s]_q) \\ \quad \textbf{else } \langle t \rangle \end{cases}$$

By the previous proposition, a normal form can be obtained as $last(seq_1(t))$ (for infinite sequences this is undefined). The computational drawback is that after each step the whole term $t[s]_q$ must be traversed to find the next rewrite. This repeats a lot of work of the previous step. It would be nice if the search could be continued at position q. Therefore we propose the following partial function, $norm(t, L)$, which tries to find a normal form of t, according to annotation L. We allow a slight overloading, as in $norm(t)$, in case L is the fixed strategy annotation *strat*. We view this function as the design for an interpreter.

Definition 4. $norm(t) = norm(t, strat(t))$*, where:*

$$\begin{aligned} norm(t, []) &= t \\ norm(t, [l \mapsto r|L]) &= \begin{cases} \textbf{if } t = l^\sigma \textit{for some } \sigma \\ \quad \textbf{then } norm(r^\sigma) \\ \quad \textbf{else } norm(t, L) \end{cases} \\ norm(t, [i|L]) &= norm(t[norm(t|_i)]_i, L) \end{aligned}$$

Avoiding position-notation, the last clause can be written alternatively as $norm(f(t_1, \ldots, t_i, \ldots, t_n), [i|L]) = norm(f(t_1, \ldots, norm(t_i), \ldots, t_n), L)$.

If t is a non-terminating term, $norm(t)$ might diverge, in which case it is undefined. The next section is devoted to the technical core of this paper, viz. correctness of *norm*. That is, we must prove

Theorem 5. *If strat is in-time, then* $norm(t) = last(seq_1(t))$.

This follows immediately from Propositions 6 and 12. In combination with Propositions 3, we obtain that for full and in-time strategies, if $norm(t) = s$, then s is a normal form.

2.4 Counter examples

It may be illustrative to show why the conditions on annotations are needed. Consider the system with three *if*-rules from the introduction, and an additional rule $T \wedge T \mapsto T$.

The strategy-annotation $[\alpha, \beta, 1, 2, 3, \gamma]$ is not in-time, because α matches on the first argument. Consider the term $\mathit{if}(T \wedge T, x, y)$. Rule α and β are not immediately applicable. After reduction of $T \wedge T$, α and β will not be tried again. So under this annotation, $norm(\mathit{if}(T \wedge T, x, y)) = \mathit{if}(T, x, y)$, which is not normal. Similarly, $[1, \alpha, \beta, \gamma, 2, 3]$ is not in-time, because γ is non-linear in its second and third argument. Under this annotation, $norm(\mathit{if}(x, T \wedge T, T)) = \mathit{if}(x, T, T)$. This is not normal, due to the fact that γ was tried too early. Finally, $[\alpha, \beta, 2, 3, \gamma]$ is not full, because argument position 1 is missing. Under this annotation $norm(\mathit{if}(T \wedge T, x, y)) = \mathit{if}(T \wedge T, x, y)$, which is not normal.

These examples show that the conditions cannot be dropped in general. In certain cases they could be weakened. For instance in $\alpha : f(x) \mapsto g(x)$, the annotation $[\alpha]$ is not full, but this is harmless because α applies to any term with head symbol f. This weakening is inessential, because the behaviour of the interpreter is exactly the same as with the full strategy $[\alpha, 1]$.

3 Correctness Proof

The proof has two distinct parts. First we identify the series of redexes contracted by *norm*. This is not straightforward due to its doubly recursive definition. By program transformation we find an equivalent function $norm_2$, where the double recursion is eliminated in favour of a stack containing the return points. From this definition the series of redexes can be easily extracted (Section 3.1).

At first sight, *norm* doesn't follow a memory-less strategy, because after finding a redex at position q, it continues its search from q onwards. Therefore, the found rewrite depends on a certain internal "state" or "memory", S. Hence the strategy will be a function of the form $rewr_2(t, S) = (q, s, R)$, where (q, s) is the found rewrite, and R denotes the next state. In the sequel, the triple (q, s, R) will also be called a rewrite.

The second step in the proof (Section 3.2) shows that if the annotation is in-time, then the state doesn't influence the rewrite found. That is, the next rewrite can be found in two equivalent ways: $rewr_2(t[s]_q, S) = rewr_2(t[s]_q, R)$. The proof is then finished by the observation that for the initial state I, $rewr_2(t, I) = rewr_1(t)$.

3.1 Making Recursion Explicit

This section eliminates the double recursion from *norm*. In the first transformation, we replace recursion on subterms by recursion on positions. This makes it possible to return to a previous stage. First specify:

$$norm_1(t, p, L) = t[norm(t|_p, L)]_p$$

Next, using this definition and the defining equations for *norm*, we can calculate (Section A.1) the following recursive definition for $norm_1$:

$$\begin{aligned}
norm_1(t, p, []) &= t \\
norm_1(t, p, [l \mapsto r|L]) &= \begin{cases} \textbf{if } t|_p = l^\sigma \text{ for some } \sigma \\ \quad \textbf{then } norm_1(t[r^\sigma]_p, p, strat(r^\sigma)) \\ \quad \textbf{else } norm_1(t, p, L) \end{cases} \\
norm_1(t, p, [i|L]) &= norm_1(norm_1(t, p.i, strat(t|_{p.i})), p, L)
\end{aligned}$$

Next, we eliminate the double recursion in favour of a stack, which is a list of pairs of previous positions and strategies that are still to be executed. To this end, we introduce the recursive specification for $norm_2$:

$$\begin{aligned}
norm_2(t, []) &= t \\
norm_2(t, [(p, L)|S]) &= norm_2(norm_1(t, p, L), S)
\end{aligned}$$

From this specification, and the recursive equations derived for $norm_1$, we can derive (Section A.2) the following recursive equations for $norm_2$:

$$\begin{aligned}
norm_2(t, []) &= t \\
norm_2(t, [(p, [])|S]) &= norm_2(t, S) \\
norm_2(t, [(p, [l \mapsto r|L])|S]) &= \begin{cases} \textbf{if } t|_p = l^\sigma \text{ for some } \sigma \\ \quad \textbf{then } norm_2(t[r^\sigma]_p, [(p, strat(r^\sigma))|S]) \\ \quad \textbf{else } norm_2(t, [(p, L)|S]) \end{cases} \\
norm_2(t, [(p, [i|L])|S]) &= norm_2(t, [(p.i, strat(t|_{p.i})), (p, L)|S])
\end{aligned}$$

From this explicit definition it is easy to guess the next rewrite that $norm_2$ will take, given the current state (stack) S. This gives rise to the following definition $rewr_2(t, S)$. The result will be either $\perp$, or a triple (q, s, T), where (q, s) denotes the rewrite as previously, and T is the stack after replacing $t[s]_q$.

$$\begin{aligned}
rewr_2(t, []) &= \perp \\
rewr_2(t, [(p, [])|S]) &= rewr_2(t, S) \\
rewr_2(t, [(p, [l \mapsto r|L])|S]) &= \begin{cases} \textbf{if } t|_p = l^\sigma \text{ for some } \sigma \\ \quad \textbf{then } (p, r^\sigma, [(p, strat(r^\sigma))|S]) \\ \quad \textbf{else } rewr_2(t, [(p, L)|S]) \end{cases} \\
rewr_2(t, [(p, [i|L])|S]) &= rewr_2(t, [(p.i, strat(t|_{p.i})), (p, L)|S])
\end{aligned}$$

Given this function $rewr_2$, we can define a second rewrite sequence. This time the sequence is not memory-less, because each step changes the state (stack S)

of the system.

$$seq_2(t,S) = \begin{cases} \textbf{if } rewr_2(t,S) = (q,s,R) \text{ for some } q,\ s,\ R \\ \quad \textbf{then } t :: seq_2(t[s]_q, R) \\ \quad \textbf{else } \langle t \rangle \end{cases}$$

In order to check that $rewr_2(t,S)$ indeed yields the next rewrite taken by $norm_2$ in state S, one can take the specification

$$norm_3(t,S) = last(seq_2(t,S))$$

Using the definitions of seq_2 and $rewr_2$, one can derive recursive equations for $norm_3$ (Section A.3), which appear to be exactly the same as those for $norm_2$. We summarize the result of this section:

Proposition 6. $norm(t, strat(t)) = last(seq_2(t, [(\varepsilon, strat(t))]))$.

Proof. First, $norm_1(t,p,L) = norm_2(norm_1(t,p,L),[]) = norm_2(t,[(p,L)])$. Also, $norm(t,L) = t[norm(t|_\varepsilon, L)]_\varepsilon = norm_1(t,\varepsilon,L)$. The result follows because by the previous remark $norm_2(t,S) = last(seq_2(t,S))$. □

3.2 Connecting Memory-less and State-based Strategies

It is now sufficient to prove that $seq_2(t,[(\varepsilon, strat(t))]) = seq_1(t)$. This is the case if $rewr_2(t,S)$ yields the same rewrite as $rewr_1(t)$, for all reachable states S. To this end, we first show that the stack will be always *well-formed* (defined below). Then we show that in fact $rewr_2$ is actually independent of the current state, i.e. $rewr_2(t,S) = rewr_2(t,[(\varepsilon, strat(t))])$ for all stacks S encountered (Lemma 10). Finally, we show that $rewr_2(t,[(\varepsilon, strat(t))]) = rewr_1(t)$ (Lemma 11).

We now define the set of well-formed stacks w.r.t. t. Intuitively, the positions in the stack form a proper path in t, all annotations on the stack are in-time, and nodes can be visited at most once.

Definition 7. *The set of well-formed stacks w.r.t. t are defined inductively as follows:*

- $[]$ *is well-formed.*
- $[(\varepsilon, L)]$ *is well-formed, if L is an in-time strategy annotation for $head(t)$.*
- $[(p.i, K), (p,L)|S]$ *is well-formed, if $[(p,L)|S]$ is well-formed, and $p.i$ is a position in t and K is an in-time strategy annotation for $head(t|_{p.i})$ and $i \notin L$.*

Lemma 8. *If strat is in-time, then it is an invariant of $rewr_2$ (and $norm_2$ and seq_2) that the stack is well-formed.*

Proof. $[(\varepsilon, strat(t))]$ is well-formed (initial condition) and the property is preserved in all recursive calls. This relies on the assumption that strategy annotations contain no duplicates. □

The following technical lemma is the core of the proof. It shows that if searching in t from state S yields a rewrite (q, s, R), then the search in t from S can be mimicked in $t[s]_q$, and will lead again to state R (see Figure 1). The key of the proof is that if rule α is not applicable at $t|_p$, then a reduction inside $t|_p$ can only occur in an argument which is not needed, so also in $t[s]_q|_p$ rule α will not be applicable. The full proofs of Lemma 9–12 are in Appendix B.

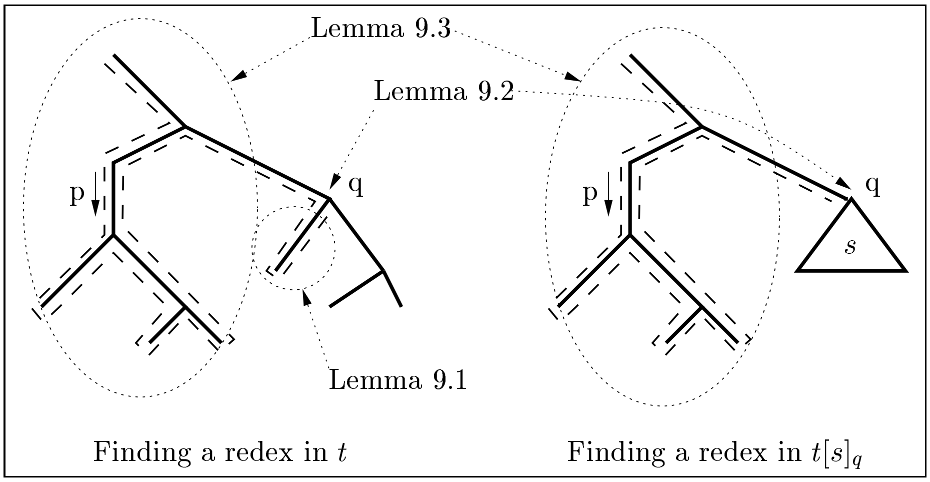

Fig. 1. The search in t can be mimicked in $t[s]_q$.

Lemma 9. *Let strat be in-time. Let $[(p, L)|S]$ be a well-formed stack. Assume $rewr_2(t, [(p, L)|S]) = (q, s, R)$, for some q, s and R. Then we have:*

1. *If $q \not\geq p$ then $rewr_2(t, [(p, L)|S]) = rewr_2(t, S)$.*
2. *If $p = q$ then $R = [(p, strat(s))|S]$.*
3. *If $q \not\leq p$, then $rewr_2(t[s]_q, [(p, L)|S]) = rewr_2(t[s]_q, R)$.*

Proof. 1. Induction on the structure of $t|_p$, and for equal $t|_p$ on the structure of L. The proof proceeds by case distinction on L.
2. Induction on L, using (1) in case $L = [i|L']$.
3. Starting with stack $[(p, L)|S]$ and term t, $rewr_2$ reduces in a number of steps to (q, s, R). The proof proceeds by mimicking this reduction starting with the same stack in term $t[s]_q$. The proof is by induction on the number of recursive calls of $rewr_2(t, [(p, L)|S])$ to (q, s, R).

Lemma 10. *Let strat be in-time. If $rewr_2(t, [(\varepsilon, strat(t))]) = (q, s, R)$ then $rewr_2(t[s]_q, R) = rewr_2(t[s]_q, [(\varepsilon, strat(t[s]_q))])$.*

Proof. If $q = \varepsilon$, this follows from the previous lemma (2). Otherwise, $q > \varepsilon$, and this follows from the previous lemma (3). □

Finally, we prove the relationship between $rewr_1$ and $rewr_2$:

Lemma 11.

1. $rewr_1(t) = (q, s) \iff$ *for some* R, $rewr_2(t, [(\varepsilon, strat(t))]) = (q, s, R)$
2. $rewr_1(t) = \bot \iff rewr_2(t, [(\varepsilon, strat(t))]) = \bot$

Proof. The proof follows from the following propositions, which can be proved by simultaneous induction on the structure of $t|_p$ and, for equal $t|_p$, on L.

1. if $rewr_1(t|_p, L) = (q, s)$, then for some R, $rewr_2(t, [(p, L)|S]) = (p.q, s, R)$.
2. if $rewr_1(t|_p, L) = \bot$, then $rewr_2(t, [(p, L)|S]) = rewr_2(t, S)$. □

Proposition 12. *If strat is in-time, then* $seq_2(t, [(\varepsilon, strat(t))]) = seq_1(t)$.

Proof. Using Lemma 10 and Lemma 11 the definition of $seq_2(t, [(\varepsilon, strat(t))])$ can be transformed into the definition of $seq_1(t)$. □

4 Implementation and Applications

The just-in-time strategy is defined as follows. For any function symbol f, with arity n, take the list $[1, \ldots, n]$. Next, insert each rule α directly after the last argument position that it needs (due to matching or non-linearity). If several rules are placed between i and $i+1$, the textual order of the original specification is maintained.

We applied this interpreter to several specifications, with satisfactory results. The application domain is verification of distributed systems, where a system specification has a process part and an algebraic data specification part. A theorem prover is being implemented, to solve boolean combinations of equalities over the algebraic data specification, by a combination of BDDs and term rewriting, along the lines of [10]. In many cases, innermost rewriting didn't lead to a normal form. Below we list a number of rules in order to illustrate this point:

$$\begin{aligned}
\alpha &: & F \vee x &\mapsto x \\
\beta &: & T \vee x &\mapsto T \\
\gamma &: & count(l) &\mapsto if(empty(l), 0, 1 + count(tail(l))) \\
\delta &: & div(m, n) &\mapsto if(m < n, 0, 1 + div(m - n, n)) \\
\epsilon &: & rem(m, n) &\mapsto if(m < n, m, rem(m - n, n))
\end{aligned}$$

On closed lists, *count* terminates with the just-in-time annotation $[\gamma, 1]$ (assuming standard definitions of *empty* and *tail*), but it diverges with innermost rewriting. This could be solved by replacing *if* by pattern matching, providing rules for $count([])$ and $count([x|L])$. However, this solution is not easily available for *div* and *rem* (division and remainder). Assuming standard definitions of $-$ and $<$, these functions terminate on closed numerals m and n for positive n, with the just-in-time annotation $[\delta/\epsilon, 1, 2]$, but they diverge with innermost rewriting. The just-in-time annotation for $\vee$ is $[1, \alpha, \beta, 2]$. With this annotation,

$eq(n, 0) \vee div(m, n) < m$ terminates for all numerals m and n, even for $n = 0$, provided $eq(0, 0) \to T$.

The just-in-time strategy works from left-to-right. Sometimes it is more efficient to start with another argument. One could devise an algorithm to transform a TRS by reordering the arguments to the function symbols. In general, one could study which of the full and in-time annotations yields the most efficient strategy for a given TRS.

We have constructed a C-implementation of the function *norm*, which acts as an interpreter of a given TRS annotated by some strategy. As a default, the system computes the just-in-time strategy during initialization. There are some well-known implementation issues, which we now shortly mention.

First, a rule $\alpha : f(x) \mapsto g(x, x, x)$ with annotation $[\alpha, 1]$, would copy all redexes in x three times. Therefore, in our implementation we use maximally shared terms (DAGs), in which x occurs only once. The implementation uses the efficient annotated term library [3], providing maximally shared terms, garbage collection and term tables (for memoization) for free.

Another issue is that in a rule $\alpha : f(x) \mapsto g(x)$ with annotation $[1, \alpha]$, first x is normalized by f to n, and then $g(n)$ is called. Because g doesn't know that n is normal, it will traverse the whole n. To avoid this, all subterms which are known to be normal are marked. So g will get a marked argument, which it doesn't traverse. If the annotation would be $[\alpha, 1]$, as with the just-in-time strategy, g would get x unmarked. A similar approach can be found in [16].

Finally, consider the rule $\alpha : f(x) \mapsto g(h(x))$. In innermost rewriting, x can be normalized, passing the result to function f. Then f calls function h and g, respectively. These functions expect normal forms as arguments. Note that the term $h(x)$ is not actually built by f. With the annotation $[\alpha, 1]$ this is not possible. We have to build at least the term $h(x)$, which must be passed to g *before normalization*. At this point we have to face some penalty compared to innermost rewriting, because term formation is relatively expensive, especially for maximally shared terms.

5 Future Work

We mentioned that in many examples, the just-in-time strategy has a better termination behaviour than innermost rewriting. We now generalize this to the following:

Conjecture 13. Let R be a TRS with a full and in-time strategy annotation *strat*. If t is strongly normalizing under the leftmost-innermost strategy, then strategy *strat* on t terminates.

(If the rules are non-root-overlapping, "strongly" could be dropped). This would be an important result, implicating that a rewrite implementation can make the transition from leftmost-innermost to just-in-time rewriting, without repercussions for the users. The improvement would be conservative, in the sense that all previous examples still terminate, and some more.

We restricted attention to deterministic strategies. By dividing the annotations in groups, one could denote non-deterministic strategies. I.e. the innermost strategy (not just *left-most* innermost) is specified by an annotation like $[\{1,2,3\},\{\alpha,\beta,\gamma\}]$. In the proof machinery, rewrites must be replaced by sets of rewrites, and the deterministic sequences by transition systems. In fact our proof is a bisimulation proof on sequences and this technique carries over to transition systems in a straightforward way[2]. However, the straightforward implementation of annotation $[1,2]$ would either choose to normalize the first argument completely, or the second which is not memory-less. A memory-less strategy would allow alternations of steps in the first and second argument.

Another indication that the non-deterministic case is different is that the following TRS (after Toyama), is a counter-example to our conjecture:

$$\alpha : f(0,1,x) \mapsto f(x,x,x) \qquad \beta : 2 \mapsto 0 \qquad \gamma : 2 \mapsto 1$$

Any innermost reduction of $f(0,1,2)$ terminates. But the non-deterministic strategy indicated by $f : [0,1,\alpha,2]$ and $2 : [\{\beta,\gamma\}]$ allows an infinite reduction $f(0,1,2) \to f(2,2,2) \to f(0,2,2) \to f(0,1,2)$.

A Program Transformations

A.1 From *norm* to *norm*$_1$

Take as definition:

$$norm_1(t,p,L) = t[norm(t|_p,L)]_p$$

Then calculate:

$$\begin{aligned} norm_1(t,p,[]) &= t[norm(t|_p,[])]_p \\ &= t[t|_p]_p \\ &= t \end{aligned}$$

Let $\alpha : l \mapsto r$. If $t|_p = l^\sigma$, for some σ, then:

$$\begin{aligned} norm_1(t,p,[\alpha|L]) &= t[norm(t|_p,[\alpha|L])]_p \\ &= t[norm(r^\sigma, strat(r^\sigma))]_p \\ &= t[r^\sigma]_p[norm(t[r^\sigma]_p|_p, strat(r^\sigma))]_p \\ &= norm_1(t[r^\sigma]_p, p, strat(r^\sigma)) \end{aligned}$$

If $t|_p \neq l^\sigma$, for any σ, then:

$$\begin{aligned} norm_1(t,p,[\alpha|L]) &= t[norm(t|_p,[\alpha|L])]_p \\ &= t[norm(t|_p,L)]_p \\ &= norm_1(t,p,L) \end{aligned}$$

[2] History-sensitive non-deterministic strategies form a coalgebra: $State \to \mathcal{P}(Rewrite \times State)$, where $State \simeq Term \times Memory$. The strategy is memory-less if $State \simeq Term$; it is deterministic if the set on the right is a singleton.

Finally,

$$
\begin{aligned}
norm_1(t, p, [i|L]) &= t[norm(t|_p, [i|L])]_p \\
&= t[norm(t|_p[norm(t|_p|_i, strat(t|_p|_i))]_i, L)]_p \\
&= t[norm(t|_p[norm(t|_{p.i}, strat(t|_{p.i}))]_i, L)]_p \\
&= \quad \{\text{introduce abbreviation } A\} \\
&\quad\; t[norm(t|_p[A]_i, L)]_p \\
&= t[A]_{p.i}[norm(t|_p[A]_i, L)]_p \\
&= t[A]_{p.i}[norm(t[A]_{p.i}|_p, L)]_p \\
&= norm_1(t[A]_{p.i}, p, L) \\
&= norm_1(t[norm(t|_{p.i}, strat(t|_{p.i}))]_{p.i}, p, L) \\
&= norm_1(norm_1(t, p.i, strat(t|_{p.i})), p, L)
\end{aligned}
$$

A.2 From $norm_1$ to $norm_2$

Take as definition:

$$
\begin{aligned}
norm_2(t, []) &= t \\
norm_2(t, [(p, L)|S]) &= norm_2(norm_1(t, p, L), S)
\end{aligned}
$$

Then calculate:

$$
\begin{aligned}
norm_2(t, [(p, [])|S]) &= norm_2(norm_1(t, p, []), S) \\
&= norm_2(t, S)
\end{aligned}
$$

Let $\alpha : l \mapsto r$, if $t|_p = l^\sigma$ for some σ, then:

$$
\begin{aligned}
norm_2(t, [(p, [\alpha|L])|S]) &= norm_2(norm_1(t, p, [\alpha|L]), S) \\
&= norm_2(norm_1(t[r^\sigma]_p, p, strat(r^\sigma)), S) \\
&= norm_2(t[r^\sigma]_p, [(p, strat(r^\sigma))|S])
\end{aligned}
$$

If $t|_p \neq l^\sigma$ for any σ, then:

$$
\begin{aligned}
norm_2(t, [(p, [\alpha|L])|S]) &= norm_2(norm_1(t, p, [\alpha|L]), S) \\
&= norm_2(norm_1(t, p, L), S) \\
&= norm_2(t, [(p, L)|S])
\end{aligned}
$$

Finally,

$$
\begin{aligned}
norm_2(t, [(p, [i|L])|S]) &= norm_2(norm_1(t, p, [i|L]), S) \\
&= norm_2(norm_1(norm_1(t, p.i, strat(t|_{p.i})), p, L), S) \\
&= norm_2(norm_1(t, p.i, strat(t|_{p.i})), [(p, L)|S]) \\
&= norm_2(t, [(p.i, strat(t|_{p.i})), (p, L)|S])
\end{aligned}
$$

A.3 From $last(seq_2)$ to $norm_3$

Take as a definition:

$$norm_3(t, S) = last(seq_2(t, S))$$

We will use several times that if $rewr_2(t, S) = rewr_2(t, R)$, then $seq_2(t, S) = seq_2(t, R)$. Now calculate:

$$\begin{aligned} norm_3(t, []) &= last(seq_2(t, [])) \\ &= \quad \{rewr_2(t, []) = \bot\} \\ &\quad\ last(\langle t \rangle) \\ &= t \end{aligned}$$

Next,

$$\begin{aligned} norm_3(t, [(p, [])|S]) &= last(seq_2(t, [(p, [])|S])) \\ &= \quad \{rewr_2(t, [(p, [])|S]) = rewr_2(t, S)\} \\ &\quad\ last(seq_2(t, S)) \\ &= norm_3(t, S) \end{aligned}$$

Next, if $\alpha : l \mapsto r$ and $t|_p = l^\sigma$, for some σ:

$$\begin{aligned} norm_3(t, [(p, [\alpha|L])|S]) &= last(seq_2(t, [(p, [\alpha|L])|S])) \\ &= \quad \{rewr_2(t, [(p, [\alpha|L])|S]) = (p, r^\sigma, [(p, strat(r^\sigma))|S])\} \\ &\quad\ last(t :: seq_2(t[r^\sigma]_p, [(p, strat(r^\sigma))|S])) \\ &= last(seq_2(t[r^\sigma]_p, [(p, strat(r^\sigma))|S])) \\ &= norm_3(t[r^\sigma]_p, [(p, strat(r^\sigma))|S]) \end{aligned}$$

Otherwise, if $t|_p \neq l^\sigma$ for any σ:

$$\begin{aligned} norm_3(t, [(p, [\alpha|L])|S]) &= last(seq_2(t, [(p, [\alpha|L])|S])) \\ &= \quad \{rewr_2(t, [(p, [\alpha|L])|S]) = rewr_2(t, [(p, L)|S])\} \\ &\quad\ last(seq_2(t, [(p, L)|S])) \\ &= norm_3(t, [(p, L)|S]) \end{aligned}$$

Finally,

$$\begin{aligned} &norm_3(t, [(p, [i|L])|S]) \\ &= last(seq_2(t, [(p, [i|L])|S])) \\ &= \quad \{rewr_2(t, [(p, [i|L])|S]) = rewr_2(t, [(p.i, strat(t|_{p.i})), (p, L)|S])\} \\ &\quad\ last(seq_2(t, [(p.i, strat(t|_{p.i})), (p, L)|S])) \\ &= norm_3(t, [(p.i, strat(t|_{p.i})), (p, L)|S]) \end{aligned}$$

B Full Proofs of Lemma 9–12

Proof of Lemma 9.1. Let *strat* be in-time, let $[(p,L)|S]$ be a well-formed stack, and assume that $rewr_2(t,[(p,L)|S]) = (q,s,R)$, where $q \not\geq p$. We have to prove: $rewr_2(t,[(p,L)|S]) = rewr_2(t,S)$. This is by induction on $t|_p$ and within that on L. We proceed by case distinction on L.

- $L = []$: In this case, $rewr_2(t,[(p,[])|S]) = rewr_2(t,S)$ by definition.
- $L = [l \mapsto r|L']$: If $l^\sigma = t|_p$, then $p = q$, in contradiction with the assumption $q \not\geq p$. So $l^\sigma \neq t|_p$ for any σ. Then

$$\begin{aligned} rewr_2(t,[(p,L)|S]) &= rewr_2(t,[(p,L')|S]) \\ &= \{\text{Induction Hypothesis } (L' < L)\} \\ &\quad rewr_2(t,S) \end{aligned}$$

- $L = [i|L']$: First note that if $q \geq p.i$ then $q \geq p$.

$$\begin{aligned} rewr_2(t,[(p,L)|S]) &= rewr_2(t,[(p.i, strat(t|_{p.i})),(p,L')|S]) \\ &= \{\text{Induction Hypothesis } (t|_{p.i} < t|_p)\} \\ &\quad rewr_2(t,[(p,L')|S]) \\ &= \{\text{Induction Hypothesis } (L' < L)\} \\ &\quad rewr_2(t,S) \end{aligned}$$

□

Proof of Lemma 9.2. Let *strat* be in-time, let $[(p,L)|S]$ be a well-formed stack, and let $rewr_2(t,[(p,L)|S]) = (p,s,R)$. We have to prove: $R = [(p, strat(s))|S]$. This is by induction on L:

- $L = []$: Impossible, because p is not revisited from stack S (here well-formedness of $[(p,L)|S]$ is used).
- $L = [i|L']$:

$$\begin{aligned} (p,s,R) &= rewr_2(t,[(p,L)|S]) \\ &= rewr_2(t,[(p.i, strat(t|_{p.i})),(p,L')|S]) \\ &= \{p \not\geq p.i, \text{ so use Lemma 9.1}\} \\ &\quad rewr_2(t,[(p,L')|S]) \end{aligned}$$

  Hence by Induction Hypothesis $(L' < L)$, $R = [(p, strat(s))|S]$.
- $L = [l \mapsto r|L']$: If $t|p = l^\sigma$, for some σ, then $R = [(p, strat(s))|S]$ by definition of $rewr_2$. Otherwise, $t|_p \neq l^\sigma$, for any σ, then

$$(p,s,R) = rewr_2(t,[(p,L)|S]) = rewr_2(t,[(p,L')|S])$$

  Hence by Induction Hypothesis, $R = [(p, strat(s))|S]$. □

Proof of Lemma 9.3. Let *strat* be in-time, let $[(p, L)|S]$ be a well-formed stack, and assume that $\mathit{rewr}_2(t, [(p, L)|S]) = (q, s, R)$, with $q \not\leq p$. We prove that $\mathit{rewr}_2(t[s]_q, [(p, L)|S]) = \mathit{rewr}_2(t[s]_q, R)$.

Starting with stack $[(p, L)|S]$ and term t, rewr_2 reduces in a number of steps to (q, s, R). The proof proceeds by mimicking this reduction starting with the same stack in term $t[s]_q$ (see Figure 1). The proof is by induction on the number of recursive calls of $\mathit{rewr}_2(t, [(p, L)|S])$ to (q, s, R). Distinguish cases for L.

- $L = []$. First, $\mathit{rewr}_2(t, [(p, [])|S]) = \mathit{rewr}_2(t, S)$ and $\mathit{rewr}_2(t[s]_q, [(p, [])|S]) = \mathit{rewr}_2(t[s]_q, S)$. $S \neq []$, for then the result would be $\perp$. By well-formedness of S, $p = p'.j$ and $S = [(p', L')|S']$ for some p', L', S'. Note that $q \not\leq p'$. Hence by induction hypothesis, $\mathit{rewr}_2(t[s]_q, S) = \mathit{rewr}_2(t[s]_q, R)$.
- $L = [i|L']$. Then $\mathit{rewr}_2(t, [(p.i, \mathit{strat}(t|_{p.i})), (p, L')|S]) = \mathit{rewr}_2(t, [(p, [i|L'])|S]) = (q, s, R)$. Distinguish cases:
  - If $p.i = q$, then by Lemma 9.2, $R = [(p.i, \mathit{strat}(s)), (p, L')|S]$.

$$\begin{aligned}\mathit{rewr}_2(t[s]_q, [(p, [i|L'])|S]) &= \mathit{rewr}_2(t[s]_q, [(p.i, \mathit{strat}(t[s]_q|_{p.i})), (p, L')|S]) \\ &= \mathit{rewr}_2(t[s]_q, [(p.i, \mathit{strat}(s)), (p, L')|S]) \\ &= \mathit{rewr}_2(t[s]_q, R)\end{aligned}$$

  - Otherwise, if $p.i \neq q$, then $q \not\leq p.i$. Note that the new stack is well-formed, because $i \notin L'$ by the assumption that annotations have no duplicates. So the induction hypothesis can be used.

$$\begin{aligned}\mathit{rewr}_2(t[s]_q, [(p, [i|L'])|S]) &= \mathit{rewr}_2(t[s]_q, [(p.i, \mathit{strat}(t[s]_q|_{p.i})), (p, L')|S]) \\ &= \quad \{\mathit{head}(t[s]_q|_{p.i}) = \mathit{head}(t|_{p.i})\} \\ &\quad\; \mathit{rewr}_2(t[s]_q, [(p.i, \mathit{strat}(t|_{p.i})), (p, L')|S]) \\ &= \quad \{\text{By Induction Hypothesis}\} \\ &\quad\; \mathit{rewr}_2(t[s]_q, R)\end{aligned}$$

- $L = [l \mapsto r|L']$. If $t|_p = l^\sigma$ for some σ, then $(q, s, R) = \mathit{rewr}_2(t, [(p, L)|S]) = (p, r^\sigma, [(p, \mathit{strat}(r^\sigma))|S])$, which contradicts $q \not\leq p$ (in fact this case is dealt with in part 2).
  Hence $t|_p \neq l^\sigma$ for any σ. We first prove that $t[s]_q|_p \neq l^\sigma$ for any σ. This is done by distinguishing two cases:
  - If $q \not> p$, then $t[s]_q|_p = t|_p$, so $t[s]_q|_p \neq l^\sigma$ for any σ.
  - If $q > p$, then $q = p.i.p'$ for some i, p'. In this case $i \in L$, because $p.i$ will not be revisited from S by the well-formedness of S. Because L is in-time, argument i is not needed by rule $l \mapsto r$, so by Lemma 1, $t[s]_q|_p \neq l^\sigma$, for any σ.

  Now $(q, s, R) = \mathit{rewr}_2(t, [(p, L)|S]) = \mathit{rewr}_2(t, [(p, L')|S])$. Similarly, we have $\mathit{rewr}_2(t[s]_q, [(p, L)|S]) = \mathit{rewr}_2(t[s]_q, [(p, L')|S])$. By induction hypothesis the latter equals $\mathit{rewr}_2(t[s]_q, R)$. □

Proof of Lemma 10. Let *strat* be in-time, and let $\mathit{rewr}_2(t, [(\varepsilon, \mathit{strat}(t))]) = (q, s, R)$. We must prove: $\mathit{rewr}_2(t[s]_q, R) = \mathit{rewr}_2(t[s]_q, [(\varepsilon, \mathit{strat}(t[s]_q))])$.

- If $q = \varepsilon$, then

$$\begin{aligned} \mathit{rewr_2}(t[s]_\varepsilon, R) = &\quad \{\text{by Lemma 9.2}\} \\ &\quad \mathit{rewr_2}(t[s]_\varepsilon, [(\varepsilon, \mathit{strat}(s))]) \\ = &\quad \mathit{rewr_2}(t[s]_\varepsilon, [(\varepsilon, \mathit{strat}(t[s]_\varepsilon))]) \end{aligned}$$

- Otherwise, $q > \varepsilon$, so $q \not\leq \varepsilon$. Then we have:

$$\begin{aligned} \mathit{rewr_2}(t[s]_q, R) = &\quad \{\text{by Lemma 9.3}\} \\ &\quad \mathit{rewr_2}(t[s]_q, [(\varepsilon, \mathit{strat}(t))]) \\ = &\quad \{\mathit{head}(t[s]_q) = \mathit{head}(t)\} \\ &\quad \mathit{rewr_2}(t[s]_q, [(\varepsilon, \mathit{strat}(t[s]_q))]) \end{aligned}$$

Proof of Lemma 11. We have to prove the following:

1. if $\mathit{rewr_1}(t|_p, L) = (q, s)$, then for some R, $\mathit{rewr_2}(t, [(p, L)|S]) = (p.q, s, R)$.
2. if $\mathit{rewr_1}(t|_p, L) = \perp$, then $\mathit{rewr_2}(t, [(p, L)|S]) = \mathit{rewr_2}(t, S)$.

The proof is by simultaneous induction, on $t|_p$ and within that on L. The proof proceeds by case distinction on L:

- $L = []$:
  1. $\mathit{rewr_1}(t|_p, L) = (q, s)$: Impossible
  2. $\mathit{rewr_1}(t|_p, L) = \perp$: By definition, $\mathit{rewr_2}(t, [(p, [])|S]) = \mathit{rewr_2}(t, S)$.
- $L = [i|L']$:
  1. $\mathit{rewr_1}(t|_p, L) = (q, s)$: Distinguish cases:
     - $\mathit{rewr_1}(t|_{p.i}, \mathit{strat}(t|_{p.i})) = \perp$: Then $\mathit{rewr_1}(t|_p, L') = \mathit{rewr_1}(t|_p, L) = (q, s)$, and

$$\begin{aligned} \mathit{rewr_2}(t, [(p, L)|S]) &= \mathit{rewr_2}(t, [(p.i, \mathit{strat}(t|_{p.i})), (p, L')|S]) \\ &= \quad \{\text{By Induction Hypothesis (2) } (t|_{p.i} < t|_p)\} \\ &\quad\ \mathit{rewr_2}(t, [(p, L')|S]) \\ &= \quad \{\text{By Induction Hypothesis (1) } (L' < L)\} \\ &\quad\ (p.q, s, R) \text{ for some } R \end{aligned}$$

     - $\mathit{rewr_1}(t|_{p.i}, \mathit{strat}(t|_{p.i})) = (q', s')$: Then $q = i.q'$ and $s' = s$.

$$\begin{aligned} \mathit{rewr_2}(t, [(p, L)|S]) &= \mathit{rewr_2}(t, [(p.i, \mathit{strat}(t|_{p.i})), (p, L')|S]) \\ &= \quad \{\text{By Induction Hypothesis (1) } (t|_{p.i} < t|_p)\} \\ &\quad\ (p.i.q', s', R) \text{ for some } R \\ &= (p.q, s, R) \end{aligned}$$

  2. $\mathit{rewr_1}(t|_p, L) = \perp$: Then both $\mathit{rewr_1}(t|_{p.i}, \mathit{strat}(t|_{p.i})) = \perp$ and also $\mathit{rewr_1}(t|_p, L') = \perp$. Hence

$$\begin{aligned} \mathit{rewr_2}(t, [(p, L)|S]) &= \mathit{rewr_2}(t, [(p.i, \mathit{strat}(t|_{p.i})), (p, L')|S]) \\ &= \quad \{\text{By Induction Hypothesis (2) } (t|_{p.i} < t|_p)\} \\ &\quad\ \mathit{rewr_2}(t, [(p, L')|S]) \\ &= \quad \{\text{By Induction Hypothesis (2) } (L' < L)\} \\ &\quad\ \mathit{rewr_2}(t, S) \end{aligned}$$

- $L = [l \mapsto r|L']$:
  1. $\mathit{rewr_1}(t|_p, L) = (q, s)$: Distinguish cases.
     - If $t|_p = l^\sigma$ for some σ, then $(q, s) = \mathit{rewr_1}(t|_p, L) = (\varepsilon, r^\sigma)$ and

$$\begin{aligned} \mathit{rewr_2}(t, [(p, L)|S]) &= (p, r^\sigma, [(p, \mathit{strat}(r^\sigma))|S]) \\ &= (p.\varepsilon, r^\sigma, R) \text{ for some } R \end{aligned}$$

     - Otherwise, if $t|_p \neq l^\sigma$ for any σ, then $(q, s) = \mathit{rewr_1}(t|_p, L) = \mathit{rewr_1}(t|_p, L')$, and

$$\begin{aligned} \mathit{rewr_2}(t, [(p, L)|S]) &= \mathit{rewr_2}(t, [(p, L')|S]) \\ &= \ \{\text{By Induction Hypothesis (1) } (L' < L)\} \\ &\quad\ (p.q, s, R) \text{ for some } R \end{aligned}$$

  2. $\mathit{rewr_1}(t|_p, L) = \bot$: Then $t|_p \neq l^\sigma$ for any σ. So $\bot = \mathit{rewr_1}(t|_p, L) = \mathit{rewr_1}(t|_p, L')$, and

$$\begin{aligned} \mathit{rewr_2}(t, [(p, L)|S]) &= \mathit{rewr_2}(t, [(p, L')|S]) \\ &= \ \{\text{By Induction Hypothesis (2) } (L' < L)\} \\ &\quad\ \mathit{rewr_2}(t, S) \end{aligned}$$

Proof of Proposition 12. We must prove that $\mathit{seq}_2(t, [(\varepsilon, \mathit{strat}(t))]) = \mathit{seq}_1(t)$. In order to present this as a program transformation, we introduce the following specification as a definition:

$$\mathit{seq}_3(t) = \mathit{seq}_2(t, [(\varepsilon, \mathit{strat}(t))])$$

Now we calculate:

$$\begin{aligned} \mathit{seq}_3(t) &= \mathit{seq}_2(t, [(\varepsilon, \mathit{strat}(t))]) \\ &= \begin{cases} \textbf{if } \mathit{rewr_2}(t, [(\varepsilon, \mathit{strat}(t))]) = (q, s, R) \text{ for some } q,\ s,\ R \\ \quad \textbf{then } t :: \mathit{seq}_2(t[s]_q, R) \\ \quad \textbf{else } \langle t \rangle \end{cases} \\ &= \ \{\text{By Lemma 10}\} \\ &\quad \begin{cases} \textbf{if } \mathit{rewr_2}(t, [(\varepsilon, \mathit{strat}(t))]) = (q, s, R) \text{ for some } q,\ s,\ R \\ \quad \textbf{then } t :: \mathit{seq}_2(t[s]_q, [(\varepsilon, \mathit{strat}(t[s]_q))]) \\ \quad \textbf{else } \langle t \rangle \end{cases} \\ &= \begin{cases} \textbf{if } \mathit{rewr_2}(t, [(\varepsilon, \mathit{strat}(t))]) = (q, s, R) \text{ for some } q,\ s,\ R \\ \quad \textbf{then } t :: \mathit{seq}_3(t[s]_q) \\ \quad \textbf{else } \langle t \rangle \end{cases} \\ &= \ \{\text{By Lemma 11}\} \\ &\quad \begin{cases} \textbf{if } \mathit{rewr_1}(t) = (q, s) \text{ for some } q,\ s \\ \quad \textbf{then } t :: \mathit{seq}_3(t[s]_q) \\ \quad \textbf{else } \langle t \rangle \end{cases} \end{aligned}$$

This is exactly the defining equation of seq_1.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, March 2001.
3. M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software – Practice and Experience*, 30(3):259–291, 2000.
4. M.G.J. van den Brand, P. Klint, and P.A. Olivier. Compilation and memory management for ASF+SDF. In S. Jähnichen, editor, *8th Int. Conf. on Compiler construction CC'99*, volume 1575 of *LNCS*, pages 198–213. Springer, 1999.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. The Maude system. In P. Narendran and M. Rusinowitch, editors, *Proc. of the 10th Int. Conf. on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *LNCS*, pages 240–243. Springer, 1999.
6. W. Fokkink, J. Kamperman, and Walters. P. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Trans. on Programming Languages and Systems*, 20(3):679–706, 1998.
7. W. Fokkink, J. Kamperman, and Walters. P. Lazy rewriting on eager machinery. *ACM Trans. on Programming Languages and Systems*, 22(1):45–86, January 2000.
8. W. Fokkink and J.C. van de Pol. Simulation as a correct transformation of rewrite systems. In I. Prívara and P. Ružička, editors, *22th Int. Symp. on Mathematical Foundations of Computer Science (MFCS'97)*, LNCS 1295, pages 249–258, Bratislava, 1997. Springer.
9. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action.* Kluwer, 2000.
10. J.F. Groote and J.C. van de Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Reasoning, LPAR2000*, volume 1955 of *LNAI*, pages 161–178. Springer, 2000.
11. J.W. Klop. Term rewriting systems. In D. Gabbay S. Abramski and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
12. S.P. Luttik and E. Visser. Specification of rewriting strategies. In M.P.A. Sellink, editor, *Second International Conference on the Theory and Practice of Algebraic Specification (ASF+SDF'97)*, Electronic Workshops in Computing. Springer, 1997.
13. T. Nagaya. *Reduction Strategies for Term Rewriting Systems.* PhD thesis, JAIST, Japan, 1999.
14. A.T. Nakagawa, T. Sawada, and K. Futatsugi. CafeOBJ user's manual, version 1.3. Available via `http://www.ipa.go.jp/STC/CafeP/index-e.html`.
15. M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *The 3rd Int. W. on Rewriting Logic and its Applications (WRLA2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 2001.
16. K. Ogata and K. Futatsugi. Implementation of term rewritings with the evaluation strategy. In H. Glaser, P. Hartel, and H. Kuchen, editors, *9th Int. Symp. on Programming Languages: Implmentations, Logics, and Programs, PLILP97*, volume 1292 of *LNCS*, pages 225–239, 1997.
17. E. Visser. The Stratego tutorial and reference manual. Available via `http://www.stratego-language.org/`.