Lazy Rewriting on Eager Machinery

WAN FOKKINK
University of Wales Swansea
JASPER KAMPERMAN
Reasoning, Inc.
and
PUM WALTERS
Babelfish

The article introduces a novel notion of lazy rewriting. By annotating argument positions as lazy, redundant rewrite steps are avoided, and the termination behavior of a term-rewriting system can be improved. Some transformations of rewrite rules enable an implementation using the same primitives as an implementation of eager rewriting.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Languages

Additional Key Words and Phrases: Innermost reduction, lazy rewriting, specificity ordering

1. INTRODUCTION

A term-rewriting system (TRS) provides a mechanism to reduce terms to their normal forms, which cannot be reduced any further. A TRS consists of a number of rewrite rules $\ell \to r$, where ℓ and r are terms. For any term t, such a rule allows one to replace subterms $\sigma(\ell)$ of t by $\sigma(r)$, for substitutions σ. The structure $\sigma(\ell)$ is called a redex (reducible expression) of t. A term can contain several redexes, so that the question arises: which of these redexes should actually be reduced. There are two standard strategies:

—*outermost* rewriting reduces a redex as close as possible to the root of the parse tree of the term;

—*innermost* (or *eager*) rewriting reduces a redex as close as possible to the leaves of the parse tree of the term.

Authors' addresses: W. J. Fokkink, University of Wales Swansea, Department of Computer Science, Singleton Park, Swansea SA2 8PP, Wales, UK; email: W.J.Fokkink@swan.ac.uk; J. F. Th. Kamperman, Reasoning, Inc., 700 East El Camino Real, Mountain View, CA 94040; email: jasper.kamperman@reasoning.com; H. R. Walters, Babelfish, Vondellaan 14, 1217 RX Hilversum, The Netherlands; email: pum@babelfish.nl.

On the one hand, outermost rewriting often displays better termination behavior (e.g., see O'Donnell [1977]), meaning that it can give rise to fewer infinite reductions. Namely, an innermost redex that gives rise to an infinite reduction may be eliminated by the contraction of an outermost redex. Moreover, outermost rewriting can produce a sequence of partial results (prefixes of the normal form) that is useful even when this strategy does not terminate. This is interesting if the output of one program is connected to the input of another; it may be that the nonterminating subcomputations in the first program are not necessary for the termination of the second program. On the other hand, the implementation of innermost rewriting usually involves less run-time overhead, mainly for two reasons. First, it does not require the expensive building of redexes, as their contexts can be inspected after contraction of the redex. Second, after the application of a rewrite rule, the subterms in the reduct that are innermost with respect to the pattern at the right-hand side of the rewrite rule are known to be irreducible, which avoids repeated inspection of such structures.

The aim of *lazy evaluation* [Friedman and Wise 1976; Henderson and Morris 1976], which underlies lazy functional languages (e.g., see Plasmeijer and van Eekelen [1993]) and equational programming [Hoffmann and O'Donnell 1982b], is to combine an efficient implementation with good termination properties. During the lazy evaluation of a term it is decided whether the input term is in weak head normal form, meaning that it is not a redex. If this decision procedure requires the reduction of a proper subterm, then ideally it selects a needed redex [Huet and Lévy 1991], for which it is certain that no rewriting strategy can eliminate it. Needed redexes can be determined (efficiently) for orthogonal TRSes that are (strongly) sequential; see Huet and Lévy [1991] and O'Donnell [1985]. If a term is decided to be in weak head normal form, then its outermost function symbol is presented as output, and its arguments are subsequently reduced to weak head normal form.

Hartel et al. [1996, p. 649] conclude that "non-strict compilers do not achieve ... the performance of eager implementations" and that "interpreters for strict languages (Caml Light, Epic) do seem on the whole to be faster than interpreters for non-strict languages (NHC, Gofer, RUFLI, Miranda)." A bottleneck for the performance of lazy evaluation, compared to eager evaluation, is that it requires a considerable amount of bookkeeping. In order to reduce this kind of overhead, in practice lazy evaluation is often adapted to do a lot of eager evaluation. Strandh [1988; 1989] showed for the restricted class of forward-branching equational programs that lazy evaluation can be performed efficiently on eager hardware.

Arguments that can be rewritten eagerly without affecting termination behavior are called strict. Strictness analysis, initiated by Mycroft [1980], attempts to identify these arguments statically. In lazy functional languages, strictness analysis and programmer-provided strictness annotations are used to do as much eager evaluation as possible. For example, Clean [Brus et al. 1987] supports the annotation of strict arguments, which are evaluated in an eager fashion. In Clean, experience shows how in a lazy program most arguments can be annotated as strict without influencing the termination behavior of the program [Plasmeijer 1998].

We investigate how an eager implementation can be adapted to do some lazy evaluation. We propose the notion of a laziness annotation, which assigns to each argument of a function symbol either the label "eager" or the label "lazy." Only

redexes that are not the subterm of a lazy argument (the so-called "active" redexes) are reduced whenever possible, according to the innermost rewriting strategy. This notion of lazy rewriting avoids rewriting at lazy arguments to a large extent. Laziness annotations can be provided by a strictness analyzer, in which case all arguments that are not found to be strict get the annotation lazy, or by a programmer. In practice, strictness analyzers can be quite conservative, so that they may indicate far too many lazy arguments; e.g., see Sekar et al. [1997].

A standard application of lazy evaluation, where compared to eager evaluation it improves both termination behavior and performance, is the if-then-else construct:

$$\begin{aligned} \mathit{if}(\mathtt{true}, x, y) &\to x \\ \mathit{if}(\mathtt{false}, x, y) &\to y. \end{aligned}$$

A sensible laziness annotation makes the first argument of *if* eager and the last two arguments lazy. Then, evaluation of a term $\mathit{if}(s, t_0, t_1)$ first reduces s to either `true` or `false`; next the term is rewritten to t_0 or t_1, respectively, and finally this reduct is evaluated. Thus, laziness avoids that both t_0 and t_1 are reduced, because one of these evaluations would be irrelevant, depending on the evaluation of s.

We give an example, which also appears in Ogata and Futatsugi [1997], to explain the use of laziness annotations in more detail. The following TRS returns an element from an infinite list of natural numbers. Note that the left-hand sides of the second and third rule are overlapping. Following ideas from Baeten et al. [1989] and Kennaway [1990], the third rule is given priority over the second rule, because its left-hand side is more specific.

$$\begin{array}{lrcl} (1) & \mathit{inf}(x) & \to & \mathit{cons}(x, \mathit{inf}(\mathit{succ}(x))) \\ (2) & \mathit{nth}(x, \mathit{cons}(y, z)) & \to & y \\ (3) & \mathit{nth}(\mathit{succ}(x), \mathit{cons}(y, z)) & \to & \mathit{nth}(x, z) \end{array}$$

For $k \geq 0$, outermost rewriting reduces the term $\mathit{nth}(\mathit{succ}^k(0), \mathit{inf}(0))$ to its normal form $\mathit{succ}^k(0)$, by k alternating applications of rules (1) and (3), followed by a final application of rules (1) and (2). However, the innermost rewriting strategy does not detect this reduction, but produces an infinite reduction instead, using only rule (1). For instance, in the case $k = 0$, eager evaluation yields

$$\begin{aligned} \mathit{nth}(0, \mathit{inf}(0)) &\xrightarrow{(1)} \mathit{nth}(0, \mathit{cons}(0, \mathit{inf}(\mathit{succ}(0)))) \\ &\xrightarrow{(1)} \mathit{nth}(0, \mathit{cons}(0, \mathit{cons}(\mathit{succ}(0), \mathit{inf}(\mathit{succ}^2(0))))) \\ &\xrightarrow{(1)} \ldots \end{aligned}$$

In order to avoid such infinite reductions, we can define the second argument of *cons* to be lazy, while the arguments of the function symbols *inf*, *nth*, and *succ*, and the first argument of *cons* are defined to be eager. Lazy rewriting with respect to this laziness annotation provides a finite reduction for the term $\mathit{nth}(0, \mathit{inf}(0))$. Namely, after the first reduction step

$$\mathit{nth}(0, \mathit{inf}(0)) \xrightarrow{(1)} \mathit{nth}(0, \mathit{cons}(0, \mathit{inf}(\mathit{succ}(0))))$$

it is not allowed to reduce the lazy second argument $inf(succ(0))$ of $cons$. So the only reduction that remains is

$$nth(0, cons(0, inf(succ(0)))) \stackrel{(2)}{\rightarrow} 0$$

which produces the normal form 0.

Lazy rewriting extends the class of weak head normal forms, owing to the fact that we do not require the lazy arguments of a normal form to be in normal form. For instance, in the example above, the laziness annotation even produces a normal form for the term $inf(0)$, which does not have a finite reduction at all. Namely, after the first rewrite step

$$inf(0) \stackrel{(1)}{\rightarrow} cons(0, inf(succ(0)))$$

it is not allowed to reduce the lazy argument $inf(succ(0))$ of $cons$. So the term $cons(0, inf(succ(0)))$ is a *lazy* normal form, i.e., a normal form with respect to the laziness annotation.

Laziness annotations can be implemented by irreducible encodings called thunks [Ingerman 1961], which make it possible to forget about laziness, and focus on eager evaluation only. That is, before a term is rewritten, first its lazy arguments $f(t_1, \ldots, t_n)$ are thunked as follows. Such a subterm is transformed into $\Theta(\tau_f, t_1, \ldots, t_n)$, where the special function symbol Θ represents a thunk, and the special constant τ_f is a token related to the function symbol f. Moreover, the arguments $t_1, \ldots, t_n$ are thunked following the same procedure. This conversion turns the lazy argument $f(t_1, \ldots, t_n)$ into a normal form, while its original term structure is stored by means of the tokens τ_g for function symbols g that occur in $f(t_1, \ldots, t_n)$. Extra rewrite rules are added to restore the original term structure, if this thunk is placed in an eager argument. Furthermore, lazy arguments in right-hand sides of rewrite rules are transformed as follows. If a rewrite rule $\ell \rightarrow r$ contains a lazy argument t in its right-hand side r, with variables $x_1, \ldots, x_n$, then we replace t in r by $\Theta(\lambda, x_1, \ldots, x_n)$. The λ is a special constant that registers the term structure of t. Rewrite rules are added to transform this structure into t, if it is placed in an eager argument. An additional advantage of this last procedure is that the lazy subterm t is compressed to a structure whose size is determined by its number of variables. We only have to spend the time and space needed to build t if it is made eager. If, however, the lazy argument t is eliminated, then this construction is avoided. Strandh [1987] presented a similar compression technique in equational programming.

Lazy rewriting of terms would hamper the efficiency of its implementation, because this could lead to duplication of arguments that are not in normal form. For example, suppose a rule

$$f(x) \rightarrow g(x, x)$$

is applied to a term $f(t)$, in the setting of eager evaluation with a laziness annotation. Then it may be the case that the term t is not a normal form, due to the fact that it contains subterms that are thunks. These thunks are duplicated in the reduct $g(t, t)$. In order to solve this inefficiency, lazy evaluation is usually performed by *graph* rewriting [Staples 1980] instead of term rewriting. In graph rewriting, the two arguments of $g(x, x)$ point to the same node in the graph, so that in $g(t, t)$

the thunks in t are not duplicated. A term can be considered as a graph, and the graph reducts of this graph, with respect to some TRS, can be expanded into terms again. These terms can also be obtained from the original term by term rewriting with respect to the same TRS; see Barendregt et al. [1987]. Hence, each parallel reduction in graph rewriting simulates a reduction in term rewriting. As a matter of fact, even without laziness annotation, implementation of eager evaluation of terms usually boils down to graph rewriting. Namely, storing a term such as $g(t,t)$ as a graph, in which the two arguments t point to the same node, saves memory space. See O'Donnell [1985] for an overview of the pragmatics of implementing graph rewriting in the setting of equational programming.

It is customary in graph rewriting to restrict to left-linear TRSes, in which left-hand sides of rewrite rules do not contain multiple occurrences of the same variable. TRSes that are not left-linear require checks on syntactic equality, which have a complexity that is related to the sizes of the graphs to be checked. In innermost rewriting, each TRS can be simulated efficiently by a left-linear TRS, using an equality function; e.g., see Kamperman [1996, p. 28]. Most functional languages feature many-sortedness, conditional rewrite rules, and a higher-order syntax. A TRS over a many-sorted signature can be treated as a single-sorted TRS after it has been type-checked, so we take the liberty to only consider single-sorted TRSes. A sensible way to compile conditional TRSes is to eliminate conditions first. In innermost rewriting, conditions of the form $s \downarrow t$ (i.e., s and t have the same normal form) can be expressed by means of an equality function. For example, a rewrite rule $x \downarrow y \Rightarrow f(x,y) \to r$ is simulated by the TRS

$$\begin{aligned} f(x,y) &\to g(eq(x,y),x,y) \\ g(\mathtt{true},x,y) &\to r \\ g(\mathtt{false},x,y) &\to f'(x,y). \end{aligned}$$

Alternatively, conditions can be evaluated *after* pattern matching, by an extension of the pattern-matching automaton. Since the complications related to the implementation of conditions are orthogonal to the matters investigated in this article, we only consider unconditional TRSes. Finally, we focus on first-order terms. The notion of a laziness annotation extends to a higher-order syntax without complications, giving rise to our notion of lazy rewriting. The implementation of this generalization, however, requires further study.

Left-hand sides of rewrite rules are allowed to be overlapping. In most functional languages such ambiguities are resolved by means of a textual ordering, which makes the priority of a rewrite rule dependent on its position in the layout of the TRS. Kennaway [1990] argued that textual ordering has an unclear semantics, and advocated the use of specificity ordering, in which a rewrite rule has higher priority if its left-hand side has more syntactic structure. Specificity ordering makes sense, because reversal of this priority would mean that only rewrite rules with minimal syntactic structure in their left-hand sides would ever be applied. Kennaway showed how to transform a TRS with specificity ordering into an orthogonal, strongly sequential TRS. Specificity ordering from Kennaway does not resolve ambiguities between overlapping rules such as $f(a,x) \to r$ and $f(y,b) \to r'$. In Kamperman and Walters [1996] and Fokkink et al. [1998], this ordering was refined by comparing the syntactic structure of arguments of left-hand sides of rewrite rules from left to

right. For example, $f(a, x) \to r$ has higher priority than $f(y, b) \to r'$, because the first argument a of the left-hand side $f(a, x)$ has more syntactic structure than the first argument y of the left-hand side $f(y, b)$.

In this article the specificity ordering from Kamperman and Walters [1996] and Fokkink et al. [1998] is adapted in such a way that it takes into account the laziness annotation. Left-hand sides of rewrite rules are minimized, so that they agree with the success-transitions of the corresponding pattern-matching automaton of Hoffmann and O'Donnell [1982a]. Minimization of left-hand sides mimics scanning left-hand sides with respect to their specificity. As minimization breaks up this scan into small steps, it is advantageous for the presentation and for the correctness proofs. Minimization has the additional benefit that it leads to implementations that are better structured and thus more easy to maintain. In comparison with a straightforward scan, minimization incurs a small cost at compile-time, but not at run-time. Although the number of rewrite steps increases in a linear fashion, the complexity of executing a single rewrite step decreases, which in practice leads to comparable performance (cf., Fokkink et al. [1998]).

We explained that lazy nodes in graphs are thunked, and that the original rewrite rules are adapted to introduce thunks in right-hand sides and to minimize left-hand sides. The transformations involve an extension of the original alphabet and an adaptation of the class of normal forms. We prove that eager evaluation with respect to the transformed TRS simulates lazy rewriting with respect to the original TRS, using a simulation notion from Fokkink and van de Pol [1997]. This simulation is shown to be sound, complete, and termination preserving, which implies that no information on normal forms in the original TRS is lost.

This article is set up as follows. Sections 2 and 3 present the notions of standard and lazy graph rewriting, respectively. Section 4 explains how lazy rewriting can be converted to eager rewriting by the introduction of thunks. Section 5 contains a comparison with related work. Appendix A presents the correctness proof. Earlier versions of this article appeared as Kamperman and Walters [1995] and Kamperman [1996, Chapt. 6].

2. STANDARD GRAPH REWRITING

In this section we present the preliminaries, including a formal definition of graphs. Furthermore, we define the notion of standard graph rewriting, following for example Staples [1980] and Barendregt et al. [1987].

2.1 Term-Rewriting Systems

Definition 2.1.1. A *signature* Σ consists of

—a countably infinite set $\mathcal{V}$ of variables $x, y, z, \ldots$;

—a nonempty set $\mathcal{F}$ of function symbols $f, g, h, \ldots$, disjoint from $\mathcal{V}$, where each function symbol f is provided with an arity $ar(f)$, being a natural number.

Each $i \in \{1, \ldots, ar(f)\}$ is called an *argument* of f. Function symbols of arity 0 are called *constants*.

Definition 2.1.2. Assume a signature $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$. The set of *(open) terms* $\ell, r, s, t, \ldots$ over Σ is the smallest set satisfying

—all variables are terms;

—if $f \in \mathcal{F}$ and $t_1, \ldots, t_{ar(f)}$ are terms, then $f(t_1, \ldots, t_{ar(f)})$ is a term.

t_i is called the ith *argument* of $f(t_1, \ldots, t_{ar(f)})$. Syntactic equality between terms is denoted by $=$.

Definition 2.1.3. A *rewrite rule* is an expression $\ell \to r$ with ℓ and r terms, where all variables that occur in the right-hand side r also occur in the left-hand side ℓ.

A *term-rewriting system* (TRS) $\mathcal{R}$ consists of a finite set of rewrite rules.

Definition 2.1.4. A term is *linear* if it does not contain multiple occurrences of the same variable.

A rewrite rule is *left-linear* if its left-hand side is linear. A TRS is *left-linear* if all its rules are.

2.2 Term Graphs

A term graph is an acyclic rooted graph, in which each node has as label a function symbol or variable, and in which its number of children is compatible with the arity of its label.

Definition 2.2.1. A *term graph* g consists of

—a finite collection $\mathcal{N}$ of nodes;

—a root node $\nu_0 \in \mathcal{N}$;

—a labeling mapping $\mathit{label}_{\mathsf{g}} : \mathcal{N} \to \mathcal{F} \cup \mathcal{V}$;

—a children mapping $\mathit{childr}_{\mathsf{g}} : \mathcal{N} \to \mathcal{N}^*$, where $\mathcal{N}^*$ denotes the collection of (possibly empty) strings over $\mathcal{N}$.

Furthermore, the term graph g must be acyclic; that is, there exists a well-founded ordering $<$ on $\mathcal{N}$ such that if $\nu' \in \mathit{childr}_{\mathsf{g}}(\nu)$ then $\nu' < \nu$. Finally, if $\mathit{label}_{\mathsf{g}}(\nu) = f$, then $\mathit{childr}_{\mathsf{g}}(\nu) = \nu_1 \ldots \nu_{ar(f)}$, where intuitively ν_i represents the ith argument of f. If $\mathit{label}_{\mathsf{g}}(\nu) \in \mathcal{V}$, then $\mathit{childr}_{\mathsf{g}}(\nu)$ is the empty string.

Each term t can be adapted to a term graph $G(t)$.

—If $t = x$, then $G(t)$ consists of a single root node, with label x and no children.

—If $t = f(t_1, \ldots, t_{ar(f)})$, then the root node of $G(t)$ carries the label f, and its ith child is the root node of $G(t_i)$ for $i = 1, \ldots, ar(f)$.

Reversely, each term graph g can be unraveled to obtain a term $U(\mathsf{g})$. Let ν_0 denote the root node of g.

—If $\mathit{label}_{\mathsf{g}}(\nu_0) = x$, then $U(\mathsf{g}) = x$.

—If $\mathit{label}_{\mathsf{g}}(\nu_0) = f$ and $\mathit{childr}_{\mathsf{g}}(\nu_0) = \nu_1 \ldots \nu_{ar(f)}$, then we construct the terms $U(\mathsf{g}_1), \ldots, U(\mathsf{g}_{ar(f)})$, where g_i differs from g only in the fact that it has root node ν_i. We define $U(\mathsf{g}) = f(U(\mathsf{g}_1), \ldots, U(\mathsf{g}_{ar(f)}))$.

In the sequel, term graph is abbreviated to graph.

2.3 Graph Rewriting

We present the notion of standard graph rewriting, induced by a left-linear TRS. First, we give an inductive definition for matching a linear term with a pattern in a graph.

Definition 2.3.1. A linear term ℓ *matches* node ν in graph g if

(1) either ℓ is a variable, in which case ℓ is *linked* with ν;
(2) or $\ell = f(t_1, \ldots, t_{ar(f)})$, and
—$\mathit{label}_{\mathsf{g}}(\nu) = f$ and $\mathit{childr}_{\mathsf{g}}(\nu) = \nu_1 \ldots \nu_{ar(f)}$;
—t_i matches ν_i in g, for $i = 1, \ldots, ar(f)$.

Note that each variable in ℓ is linked with only one node in g, owing to the fact that ℓ is linear.

If the left-hand side of a left-linear rewrite rule $\ell \to r$ matches a node ν in a graph g, then this may give rise to a rewrite step, in which the pattern ℓ at node ν in g is replaced by the pattern $G(r)$. In principle, this means that node ν in g is replaced by the root node of $G(r)$, and each node in $G(r)$ that has as label a variable x is replaced by the node in g linked with x. The resulting two graphs $\overline{\mathsf{g}}$ and $\overline{G(r)}$ (with dangling links) are combined to obtain the reduct g' of g. If ν is the root node of g, then some care is needed in selecting a new root node for g'.

Definition 2.3.2. The standard rewrite relation $\to$ on graphs, for a left-linear TRS $\mathcal{R}$, is defined as follows. Assume that $\ell \to r \in \mathcal{R}$, and assume that ℓ matches node ν in graph g. Each variable x in r is linked with a unique node ν_x in g, owing to the fact that ℓ is linear. Then $\mathsf{g} \to \mathsf{g}'$, where g' is constructed as follows.

Construct $G(r)$, such that its nodes do not yet occur in g. Adapt $G(r)$ to $\overline{G(r)}$ by renaming, in the image of $\mathit{childr}_{G(r)}$, all occurrences of nodes that have as label a variable x into ν_x. Adapt g to $\overline{\mathsf{g}}$ by renaming, in the image of $\mathit{childr}_{\mathsf{g}}$, all occurrences of the node ν into the root node of $\overline{G(r)}$.

—**If** ν is the root node of g, and r is a single variable x
then $\mathsf{g}' = \mathsf{g}$ with as root node ν_x.
—**If** ν is the root node of g and r is not a single variable
then $\mathsf{g}' = \overline{\mathsf{g}} \cup \overline{G(r)}$ with as root node the root node of $\overline{G(r)}$.
—**If** ν is not the root node of g
then $\mathsf{g}' = \overline{\mathsf{g}} \cup \overline{G(r)}$ with as root node the root node of $\overline{\mathsf{g}}$.

It is not hard to see that g' meets the requirements of Definition 2.2.1, so it is a graph.

Remark. For an efficient implementation of graph rewriting it is important to avoid waste of memory space, by means of so-called garbage collection. That is, nodes in a graph that are no longer connected to the root node should be reclaimed. We refrain from a description of garbage collection, because it is not of importance for the rewrite relation as such. The reader is referred to Wilson [1992] for an overview of garbage collection techniques.

Definition 2.3.3. A graph g is called a *normal form* for a left-linear TRS $\mathcal{R}$ if $\mathcal{R}$ does not induce any rewrite step $\mathsf{g} \to \mathsf{g}'$.

Henceforth we focus on graph rewriting with the implicit aim to implement *term* rewriting (e.g., see Baader and Nipkow [1998]). Each parallel reduction in graph rewriting simulates a reduction in term rewriting; see Barendregt et al. [1987] and Kennaway et al. [1994]. That is, consider the left-linear TRS $\mathcal{R}$ and a term t. If graph rewriting with respect to $\mathcal{R}$ reduces the graph $G(t)$ to the graph g, then term rewriting with respect to $\mathcal{R}$ reduces the term t to the term $U(\mathsf{g})$. Moreover, if g is a normal form for graph rewriting with respect to $\mathcal{R}$, then $U(\mathsf{g})$ is a normal form for term rewriting with respect to $\mathcal{R}$.

3. LAZY GRAPH REWRITING

In this section we introduce the notions of a lazy signature, lazy graphs, lazy graph rewriting, and lazy normal forms.

3.1 Lazy Graphs

Definition 3.1.1. In a *lazy* signature $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$, each argument of each function symbol in $\mathcal{F}$ is either eager or lazy. The laziness predicate Λ on $\mathcal{F} \times \mathbb{N}$ holds for (f, i) if and only if $1 \leq i \leq ar(f)$ and the ith argument of f is lazy.

Definition 3.1.2. In a *lazy* graph g, each node is either eager or lazy, where the root node is always eager.

Each term t over a lazy signature can be adapted to a lazy graph $G(t)$ as before (see Section 2.2), with one extra clause if $t = f(t_1, \ldots, t_{ar(f)})$:

—the root node ν_0 of $G(t)$ is eager; each node ν_i for $i = 1, \ldots, ar(f)$ is lazy if and only if $\Lambda(f, i)$; all other nodes in $G(t)$ are lazy/eager if they are so in the lazy graphs $G(t_1), \ldots, G(t_{ar(f)})$.

An eager node in a lazy graph is called *active* if it can be reached from the root node via a path of eager nodes. Our notion of lazy graph rewriting only reduces patterns at active nodes.

Definition 3.1.3. The collection of *active* nodes in a lazy graph g is defined inductively as follows:

—the root node of g is active;
—if ν is active, then the eager nodes in $childr_{\mathsf{g}}(\nu)$ are active.

A subterm in a term t is said to be active if it corresponds to an active node in $G(t)$.

3.2 Lazy Graph Rewriting

We allow the reduction of redexes at active nodes, and at lazy nodes that are essential in the sense that their contraction may lead to new redexes at active nodes. The reduction of redexes at essential lazy nodes is suspended as much as possible.

We introduce a variation of the notion of matching as formulated in Definition 2.3.1. Intuitively, a left-hand side of a rewrite rule matches *modulo laziness* an active node ν in a lazy graph if it matches up to lazy nodes. These lazy nodes are called essential, to indicate that making them eager and reducing them to normal form could mean the development of a new redex at ν.

Definition 3.2.1. A linear term ℓ *matches modulo laziness* an active node ν in a lazy graph g if

(1) either ℓ is a variable, in which case ℓ is *linked* with ν;
(2) or $\ell = f(t_1, \ldots, t_{ar(f)})$, and
—$label_{\mathsf{g}}(\nu) = f$ and $childr_{\mathsf{g}}(\nu) = \nu_1 \ldots \nu_{ar(f)}$;
—if ν_i is eager, then t_i matches modulo laziness node ν_i in g, for $i = 1, \ldots, ar(f)$.
If ν_i is lazy and t_i is not a variable, then ν_i is called *essential*, for $i = 1, \ldots, ar(f)$.

We give an example of matching modulo laziness.

Example 3.2.2. Assume a unary function symbol f and constants a and b. Let the lazy graph g consist of two nodes ν and ν', where the eager root ν has label f and its child ν' has label a.

—If ν' is lazy, then the term $f(b)$ matches modulo laziness the root node ν of g, owing to the fact that ν has label f. In this case, ν' is essential in g.
—If ν' is eager, then the term $f(b)$ does not match modulo laziness the root node ν of g, due to the fact that its subterm b does not match modulo laziness the active node ν' with label a in g.

We define the lazy graph rewrite relation $\rightarrow_L$ as induced by a left-linear TRS. If the left-hand side of a left-linear rewrite rule $\ell \rightarrow r$ matches modulo laziness an active node ν in a lazy graph g, then $\mathsf{g} \rightarrow_L \mathsf{g}'$ where g' is constructed depending on two cases.

(1) *The pattern match does not give rise to essential lazy nodes.*
Then g' can be constructed as in the standard rewrite relation in Definition 2.3.2, where the pattern ℓ in g rooted at ν is replaced by the pattern $G(r)$. That is, node ν in g is replaced by the root node of $G(r)$, and each node in $G(r)$ with as label a variable x is replaced by the node in g linked with x. If a lazy node in g is linked with a variable in ℓ, then this node is made eager in g' if and only if the variable occurs as an eager argument in r. The laziness annotation of other nodes in g' is inherited from g and $G(r)$. If ν is the root node of g, then some care is needed in selecting a new root node for g'.

(2) *The pattern match gives rise to essential lazy nodes.*
Then we may attempt to develop the pattern ℓ at node ν, by reducing the subgraph of g that is rooted at such an essential lazy node to normal form. That is, g' can be obtained from g by making an essential lazy node in g eager, so that the subgraph that is rooted at this node can be normalized.

Definition 3.2.3. The lazy rewrite relation $\rightarrow_L$ on lazy graphs, for a left-linear TRS $\mathcal{R}$, is defined as follows. Assume that $\ell \rightarrow r \in \mathcal{R}$, and assume that ℓ matches

modulo laziness an active node ν in lazy graph g. Each variable x in r is linked with a unique node ν_x in g, owing to the fact that ℓ is linear. We distinguish two cases.

(1) Suppose ℓ matching modulo laziness node ν in g does not give rise to essential lazy nodes. Then $\mathsf{g} \to_L \mathsf{g}'$, where g' is constructed as follows.
Construct $G(r)$, such that its nodes do not yet occur in g. Adapt $G(r)$ to $\overline{G(r)}$ by renaming, in the image of $childr_{G(r)}$, all occurrences of nodes with as label a variable y into ν_y. Adapt g to $\overline{\mathsf{g}}$ by renaming, in the image of $childr_{\mathsf{g}}$, all occurrences of the node ν into the root node of $\overline{G(r)}$.
—**If** ν is the root node of g, and r is a single variable x
then $\mathsf{g}' = \mathsf{g}$ with as root node ν_x.
ν_x is eager, while the laziness annotation of all other nodes in g' is inherited from g.
—**If** ν is the root node of g and r is not a single variable
then $\mathsf{g}' = \overline{\mathsf{g}} \cup \overline{G(r)}$ with as root node the root node of $\overline{G(r)}$.
For variables y in r, the node ν_y is lazy in g' if and only if ν_y is lazy in g and all nodes in $G(r)$ with label y are lazy; the laziness annotation of all other nodes in g' is inherited from g and $G(r)$.
—**If** ν is not the root node of g
then $\mathsf{g}' = \overline{\mathsf{g}} \cup \overline{G(r)}$ with as root node the root node of $\overline{\mathsf{g}}$.
For variables y in r, the node ν_y is lazy in g' if and only if ν_y is lazy in g and all nodes in $G(r)$ with label y are lazy; the laziness annotation of all other nodes in g' is inherited from g and $G(r)$.

(2) Suppose ℓ matching modulo laziness node ν in g does give rise to essential lazy nodes. Let ν' be such an essential lazy node. Then $\mathsf{g} \to_L \mathsf{g}'$, where the only distinction between g and g' is that ν' is lazy in g and eager in g'.

We give an example of a reduction of a lazy graph by means of the lazy rewrite relation induced by a left-linear TRS.

Example 3.2.4. Assume a unary function symbol f, constants a, b, and c, and a TRS $\{a \to b, f(b) \to c\}$. Let the lazy graph g consist of two nodes ν and ν', where the eager root ν has label f and its lazy child ν' has label a.

ν has label f, and its child ν' is lazy; so the left-hand side $f(b)$ matches modulo laziness the root node ν of g. Since the subterm b of this left-hand side is nonvariable, ν' is essential in g. So $\mathsf{g} \to_L \mathsf{g}'$, where g' is obtained from g by making ν' eager.

ν' has label a and is active in g'. So an application of the rule $a \to b$ with respect to ν' in g' yields $\mathsf{g}' \to_L \mathsf{g}''$, where g'' is obtained from g' by renaming the label of ν' into b.

The root node ν has label f, and its eager child ν' has label b in g''. So an application of the rule $f(b) \to c$ with respect to ν in g'' yields $\mathsf{g}'' \to_L \mathsf{g}'''$, where g''' consists only of a root node with the label c. The lazy graph g''' is in lazy normal form.

3.3 Sharing Patterns in Right-Hand Sides

Assume a left-linear TRS $\mathcal{R}$, and suppose a rule $\ell \to r$ in $\mathcal{R}$ contains multiple occurrences of a nonvariable term t in its right-hand side r. A naive application of this rewrite rule to a graph g would build a separate subgraph in the reduct of g for each occurrence of t in r, after which each of these subgraphs would be evaluated independently. In graph rewriting this waste of space and time can be avoided by the introduction of a *let* construct, which enables one to share the occurrences of t in r. We show how this *let* construct can be captured by a transformation of the rewrite rules.

Let $P(r)$ consist of the nonvariable terms that occur more than once in r. Select a $t \in P(r)$, such that $P(r)$ contains either no or more than one element of the form $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_{ar(f)})$. (Namely, if $P(r)$ contains exactly one such term, then it is more efficient to share the occurrences of $f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_{ar(f)})$ in r.) A $t \in P(r)$ that satisfies the criterion above can be determined efficiently in a bottom-up fashion. We fix a fresh variable y, and replace each occurrence of t in r by y, to obtain a term r'. Let $x_1, \ldots, x_n, y$ be the variables that occur in r'. We introduce a fresh function symbol h of arity $n+1$, with $\Lambda(h, n+1)$ if and only if t does not occur at an active position in r, and $\Lambda(h, i)$ for $i = 1, \ldots, n$ if and only if x_i is a lazy argument in ℓ and does not occur at an active position in r. We replace $\ell \to r$ by two rewrite rules:

$$\begin{aligned} \ell &\to h(x_1, \ldots, x_n, t) \\ h(x_1, \ldots, x_n, y) &\to r' \end{aligned}$$

These two new rewrite rules together simulate the original rewrite rule $\ell \to r$. The first rule builds the term t as an argument of h; this building process is suspended if t does not occur at an active position in r. In the second rule the term t is represented by the variable y; graph rewriting ensures that the multiple occurrences of y in r' are shared. We repeat this procedure until all multiple occurrences of nonvariable terms in right-hand sides of rewrite rules have been removed.

Strandh [1987] proposed a similar transformation in equational programming [O'Donnell 1985], to share patterns in right-hand sides of equations. Strandh's transformation moreover compresses the size of these patterns. In Section 4.5 we apply a similar strategy to compress the size of lazy arguments in right-hand sides of rewrite rules. Strandh proved that his transformed program simulates the original program, and concluded that the transformation preserves a useful confluence property.

Suppose we want to reduce a lazy graph g. The efficiency of lazy graph rewriting can be optimized by sharing nodes in g. It can be determined, in a bottom-up fashion, whether the subgraphs rooted at distinct nodes in g are the same. If so, then the two nodes can be compressed to a single node.

Similar to the transformation from term to graph rewriting, sharing subterms in right-hand sides and sharing nodes in lazy graphs give rise to a form of parallel rewriting: shared nodes are reduced simultaneously. Each parallel reduction with respect to the transformed TRS simulates a reduction with respect to the original TRS, where the reduction of shared nodes in the transformed TRS corresponds with the reduction of the separate nodes in the original TRS.

3.4 Lazy Normal Forms

Definition 3.4.1. A normal form for $\rightarrow_L$ is called a *lazy* normal form.

If we abstract away from laziness annotations of lazy graphs, then there are more lazy normal forms with respect to $\rightarrow_L$ than that there are normal forms with respect to the standard rewrite relation $\rightarrow$. The question arises whether all lazy normal forms are acceptable as such. Lazy rewriting takes the view that a subgraph of a lazy normal form only needs to be in normal form if it is rooted at an active node. Therefore, Proposition 3.4.3 shows that a lazy normal form only allows (possibly infinitely many) uninteresting rewrite steps. Hence, we conclude that lazy normal forms are acceptable indeed.

Definition 3.4.2. Let t be a term over a lazy signature, and assume a fresh constant ζ. The term t_ζ is obtained by replacing lazy arguments in t by ζ. We write $t \equiv_\zeta t'$ if $t_\zeta = t'_\zeta$.

Proposition 3.4.3. If g is a lazy normal form and $\mathsf{g} \rightarrow^* \mathsf{g}'$, then $U(\mathsf{g}) \equiv_\zeta U(\mathsf{g}')$.

Proof. Assume toward a contradiction that $U(\mathsf{g}) \not\equiv_\zeta U(\mathsf{g}')$. Then there is a rewrite rule that matches some active node in g. Since g is a lazy normal form, this redex in g must involve a lazy node, which according to Definition 3.2.1 (2) is essential. So according to Definition 3.2.3 (2) this lazy node in g can be rewritten, by means of $\rightarrow_L$, to an eager node. This contradicts the fact that g is a lazy normal form. □

4. TRANSFORMATIONS OF REWRITE RULES

In this section we explain how lazy rewriting can be implemented using the same primitives as an implementation of eager rewriting.

4.1 Specificity of Left-Hand Sides

We apply the rightmost innermost rewriting strategy. (The preference for rightmost is not of importance; we could also opt for leftmost innermost rewriting.) Let $dep_{\mathsf{g}}(\nu)$ denote the collection of nodes ν' in g for which there exists a path in g from ν to ν'.

$$\begin{array}{r} \nu \in dep_{\mathsf{g}}(\nu) \\ \nu' \in dep_{\mathsf{g}}(\nu) \wedge \nu'' \in childr_{\mathsf{g}}(\nu') \Rightarrow \nu'' \in dep_{\mathsf{g}}(\nu) \end{array}$$

The rightmost innermost ordering $<$ on nodes in a graph g is defined as follows:

—(Innermost) if $\nu' \in childr_{\mathsf{g}}(\nu)$ then $\nu < \nu'$;
—(Rightmost) if $childr_{\mathsf{g}}(\nu) = \nu_1 \ldots \nu_n$ and $1 \leq i < j \leq n$, then $\nu' < \nu''$ for all $\nu' \in dep_{\mathsf{g}}(\nu_i)$ and $\nu'' \in dep_{\mathsf{g}}(\nu_j)$.

We disambiguate overlapping left-hand sides of rewrite rules using a specificity relation $\prec$ from Fokkink et al. [1998]. The definition of $\prec$ below generalizes Fokkink et al. [1998, Def. 2.2.1] by taking into account the laziness annotation. If two left-hand sides are overlapping, then we give priority to the most evolved term structure, so typically $x \prec f(t_1, \ldots, t_{ar(f)})$. The specificity relation between two terms $f(s_1, \ldots, s_{ar(f)})$ and $f(t_1, \ldots, t_{ar(f)})$ is established by first comparing their

eager arguments from left to right, and then their lazy arguments from right to left. Let $=_\alpha$ denote syntactic equality modulo α-conversion, that is, modulo renaming of variables.

Definition 4.1.1. The syntactic specificity relation $\prec$ on terms is defined by

—$x \prec f(t_1, \ldots, t_{ar(f)})$;
—$f(s_1, \ldots, s_{ar(f)}) \prec f(t_1, \ldots, t_{ar(f)})$ if there is an eager argument i of f such that
  —$s_j =_\alpha t_j$ for eager arguments $j < i$ of f;
  —$s_i \prec t_i$.
—$f(s_1, \ldots, s_{ar(f)}) \prec f(t_1, \ldots, t_{ar(f)})$ if $s_j =_\alpha t_j$ for all eager arguments j of f, and there is a lazy argument i of f such that
  —$s_j =_\alpha t_j$ for lazy arguments $j > i$ of f;
  —$s_i \prec t_i$.

The specificity relation $\prec$ is extended to rewrite rules by

$$\ell \prec \ell' \;\Rightarrow\; \ell \to r \prec \ell' \to r'.$$

Note that a rewrite rule with a variable as left-hand side is less specific than any rewrite rule with a nonvariable left-hand side.

The specificity relation on terms is not transitive. For example, let the binary function symbol f have two eager arguments. Then $f(x, a) \prec f(a, x)$ and $f(a, x) \prec f(a, b)$, but $f(x, a) \not\prec f(a, b)$. However, if two linear terms s and t can be unified, then $s \prec t$, $t \prec s$, or $s =_\alpha t$. In order to ensure that each collection of overlapping left-linear rewrite rules contains a most specific rewrite rule, it suffices to require that left-hand sides of distinct rewrite rules are not equal modulo α-conversion.

We proceed to present the definition of rightmost innermost lazy rewriting with respect to the specificity relation, denoted by $\rightsquigarrow_L$.

Definition 4.1.2. Given a left-linear TRS $\mathcal{R}$, the binary rewrite relation $\rightsquigarrow_L$ is defined inductively as follows. Suppose the lazy graph g is not a lazy normal form for $\mathcal{R}$. Then there exist rewrite rules in $\mathcal{R}$ that match modulo laziness with active nodes in g.

—Select the rightmost innermost active node ν in g such that the left-hand side of some rewrite rule in $\mathcal{R}$ matches modulo laziness node ν in g.
—Select the greatest rule $\ell \to r$ in $\mathcal{R}$, with respect to the specificity relation, such that ℓ matches modulo laziness node ν in g.

We distinguish two possibilities.

(1) Suppose $\ell \to r$ matching modulo laziness node ν in g does not give rise to essential lazy nodes. Then, according to Definition 3.2.3 (1), it yields a rewrite step $\mathsf{g} \to_L \mathsf{g}'$ that replaces the pattern ℓ at node ν in g by $G(r)$. Define $\mathsf{g} \rightsquigarrow_L \mathsf{g}'$.
(2) Suppose $\ell \to r$ matching modulo laziness node ν in g does give rise to essential lazy nodes. Then, according to Definition 3.2.3 (2), it yields a rewrite step $\mathsf{g} \to_L \mathsf{g}'$ that makes the rightmost of these essential lazy nodes in g eager. Define $\mathsf{g} \rightsquigarrow_L \mathsf{g}'$.

$\rightsquigarrow_L$ is a subrelation of $\rightarrow_L$, and these two relations give rise to the same class of lazy normal forms.

4.2 Suppressing Laziness in Right-Hand Sides

For an efficient implementation of the lazy rewrite relation $\rightsquigarrow_L$ that is induced by some left-linear TRS $\mathcal{R}$, it is desirable to have only few lazy arguments in the right-hand sides of rewrite rules. We explain how to get rid of certain laziness annotations in right-hand sides of rewrite rules.

For each rewrite rule $\ell \rightarrow r$ in $\mathcal{R}$, nonvariable subterms of the right-hand side r that occur as proper subterms of the left-hand side ℓ, and variables in r that occur as an eager argument in ℓ, can be assigned the label eager in r without affecting termination behavior. Namely, such subterms of r are in lazy normal form with respect to $\rightsquigarrow_L$. We proceed to give a formal argumentation to support this claim.

Let ν be the rightmost innermost active node of a lazy graph g, such that the left-hand side of a most specific rewrite rule $\ell \rightarrow r$ in $\mathcal{R}$ matches modulo laziness node ν in g. Moreover, suppose this match does not give rise to essential lazy nodes in g. Then, according to Definition 4.1.2 (1), this match induces a reduction $\mathsf{g} \rightsquigarrow_L \mathsf{g}'$, where the pattern ℓ at node ν in g is replaced by $G(r)$. Suppose some subterm t of r occurs as a proper subterm of ℓ, with either $t \notin \mathcal{V}$ or t is an eager argument in ℓ.

ℓ matches modulo laziness node ν in g, and t is a subterm of ℓ; so t is linked with a descendant ν_0 of ν in g. Since t is a proper subterm of ℓ, $\nu < \nu_0$ with respect to the rightmost innermost ordering on nodes in g. Furthermore, since either t is a nonvariable subterm of ℓ or t is an eager argument in ℓ, and ℓ matches modulo laziness node ν in g without giving rise to essential lazy nodes, it follows then that ν_0 is active in g. Hence, the subgraph of g with root node ν_0 is a lazy normal form for $\mathcal{R}$. Let ν' denote the node in $G(r)$ (and so in g') that stems from the subterm t of r. Since t is linked with ν_0 in g, a linear term matches modulo laziness node ν_0 in g if and only if it matches modulo laziness node ν' in g'. So the subgraph of g' with root node ν' is a lazy normal form for $\mathcal{R}$.

4.3 Pattern Matching Modulo Laziness

We explain intuitively how laziness annotations in left-hand sides of rewrite rules can be taken into account in a pattern-matching algorithm. Assume that, while attempting to match a linear term ℓ with an active node ν in a lazy graph g, we encounter a lazy argument in ℓ. Then this subterm of ℓ and the node in g that should match it are stored as a pair, for future reference. Pattern matching proceeds to check whether ℓ matches modulo laziness node ν in g. If in the end this pattern match is successful, then we may have discovered several lazy arguments $\ell_1, \ldots, \ell_n$ in ℓ and corresponding nodes $\nu_1, \ldots, \nu_n$ in g on the way. (See O'Donnell [1985, Sect. 18] for several ways to represent and traverse the stack of pairs $\langle \ell_i, \nu_i \rangle$.) We make the rightmost node ν_n eager, and we reduce the subgraph rooted at ν_n. If this reduction produces a normal form, then we test whether this normal form matches the corresponding subterm ℓ_n of ℓ. If so, then the next rightmost node ν_{n-1} is made eager, etc. This procedure is repeated until

(1) either a normal form of some node ν_i does not match the corresponding subterm ℓ_i of ℓ, in which case it is concluded that ℓ does not match node ν;

(2) or all nodes $\nu_1, \ldots, \nu_n$ have been made eager, and their normal forms matched the corresponding subterms $\ell_1, \ldots, \ell_n$ of ℓ, in which case it is concluded that ℓ matches node ν.

In Fokkink et al. [1998] we introduced a pattern-matching algorithm for rightmost innermost rewriting with respect to a specificity relation, which is employed in the eager equational programming language Epic [Walters 1997; Walters and Kamperman 1996]. This pattern-matching algorithm, based on finite automata [Hoffmann and O'Donnell 1982a], can share matchings of the same function symbol in distinct patterns, thus supporting an efficient implementation; see Fokkink et al. [1998]. We adapt the transformation rules in Fokkink et al. [1998, Sect. 3.4] to minimize left-hand sides of rewrite rules, so that they take into account laziness annotations in left-hand sides.

Definition 4.3.1. A rewrite rule $f(x_1, \ldots, x_{ar(f)}) \to r$ with $x_1, \ldots, x_{ar(f)}$ distinct variables is called a *most general* rule for f.

We transform the rewrite rules of a left-linear TRS $\mathcal{R}$, so that the left-hand sides in the resulting TRS represent the success-transitions of a pattern-matching automaton for the lazy rewrite relation $\rightsquigarrow_L$ as induced by $\mathcal{R}$. Following the specificity rule (see Definition 4.1.1), first the eager arguments of a left-hand side in $\mathcal{R}$ are scanned from left to right, and next its lazy arguments are scanned from right to left. In the resulting TRS, left-hand sides contain no more than two function symbols, and lazy arguments in left-hand sides are variables. The transformation distinguishes two principles. The first principle reduces the size of a left-hand side with more than two function symbols, while the second principle makes a lazy nonvariable argument in a left-hand side eager.

(1) Assume that, for some function symbol f and an eager argument i of f, there is a rule in $\mathcal{R}$ with left-hand side $f(s_1, \ldots, s_{ar(f)})$ that contains more than two function symbols, where s_i is not a variable.

We add a fresh function symbol f^d to $\mathcal{F}$, which inherits the arity and the laziness annotation from f. All occurrences of the function symbol f *inside* left-hand sides of rewrite rules are replaced by f^d.

Each rewrite rule in $\mathcal{R}$ of the form

$$f(s_1, \ldots, s_{i-1}, g(t_1, \ldots, t_{ar(g)}), s_{i+1}, \ldots, s_{ar(f)}) \to r$$

with $s_j \in \mathcal{V}$ for eager arguments $j < i$ of f is replaced by a rule

$$f_g(s_1, \ldots, s_{i-1}, t_1, \ldots, t_{ar(g)}, s_{i+1}, \ldots, s_{ar(f)}) \to r. \tag{1}$$

Here, f_g is a fresh function symbol of arity $ar(f) + ar(g) - 1$, which inherits the laziness annotation from $f(\_, g(\_), \_)$:

—$\Lambda(f_g, j) \Leftrightarrow \Lambda(f, j)$ for $j = 1, \ldots, i-1$;
—$\Lambda(f_g, j) \Leftrightarrow \Lambda(g, j - i + 1)$ for $j = i, \ldots, ar(g) + i - 1$;
—$\Lambda(f_g, j) \Leftrightarrow \Lambda(f, j - ar(g) + 1)$ for $j = ar(g) + i, \ldots, ar(f) + ar(g) - 1$.

For each f_g we add a rule

$$\begin{aligned} & f(x_1, \ldots, x_{i-1}, g(y_1, \ldots, y_{ar(g)}), x_{i+1}, \ldots, x_{ar(f)}) \\ \to\ & f_g(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{ar(g)}, x_{i+1}, \ldots, x_{ar(f)}). \end{aligned}$$

For each f_g for which rule (1) does not contain a most general rule, we add a most general rule

$$\begin{array}{l} f_g(x_1,\ldots,x_{i-1},y_1,\ldots,y_{ar(g)},x_{i+1},\ldots,x_{ar(f)}) \\ \rightarrow\ f^d(x_1,\ldots,x_{i-1},g(y_1,\ldots,y_{ar(g)}),x_{i+1},\ldots,x_{ar(f)}). \end{array}$$

Left-hand sides $f(s_1,\ldots,s_{ar(f)})$ of rewrite rules with $s_j \in \mathcal{V}$ for eager arguments $j \leq i$ of f are replaced by $f^d(s_1,\ldots,s_{ar(f)})$. Finally, we add a rule

$$f(x_1,\ldots,x_{ar(f)}) \rightarrow f^d(x_1,\ldots,x_{ar(f)}).$$

(2) Assume that, for some function symbol f and a lazy argument i of f, there is a rule in $\mathcal{R}$ with left-hand side $f(s_1,\ldots,s_{ar(f)})$ where s_i is not a variable.

We add a fresh function symbol f^d of arity $ar(f)$, with laziness annotation:
—$\neg\Lambda(f^d,i)$;
—$\Lambda(f^d,j) \Leftrightarrow \Lambda(f,j)$ for $j \neq i$.

All occurrences of the function symbol f *inside* left-hand sides of rewrite rules are replaced by f^d.

Left-hand sides $f(s_1,\ldots,s_{ar(f)})$ of rewrite rules with $s_j \in \mathcal{V}$ for eager arguments j of f and for lazy arguments $j > i$ of f are replaced by $f^d(s_1,\ldots,s_{ar(f)})$. Finally, we add a rule

$$f(x_1,\ldots,x_{ar(f)}) \rightarrow f^d(x_1,\ldots,x_{ar(f)}).$$

The two transformation principles above are repeated until each left-hand side contains no more than two function symbols and lazy arguments in left-hand sides are single variables. This transformation halts on each left-linear TRS. Namely, each application of the first principle strictly decreases the sum of the sizes of left-hand sides with more than two function symbols, and this sum is not increased by applications of the second principle. Furthermore, each application of the second principle strictly decreases the total number of lazy nonvariable arguments in left-hand sides.

The transformation involves an extension of the alphabet with the function symbols f^d and f_g, and an adaptation of the class of normal forms: occurrences of f in normal forms are renamed into f^d. In Section A.2 in the appendix it is shown that the lazy rewrite relation $\rightsquigarrow_L$ with respect to the transformed TRS simulates the lazy rewrite relation $\rightsquigarrow_L$ with respect to the original TRS, and that this simulation is sound, complete, and termination preserving [Fokkink and van de Pol 1997]. Hence, no information on normal forms for the lazy rewrite relation $\rightsquigarrow_L$ with respect to the original TRS is lost.

The introduction of the function symbol f^d in the two transformation principles is reminiscent of a transformation by Thatte [1985] to make TRSes constructor-based; see Thatte [1988], Durand and Salinier [1993], Verma [1995] and Luttik et al. [1996] for subsequent improvements on the correctness of this transformation. Salinier and Strandh [1996] use a similar transformation technique to simulate a forward-branching TRS by a constructor-based strongly sequential TRS; they also prove correctness of their transformation.

Remark. Sherman et al. [1991] optimize the efficiency of the implementation of their pattern-matching automaton for equational programming [O'Donnell 1985] by the elimination of redundant intermediate rewrite steps. Basically, if a function symbol f is not a constructor, then a subterm $f(t_1, \ldots, t_{ar(f)})$ in the right-hand side of an equation is replaced by its possible reducts. In general this requires making the arguments t_i more specific, by substituting (open) terms for their variables. This optimization could also be applied with respect to our pattern-matching automaton for lazy rewriting.

4.4 Thunking Lazy Nodes in Lazy Graphs

The laziness annotation of a lazy graph and the rewrite relation $\rightsquigarrow_L$ can be implemented using thunks [Ingerman 1961], which enable one to abstract away from laziness annotations and to apply pure rightmost innermost rewriting.

Suppose we want to rewrite a lazy graph by means of $\rightsquigarrow_L$. Then, first the nodes of the lazy graph that are not active are thunked, to prevent that they can be rewritten in an eager fashion. If a node ν is not active, and has label f and children $\nu_1, \ldots, \nu_{ar(f)}$, then the subgraph $f(\nu_1, \ldots, \nu_{ar(f)})$ at ν is adapted to $\Theta(\tau_f, \mathbf{vec}_\alpha(\nu_1, \ldots, \nu_{ar(f)}))$, one where α is a string of $ar(f)$ zeros, to mark that the arguments of $\mathbf{vec}_\alpha$ are all lazy. This conversion turns the node ν into a normal form, while its label is stored as the token τ_f and its children are stored as the arguments of $\mathbf{vec}_\alpha$. The function symbols Θ, τ_f, and $\mathbf{vec}_\alpha$ for binary strings α are all fresh.

—Θ, called a thunk, is a binary function symbol with two eager arguments.

—The τ_f for $f \in \mathcal{F}$ are constants.

—Let α be a binary string. The function symbol $\mathbf{vec}_\alpha$ has arity $|\alpha|$, and for $i = 1, \ldots, |\alpha|$ the i-argument of $\mathbf{vec}_\alpha$ is lazy if and only if $\alpha_i = 0$. The $\mathbf{vec}_\alpha$ are auxiliary function symbols, which enable one to collect several nodes below the second argument of the thunk Θ.

Precisely, each lazy graph g over the original signature is adapted to a graph g_Θ over the extended signature as follows.

(1) If ν is active in g or if $\mathit{label}_\mathsf{g}(\nu) = x \in \mathcal{V}$, then

$$\begin{aligned} \mathit{label}_{\mathsf{g}_\Theta}(\nu) &= \mathit{label}_\mathsf{g}(\nu) \\ \mathit{childr}_{\mathsf{g}_\Theta}(\nu) &= \mathit{childr}_\mathsf{g}(\nu). \end{aligned}$$

(2) If ν is not active in g and $\mathit{label}_\mathsf{g}(\nu) = f \in \mathcal{F}$, then the children of ν are made lazy, and two fresh eager nodes ν' and ν'' are added to g_Θ; thus, we define

$$\begin{array}{llllll} \mathit{label}_{\mathsf{g}_\Theta}(\nu) = \Theta & & \mathit{label}_{\mathsf{g}_\Theta}(\nu') = \tau_f & & \mathit{label}_{\mathsf{g}_\Theta}(\nu'') = \mathbf{vec}_\alpha & \\ \mathit{childr}_{\mathsf{g}_\Theta}(\nu) = \nu' \cdot \nu'' & & \mathit{childr}_{\mathsf{g}_\Theta}(\nu') = \epsilon & & \mathit{childr}_{\mathsf{g}_\Theta}(\nu'') = \mathit{childr}_\mathsf{g}(\nu). & \end{array}$$

Here, α consists of $ar(f)$ zeros, and ϵ represents the empty string.

From now on we focus on rewriting lazy graphs over the extended signature. In the remainder of this section and in the next sections, we define and adapt rewrite rules to introduce and eliminate thunks.

Suppose a lazy graph over the extended signature is rewritten according to our lazy rewriting strategy, and an application of a rewrite rule makes a node with

the label Θ active. Then we want to restore the original graph structure at this node, because active nodes have to be rewritten in an eager fashion. Therefore, we introduce a fresh unary function symbol **inst** with an eager argument. Moreover, we add left-linear rewrite rules to the TRS, for function symbols $f \in \mathcal{F}$:

$$\mathbf{inst}(\Theta(\tau_f, \mathbf{vec}_\alpha(x_1, \ldots, x_{ar(f)}))) \;\rightarrow\; f(x_1, \ldots, x_{ar(f)})$$

where α denotes a string of $ar(f)$ zeros. These rewrite rules find application in Section 4.6, where so-called migrant variables in rewrite rules, which migrate from a lazy argument in the left-hand side to an active position in the right-hand side, are instantiated using the function symbol **inst**.

4.5 Thunking Lazy Arguments in Right-Hand Sides

Suppose a lazy graph over the extended signature is rewritten according to our lazy rewriting strategy, and suppose a rewrite rule is applied that contains lazy arguments in its right-hand side. Then we have to thunk these lazy arguments in the resulting reduct, according to the ideas explained in Section 4.4. This can be achieved by a transformation of the right-hand sides of rewrite rules.

Consider a left-linear rewrite rule $\ell \rightarrow r$, and let the collection $S(\ell, r)$ of nonvariable lazy arguments in r be nonempty. We select an element t in $S(\ell, r)$, such that its proper subterms are not in $S(\ell, r)$. (Namely, if t contains a proper nonvariable lazy argument s, then we first want to thunk s before thunking t; expanding t in an eager argument should not mean expanding s at the same time.) A $t \in S(\ell, r)$ that satisfies the criterion above can be determined efficiently in a bottom-up fashion. We introduce a fresh constant λ, which is added to the signature. Suppose t contains n variables $x_1, \ldots, x_n$. Let r' be obtained from r by replacing the lazy argument t in r by

$$\Theta(\lambda, \mathbf{vec}_\alpha(x_1, \ldots, x_n))$$

where $|\alpha| = n$, and $\alpha_i = 0$ if and only if x_i is a lazy argument in ℓ and does not occur at an active position in r. The rewrite rule $\ell \rightarrow r$ is replaced by

$$\ell \;\rightarrow\; r'.$$

Furthermore, one extra rewrite rule is introduced to restore the original term structure t:

$$\mathbf{inst}(\Theta(\lambda, \mathbf{vec}_\alpha(x_1, \ldots, x_n))) \;\rightarrow\; t$$

We continue to thunk the remaining nonvariable lazy arguments in the right-hand side of $\ell \rightarrow r'$.

4.6 Instantiation of Migrant Variables

Suppose the left-hand side of a most specific rewrite rule $\ell \rightarrow r$ matches modulo laziness a rightmost innermost active node ν in a lazy graph g without giving rise to essential lazy nodes. According to Definition 4.1.2 (1) this yields a rewrite step $\mathsf{g} \rightsquigarrow_L \mathsf{g}'$. If a variable x in ℓ is linked with a lazy node ν_x in g, and if x occurs at an active position in r, then ν_x becomes active in g'; see Definition 3.2.3 (1). Since ν_x is lazy, the variable x must occur as a lazy argument in ℓ. In order to be able to abstract away from laziness annotations, we make the transfer of variables from lazy to active explicit, using the function symbol **inst**.

Definition 4.6.1. A variable x is called *migrant* for a rewrite rule $\ell \to r$ if it occurs as a lazy argument in ℓ and at an active position in r.

The migrant variables of a rewrite rule are instantiated using the function symbol **inst** as follows. For each rewrite rule $\ell \to r$ we determine its collection of migrant variables. The term r' is obtained from r by replacing each migrant variable x by $\mathbf{inst}(x)$, and the rewrite rule $\ell \to r$ is replaced by

$$\ell \to r'.$$

Finally, we add a most general rewrite rule for **inst**, in case this function symbol is applied to a node that is not thunked:

$$\mathbf{inst}(x) \to x$$

4.7 Overview

Suppose we want to rewrite a lazy graph by means of a left-linear TRS over a lazy signature, using the lazy rewrite relation $\rightsquigarrow_L$. We give an overview of the transformations in the previous sections, which are applied so that this rewriting strategy can be performed by means of rightmost innermost rewriting.

(1) Multiple occurrences of a nonvariable term in the right-hand side of a rewrite rule are shared (Section 3.3).
(2) Left-hand sides of rewrite rules are minimized, so that they represent the success-transitions of an automaton for pattern matching modulo laziness with respect to the specificity relation (Section 4.3).
(3) Lazy arguments in right-hand sides of rewrite rules are thunked (Section 4.5).
(4) Rewrite rules are added that are able to eliminate thunks in right-hand sides of rewrite rules (Section 4.5).
(5) The migrant variables of a rewrite rule, which migrate from a lazy argument in the left-hand side to an active position in the right-hand side, are instantiated in the right-hand side (Section 4.6).
(6) A most general rule is added for the function symbol **inst** (Section 4.6).
(7) In the lazy graph that is to be rewritten, nodes that are not active are thunked (Section 4.4).
(8) General unthunk rules are added to support unthunking of arbitrary terms that may result from thunking input terms (Section 4.4).

Steps (1)–(6) and (8) are performed during compilation of the original TRS. Step (7) is performed at the input of the subject term.

Correctness of step (1) was addressed in Proposition 3.4.3, while correctness of step (2) is discussed in Section A.2 of the appendix. The correctness of steps (3)–(8) also needs to be addressed; these transformations involve an extension of the alphabet with the function symbols Θ, τ_f, $\mathbf{vec}_\alpha$, λ, and **inst**, and an adaptation of the class of normal forms. In Section A.3 of the appendix it is shown that eager rewriting with respect to the transformed TRS simulates the lazy rewrite relation $\rightsquigarrow_L$ with respect to the original TRS, and that this simulation is sound, complete, and termination preserving. Hence, no information on normal forms for the lazy rewrite relation $\rightsquigarrow_L$ is lost; see Fokkink and van de Pol [1997].

Suppose the transformed TRS reduces the thunked version of the lazy graph to a normal form, by means of rightmost innermost rewriting. This normal form may contain nodes with the label f^d or Θ. As a final step we apply the simulation mappings from the appendix to this normal form, which rename labels f^d into f, and which replace thunked nodes by their original graph structures.

4.8 An Example

We consider the left-linear TRS with overlapping left-hand sides that was used as a running example in the introduction. Its signature consists of the constant 0, the unary function symbols *succ* and *inf*, and the binary function symbols *nth* and *cons*, with only the second argument of *cons* lazy, and all other arguments eager. The TRS consists of three rewrite rules:

$$\begin{array}{lrcl} (1) & \mathit{inf}(x) & \to & \mathit{cons}(x, \mathit{inf}(\mathit{succ}(x))) \\ (2) & \mathit{nth}(x, \mathit{cons}(y, z)) & \to & y \\ (3) & \mathit{nth}(\mathit{succ}(x), \mathit{cons}(y, z)) & \to & \mathit{nth}(x, z) \end{array}$$

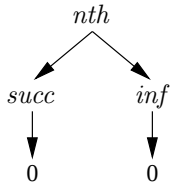
Owing to the specificity relation, rule (3) has priority over rule (2).

Recall that eager evaluation of the term $\mathit{nth}(\mathit{succ}(0), \mathit{inf}(0))$ leads to an infinite reduction. We show in detail how lazy rewriting as described in this article reduces this term to its desired normal form $\mathit{succ}(0)$. Lazy arguments in right-hand sides of rewrite rules are thunked; migrant variables in right-hand sides of rewrite rules are instantiated by means of the **inst** function; and a most general rule is added for **inst**. This produces the TRS

$$\begin{array}{lrcl} (1') & \mathit{inf}(x) & \to & \mathit{cons}(x, \Theta(\lambda, \mathbf{vec}_1(x))) \\ (2) & \mathit{nth}(x, \mathit{cons}(y, z)) & \to & y \\ (3') & \mathit{nth}(\mathit{succ}(x), \mathit{cons}(y, z)) & \to & \mathit{nth}(x, \mathbf{inst}(z)) \\ (4) & \mathbf{inst}(\Theta(\lambda, \mathbf{vec}_1(x))) & \to & \mathit{inf}(\mathit{succ}(x)) \\ (5) & \mathbf{inst}(x) & \to & x. \end{array}$$

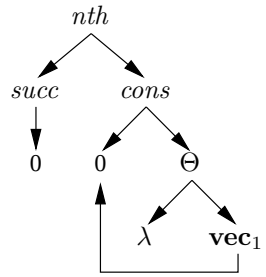
The lazy argument $\mathit{inf}(\mathit{succ}(x))$ in the right-hand side of rule (1) has been thunked, and rule (4) has been included for unthunking this argument. The migrant variable z in the right-hand side of rule (3) has been instantiated. Finally, rule (5) is the most general rule for **inst**.

Note that the term $\mathit{nth}(\mathit{succ}(0), \mathit{inf}(0))$ does not contain lazy arguments, so there is no need to include tokens τ_f, nor rewrite rules to unthunk these tokens, as described in Section 4.4. We transform $\mathit{nth}(\mathit{succ}(0), \mathit{inf}(0))$ into a graph:

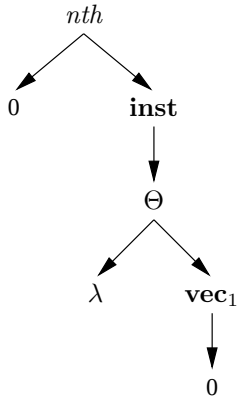


For simplicity of presentation, nodes are represented by their labels instead of by their names. We reduce this graph to its normal form, by means of rightmost innermost rewriting.

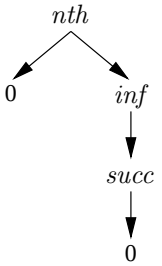
First, the redex at node *inf* of this graph is contracted by rule (1′):



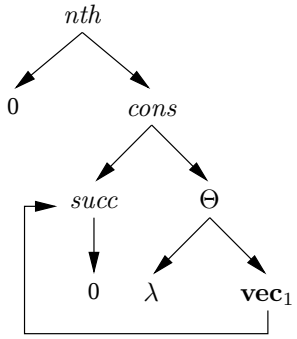
Next, the redex at the root node of this graph is contracted by rule (3′):



Next, the redex at node **inst** is contracted by rule (4):



Next, the redex at node *inf* is contracted by rule (1′):



Finally, the redex at the root node of this graph is contracted by rule (2):

This graph is in normal form. It does not contain thunks, so we transform this graph into the term $succ(0)$, and conclude that this is the normal form of $nth(succ(0), inf(0))$.

5. CONCLUSIONS AND RELATED WORK

We proposed a novel notion of lazy graph rewriting based on a laziness annotation of arguments of function symbols, in which subgraphs at lazy nodes are not required to be in normal form. We have defined transformation rules for left-linear rewrite systems, such that lazy rewriting with respect to the original rewrite system is simulated by eager evaluation with respect to the transformed rewrite system. Our work is somewhat similar in spirit to the use of evaluation transformers in Burn [1991], and to the protection constructors in the equational programming language Q, based on an eager pattern-matching strategy from Gräf [1991]. In context-sensitive term rewriting [Lucas 1995] it is indicated for each argument of a function symbol whether or not contraction of redexes inside such an argument is allowed. Recently, Giesl and Middeldorp [1999] presented a transformation of context-sensitive TRSes into ordinary TRSes such that the original context-sensitive TRS is terminating if and only if the resulting ordinary TRS is terminating.

Chew [1980] applied a directed version of congruence closure [Nelson and Oppen 1980] to obtain a term reduction method which avoids multiple evaluation of the same term structure. Similar to our notion of lazy rewriting, Chew's procedure may halt even when no normal form exists. Moreover, Chew's congruence closure can produce structural loops, leading to a finite representation of an infinite term, and the ability to terminate in cases where the normal form is not finite. Our notion of lazy rewriting does not introduce structural loops. The effect of laziness annotations in the presence of Chew's congruence closure deserves further study.

In Scheme [Rees and Clinger 1986] there are special data structures to delay the (eager) evaluation of a term and to force evaluation of such a delayed expression. Our transformations feature the introduction of similar data structures: the thunk Θ delays computation of the graph below it, while such a thunk can be eliminated by the function **inst**. The rewrite rules for **inst** can be implemented efficiently by representing the thunk Θ as a tag-bit, so that testing whether a node is a thunk can be performed as a bit test. Our representation of thunks in right-hand sides of rewrite rules is comparable to the frames used in TIM [Wray and Fairbairn 1989], the closures in the STG machine [Peyton Jones and Salkild 1989], and the closure in the heap in the $\langle \nu, G \rangle$ machine [Augustsson and Johnsson 1989].

Our implementation of lazy graph rewriting is inspired by the implementation of modern lazy functional languages. These implementations are lazy by nature, but are optimized to perform as much eager evaluation as possible. For example, Clean [Brus et al. 1987] supports the annotation of strict arguments, while OBJ3 [Goguen et al. 1993] supports annotations for the evaluation order of arguments. In Categorical ML, based on the Categorical Abstract Machine [Cousineau et al. 1987], lazy constructors are used to achieve similar effects as our transformations.

There, transformation of the program must be carried out manually for the most part.

Huet and Lévy [1991] consider orthogonal (so, in particular, nonoverlapping) rewrite systems that are sequential. This ensures that a term that is not in weak head normal form always contains a needed redex, which has to be reduced to normal form in any rewriting strategy. Strong sequentiality [Huet and Lévy 1991] allows one to determine such needed redexes efficiently. Strandh [1989] advocates restricting further to forward-branching equational programs, for which each state in the pattern-matching automaton can be reached by success-transitions only. This compares nicely with our notion of active nodes. Durand [1994] gives a quadratic algorithm to decide the forward-branching property; in contrast, no polynomial algorithm is known to decide strong sequentiality. Strandh [1989] presents an efficient innermost implementation of forward-branching equational programs, which maintains the outermost reduction strategy. This pattern-matching algorithm does not give rise to a pair stack of (lazy) nodes and patterns a la O'Donnell [1985] (see also Section 4.3). If all arguments of function symbols are annotated lazy, then our notion of lazy rewriting is closely related to Strandh's implementation method, with the distinction that we do not reduce inessential lazy nodes to normal form.

Conventionally, eager evaluation is referred to as call-by-value, while postponed evaluation without sharing results (such as lazy term rewriting) is referred to as call-by-name, and postponed evaluation with sharing results (such as lazy graph rewriting) is referred to as call-by-need. The relations between these three notions have been investigated in several articles. Plotkin [1975] gave simulations of call-by-name by call-by-value, and vice versa, in the context of the λ-calculus. In the context of higher-order functional programs, Amtoft [1993] developed an algorithm to transform call-by-name programs into call-by-value programs; Steckler and Wand [1994] used a data-flow analysis to minimize thunkification in this context. Okasaki et al. [1994] gave a continuation-passing style transformation of call-by-need into call-by-value, for a particular λ-calculus.

We list some advantages of our notion of lazy rewriting, compared to "classic" lazy evaluation.

(1) Some transformations of rewrite rules enable an implementation of lazy rewriting using the same primitives as an implementation of eager rewriting.

(2) A laziness annotation can produce normal forms for terms that do not have a finite reduction under any rewriting strategy.

(3) Only a small structure occurs at a (thunked) lazy node that is introduced in some reduct, and this structure is only expanded if the node is made active.

(4) Our implementation of lazy rewriting is fully integrated with pattern matching (based on the specificity of left-hand sides of rewrite rules).

(5) Rewrite rules can be overlapping, and the rewrite system does not need to be sequential.

(6) No rigid distinction is required between constructors and defined function symbols.

(7) Not only in theory, but also in practice, our technique does not rely on properties of built-in algebraic data types such as lists or trees.

Several abstract machines have been used for the implementation of modern lazy functional languages; these include the S-K reduction machine [Turner 1979], the G machine [Johnsson 1984], the Categorical Abstract Machine [Cousineau et al. 1987], TIM [Wray and Fairbairn 1989], the STG machine [Peyton Jones and Salkild 1989], the ABC machine [Plasmeijer and van Eekelen 1993], Gofer [Jones 1994], and HAM [Bailey 1995]. Bailey [1995, Chapt. 7] presents a detailed comparison between several of these abstract machines. We compare the cost of basic data structures and actions in our scheme with some of these implementations. It should be noted that it is extremely difficult to assess the effect of different design choices on performance (e.g., see Peyton Jones and Salkild [1989]), so we only give a qualitative discussion.

—In our approach, unthunking is only performed if all eager pattern matching is successful. This is not possible in lazy functional languages, where the order of pattern matching and its effects on the evaluation of subterms are fixed.

—In our approach, only a small structure occurs at lazy nodes. In contrast, the ABC machine [Plasmeijer and van Eekelen 1993] builds complete graphs at lazy nodes.

—In our approach, no run-time cost is incurred when all arguments in the original rewrite system are annotated eager. Even when all arguments are found to be strict, TIM and the STG machine make a function call to obtain the tag of a constructor term (this is the reason why they are called "tagless"), whereas our implementation only needs to dereference a pointer. Depending on the architecture, this can make quite a difference, because an instruction cache (if present) is invalidated by performing function calls.

—In an implementation that allows overwriting nodes with nodes of arbitrary size, there is no need for the dreaded indirection nodes [Peyton Jones 1987, Section 12.4]. In our transformed rewrite system this role is fulfilled by Θ; every node is evaluated exactly once, either by immediate innermost rewriting, or later on, after unthunking the node. In Peyton Jones [1987], the indirection nodes can only be transformed away after a complicated analysis.

A. CORRECTNESS OF THE TRANSFORMATIONS

We present formal correctness proofs for the transformations in Section 4.

A.1 Simulation

In Fokkink et al. [1998] we used a notion of simulation of one rewrite system by another rewrite system, to prove correctness of transformations of rewrite systems. A simulation consists of a partially defined, surjective mapping ϕ, which relates terms in the simulating rewrite system to terms in the original rewrite system. We focused on *deterministic* rewrite systems, in which each term can do no more than one rewrite step (cf. O'Donnell [1998, Def. 3.3.3]).

Consider a simulation ϕ with domain $D(\phi)$. The simulation ϕ is said to be *sound* if each reduction in the simulating rewrite system from a $b \in D(\phi)$ to a b' can be extended to a reduction to a $b'' \in D(\phi)$, such that $\phi(b)$ can be rewritten to $\phi(b'')$ in the original rewrite system. Furthermore, ϕ is *complete* if for each normal form b for the simulating rewrite system, $b \in D(\phi)$ and $\phi(b)$ is a normal form for the original rewrite system. Finally, ϕ *preserves termination* if for each term $b \in D(\phi)$

for which the original rewrite system is terminating for $\phi(b)$, the simulating rewrite system is terminating for b.

Soundness, completeness, and termination preservation are preserved under composition of simulations. Fokkink and van de Pol [1997] prove that if a simulation is sound, complete, and termination preserving, then no information on normal forms in the original rewrite system is lost. That is, there exist mappings *parse* from original to transformed terms and *print* from transformed to original terms such that for each original term t its normal form can be computed as follows: compute the normal form of $parse(t)$ and apply the *print* function to it.

In Fokkink and van de Pol [1997], correctness is proved for fully defined simulations, but the proof easily extends to partially defined simulations; see Fokkink et al. [1998, App. A]. Allowing partially defined simulations enables one to abstract away from fresh function symbols in the simulating system that are always eliminated from a reduct immediately after they are introduced in the reduct (such as the auxiliary function symbols f_g in the minimization of left-hand sides). This can reduce the number of cases that have to be considered in the proofs of soundness, completeness, and termination preservation.

Definition A.1.1. An *abstract reduction system (ARS)* consists of a collection $\mathbb{A}$ of elements, together with a binary reduction relation R between elements in $\mathbb{A}$.

Definition A.1.2. A *simulation* of an ARS $(\mathbb{A}, R)$ by an ARS $(\mathbb{B}, S)$ is a surjective mapping $\phi : D(\phi) \to \mathbb{A}$ with $D(\phi) \subseteq \mathbb{B}$.

In the remainder of this section we focus on deterministic ARSes $(\mathbb{A}, R)$, in which each element $a \in \mathbb{A}$ can do an R-step to no more than one element a'. Let R^* denote the reflexive transitive closure of R.

Definition A.1.3. Let the deterministic ARS $(\mathbb{B}, S)$ simulate the deterministic ARS $(\mathbb{A}, R)$ by means of a surjective mapping $\phi : D(\phi) \to \mathbb{A}$.

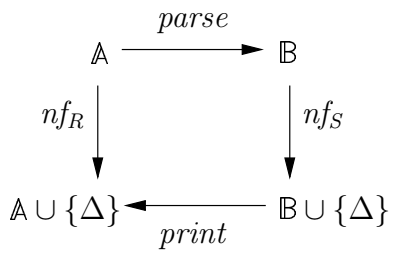
—ϕ is *sound* if, for each $b \in D(\phi)$ and $b' \in \mathbb{B}$ with bSb', there is a $b'' \in D(\phi)$ with $b'S^*b''$ and either $\phi(b) = \phi(b'')$ or $\phi(b)R\phi(b'')$.

—ϕ is *complete* if, for each normal form $b \in \mathbb{B}$ for S, $b \in D(\phi)$ and $\phi(b)$ is a normal form for R.

—ϕ *preserves termination* if, for each $b_0 \in D(\phi)$ that induces an infinite S-reduction $b_0Sb_1Sb_2S\cdots$, there is a $k \geq 1$ such that $b_k \in D(\phi)$ and $\phi(b_0)R\phi(b_k)$.

For a deterministic ARS $(\mathbb{A}, R)$, we define that $nf_R : \mathbb{A} \to \mathbb{A} \cup \{\Delta\}$ maps each $a \in \mathbb{A}$ to its normal form for R; elements that do not have a normal form, because they induce an infinite R-reduction, are mapped to Δ (which represents "divergence").

Definition A.1.4. A deterministic ARS $(\mathbb{B}, S)$ is a *correct transformation* of a deterministic ARS $(\mathbb{A}, R)$ if there exist mappings $parse : \mathbb{A} \to \mathbb{B}$ and $print : \mathbb{B} \to \mathbb{A}$ such that the diagram below commutes:



where $print(\Delta) = \Delta$.

We note that the composition of two correct transformations is again a correct transformation.

Theorem A.1.5. If a simulation ϕ between two deterministic ARSes is sound, complete, and termination preserving, then it is a correct transformation.

PROOF. See Fokkink et al. [1998, App. A]. (Basically, *parse* is some inverse of ϕ, and *print* is ϕ.) □

A.2 Minimization of Left-Hand Sides

We prove that the procedure in Section 4.3 to minimize left-hand sides is correct. Let $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denote the original lazy signature and $\mathbb{G}(\Sigma)$ the collection of lazy graphs over Σ. Furthermore, $\mathcal{R}$ is the original left-linear TRS and $\mathcal{S}$ the TRS that results after applying one of the two transformation principles in Section 4.3 to $\mathcal{R}$. Finally, $\rightsquigarrow_{\mathcal{R}}$ and $\rightsquigarrow_{\mathcal{S}}$ represent the lazy rewrite relation $\rightsquigarrow_L$ as induced by $\mathcal{R}$ and $\mathcal{S}$, respectively. We prove correctness of the two transformation principles in Section 4.3 separately, using simulations.

A.2.1 *First Transformation Rule.* Suppose we apply the first transformation rule to the left-linear TRS $\mathcal{R}$, with respect to $f \in \mathcal{F}$ and eager argument i of f. The transformation introduces fresh function symbols: f^d has the same arity and laziness annotation as f, while the f_g for $g \in \mathcal{F}$ have arity $ar(f) + ar(g) - 1$ and inherit their laziness annotation from $f(\_, g(\_), \_)$:

—$\Lambda(f_g, j) \Leftrightarrow \Lambda(f, j)$ for $j = 1, \ldots, i-1$;
—$\Lambda(f_g, j) \Leftrightarrow \Lambda(g, j - i + 1)$ for $j = i, \ldots, ar(g) + i - 1$;
—$\Lambda(f_g, j) \Leftrightarrow \Lambda(f, j - ar(g) + 1)$ for $j = ar(g) + i, \ldots, ar(f) + ar(g) - 1$.

Let $\hat{s}$ be the term that is obtained from term s by renaming all occurrences of f into f^d. Moreover, let $\hat{f} = f^d$ and $\hat{g} = g$ for $g \in \mathcal{F} \backslash \{f\}$. The transformed TRS $\mathcal{S}$ consists of six collections $\mathcal{S}_0$–$\mathcal{S}_5$:

—$g(\hat{s}_1, \ldots, \hat{s}_{ar(g)}) \to r \in \mathcal{S}_0$
iff $g(s_1, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ with $g \neq f$ or $s_j \notin \mathcal{V}$ for some eager argument $j < i$ of f;
—$f(x_1, \ldots, x_{i-1}, \hat{g}(y_1, \ldots, y_{ar(g)}), x_{i+1}, \ldots, x_{ar(f)}) \to$
$f_{\hat{g}}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{ar(g)}, x_{i+1}, \ldots, x_{ar(f)}) \in \mathcal{S}_1$
iff $f(s_1, \ldots, s_{i-1}, g(t_1, \ldots, t_{ar(g)}), s_{i+1}, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments $j < i$ of f;

—$f_{\hat{g}}(\hat{s}_1, \ldots, \hat{s}_{i-1}, \hat{t}_1, \ldots, \hat{t}_{ar(g)}, \hat{s}_{i+1}, \ldots, \hat{s}_{ar(f)}) \to r \in \mathcal{S}_2$
iff $f(s_1, \ldots, s_{i-1}, g(t_1, \ldots, t_{ar(g)}), s_{i+1}, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments $j < i$ of f;

—$f_{\hat{g}}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{ar(g)}, x_{i+1}, \ldots, x_{ar(f)}) \to$
$f^d(x_1, \ldots, x_{i-1}, \hat{g}(y_1, \ldots, y_{ar(g)}), x_{i+1}, \ldots, x_{ar(f)}) \in \mathcal{S}_3$
iff $f(s_1, \ldots, s_{i-1}, g(t_1, \ldots, t_{ar(g)}), s_{i+1}, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments $j < i$ of f, and there is no most general rule for $f_{\hat{g}}$ in $\mathcal{S}_2$;

—$\mathcal{S}_4 = \{f(x_1, \ldots, x_{ar(f)}) \to f^d(x_1, \ldots, x_{ar(f)})\}$;

—$f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r \in \mathcal{S}_5$
iff $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments $j \leq i$ of f.

Let Σ' denote the original signature Σ extended with the fresh function symbols f^d and f_g, and let

$$\mathbb{B} \;=\; \{\mathsf{g} \in \mathbb{G}(\Sigma') \mid \exists \mathsf{g}_0 \in \mathbb{G}(\Sigma)\,(\mathsf{g}_0 \leadsto^*_{\mathcal{S}} \mathsf{g})\}.$$

The next two lemmas follow by induction with respect to the length of a derivation $\mathsf{g}_0 \leadsto^*_{\mathcal{S}} \mathsf{g}$ with $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$.

LEMMA A.2.1. *If* $\mathsf{g} \in \mathbb{B}$, *then the subgraphs rooted at active nodes of* g *are in* $\mathbb{B}$.

LEMMA A.2.2. *If* $\mathsf{g} \in \mathbb{B}$, *then only active nodes in* g *can carry labels* f_g.

We prove that the transformation of $(\mathbb{G}(\Sigma), \leadsto_{\mathcal{R}})$ into $(\mathbb{B}, \leadsto_{\mathcal{S}})$ is correct, by means of a simulation. The mapping $\phi : D(\phi) \to \mathbb{G}(\Sigma)$ is defined for lazy graphs in $\mathbb{B}$ that do not contain labels f_g. The lazy graph $\phi(\mathsf{g})$ is obtained from $\mathsf{g} \in D(\phi)$ by renaming all labels f^d into f. The mapping ϕ is surjective, because it is the identity mapping on $\mathbb{G}(\Sigma) \subseteq D(\phi)$. Hence, ϕ constitutes a (partially defined) simulation of $(\mathbb{G}(\Sigma), \leadsto_{\mathcal{R}})$ by $(\mathbb{B}, \leadsto_{\mathcal{S}})$. We proceed to prove that ϕ is sound, complete, and termination preserving.

COMPLETENESS. If $\mathsf{g} \in \mathbb{B}$ is a lazy normal form for $\mathcal{S}$, then $\mathsf{g} \in D(\phi)$ and $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$.

PROOF. We apply induction on the number of nodes in g. If the root node of g has a variable as label, then clearly $\mathsf{g} \in D(\phi)$ and $\phi(\mathsf{g}) = \mathsf{g}$ is a lazy normal form for $\mathcal{R}$. We focus on the case where the root node of g has a label $h \in \mathcal{F} \cup \{f^d\}$, and children $\nu_1 \ldots \nu_{ar(h)}$.

g is a lazy normal form for $\mathcal{S}$, and $\mathcal{S}$ contains most general rules for the function symbols f_g, so the active nodes in g are not labeled f_g. Together with Lemma A.2.2 this implies that g does not contain labels f_g at all, so $\mathsf{g} \in D(\phi)$.

Let g_j for $j = 1, \ldots, ar(h)$ denote the subgraph of g that is rooted at ν_j. Lemma A.2.1 says that $\mathsf{g}_j \in \mathbb{B}$, and so $\mathsf{g}_j \in D(\phi)$, for eager arguments j of h. Since g is a lazy normal form for $\mathcal{S}$, the g_j for eager arguments j of h are lazy normal forms for $\mathcal{S}$. So by induction the $\phi(\mathsf{g}_j)$ for eager arguments j of h are lazy normal forms for $\mathcal{R}$. Hence, to conclude that $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$, it suffices to show that none of the left-hand sides of rules in $\mathcal{R}$ match modulo laziness the root node of $\phi(\mathsf{g})$. We distinguish four different forms of left-hand sides in $\mathcal{R}$.

1 Let $g(s_1, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ with $g \neq f$.

Then $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)}) \to r$ is in $\mathcal{S}_0$. Since g is a lazy normal form for $\mathcal{S}$, the term $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)})$ does not match modulo laziness the root node of g. The root node of g has label h and children $\nu_1 \ldots \nu_{ar(h)}$, so we can distinguish two cases.

1.1 $h \neq g$.
Since $g \neq f$, it follows that g is not the root label of $\phi(\mathsf{g})$. So $g(s_1, \ldots, s_{ar(g)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

1.2 $\hat{s}_j$ does not match modulo laziness node ν_j in g, for some eager argument j of h.
g is a lazy normal form for $\mathcal{S}$, and $\mathcal{S}$ contains most general rules for function symbols f_g and f, so the active nodes in g are not labeled f_g or f. Therefore, active nodes in $\phi(\mathsf{g})$ with label f correspond to active nodes in g with label f^d. So, since $\hat{s}_j$ does not match modulo laziness node ν_j in g, and j is an eager argument h, s_j does not match modulo laziness node ν_j in $\phi(\mathsf{g})$. Hence, $g(s_1, \ldots, s_{ar(g)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

2 Let $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \notin \mathcal{V}$ for some eager argument $j < i$ of f. Then $f(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r$ is in $\mathcal{S}_0$. We distinguish two cases for the root label h of g: $h = f^d$ or $h \neq f^d$.

2.1 $h \neq f^d$.
Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, the root label of $\phi(\mathsf{g})$ is then unequal to f. So $f(s_1, \ldots, s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

2.2 $h = f^d$.
Let g' be obtained from g by renaming its root label f^d into f. Since $\mathsf{g} \in \mathbb{B}$, $\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g}$ for some $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$. The outermost function symbols of right-hand sides of rules in $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$, and $\mathcal{S}_5$, are unequal to f^d. Therefore, the label f^d of the root node of g must have been introduced by the reduction of g' to g, either in one step by applying rule $\mathcal{S}_4$ to the root node of g', or in two steps by subsequent applications of rules in $\mathcal{S}_1$ and $\mathcal{S}_3$ to the root node of g'.
$\hat{s}_j \notin \mathcal{V}$ for an eager argument $j < i$ of f, so $f(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r \in \mathcal{S}_0$ is more specific than any of the rules in $\mathcal{S}_1 \cup \mathcal{S}_4$. Hence, its left-hand side cannot match modulo laziness the root node of g'. That is, $\hat{s}_k$ for some eager argument k of f does not match modulo laziness node ν_k in g'. So $\hat{s}_k$ does not match modulo laziness node ν_k in g. Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, this implies that s_k does not match modulo laziness node ν_k in $\phi(\mathsf{g})$. Hence, $f(s_1, \ldots, s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

3 Let $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments $j < i$ of f and $s_i = g(t_1, \ldots, t_{ar(g)})$.
Then

$$\begin{aligned} & f(x_1, \ldots, x_{i-1}, \hat{g}(y_1, \ldots, y_{ar(g)}), x_{i+1}, \ldots, x_{ar(f)}) \\ \to\ & f_{\hat{g}}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{ar(g)}, x_{i+1}, \ldots, x_{ar(f)}) \end{aligned}$$

is in $\mathcal{S}_1$ and $f_{\hat{g}}(\hat{s}_1, \ldots, \hat{s}_{i-1}, \hat{t}_1, \ldots, \hat{t}_{ar(g)}, \hat{s}_{i+1}, \ldots, \hat{s}_{ar(f)}) \to r$ is in $\mathcal{S}_2$; these rules are abbreviated to ρ_1 and ρ_2, respectively. Let h' denote the label of node ν_i in g. We distinguish three cases, depending on whether or not $h = f^d$ and $h' = \hat{g}$.

3.1 $h \neq f^d$.

Then the root label of $\phi(\mathsf{g})$ is unequal to f, so $f(s_1,\ldots,s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

3.2 $h' \neq \hat{g}$.
Then the label of node ν_i in $\phi(\mathsf{g})$ is unequal to g, so $s_i = g(t_1,\ldots,t_{ar(g)})$ does not match modulo laziness node ν_i in $\phi(\mathsf{g})$. Since i is an eager argument of f, it follows that $f(s_1,\ldots,s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

3.3 $h = f^d$ and $h' = \hat{g}$.
Let g' be obtained from g by renaming its root label f^d into f. Since $\mathsf{g} \in \mathbb{B}$, $\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ for some $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$. The outermost function symbols of right-hand sides of rules in $\mathcal{S}_0, \mathcal{S}_1, \mathcal{S}_2$, and $\mathcal{S}_5$, are unequal to f^d. Therefore, the label f^d of the root node of g has been introduced by the reduction of g' to g in two steps, by subsequent applications to the root node of g' of rule ρ_1 in $\mathcal{S}_1$ and rule

$$\begin{aligned} & f_{\hat{g}}(x_1,\ldots,x_{i-1},y_1,\ldots,y_{ar(g)},x_{i+1},\ldots,x_{ar(f)}) \\ \rightarrow\ & f^d(x_1,\ldots,x_{i-1},\hat{g}(y_1,\ldots,y_{ar(g)}),x_{i+1},\ldots,x_{ar(f)}) \end{aligned}$$

in $\mathcal{S}_3$; this last rule is abbreviated to ρ_3. Let g'' denote the lazy graph that results after application of ρ_1 to the root node of g'; in other words, $\mathsf{g}' \leadsto_{\mathcal{S}} \mathsf{g}'' \leadsto_{\mathcal{S}} \mathsf{g}$.
Let $\nu'_1 \ldots \nu'_{ar(g)}$ denote the children of node ν_i in g. Rule $\rho_2 \in \mathcal{S}_2$ is more specific than rule $\rho_3 \in \mathcal{S}_3$, so the left-hand side of ρ_2 cannot match modulo laziness the root node of g''. Then either $\hat{s}_k$ for some eager argument $k \neq i$ of f does not match modulo laziness node ν_k in g'', or $\hat{t}_l$ for some eager argument l of g does not match modulo laziness node ν'_l in g''. So either $\hat{s}_k$ or $\hat{t}_l$ does not match modulo laziness node ν_k or ν'_l in g, respectively. Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, this implies that s_k or t_l does not match modulo laziness node ν_k or ν'_l in $\phi(\mathsf{g})$, respectively. So $f(s_1,\ldots,s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

4 Let $f(s_1,\ldots,s_{ar(f)}) \rightarrow r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments $j \leq i$ of f.
Then $f^d(\hat{s}_1,\ldots,\hat{s}_{ar(f)}) \rightarrow r$ is in $\mathcal{S}_5$. Since g is a lazy normal form for $\mathcal{S}$, the term $f^d(\hat{s}_1,\ldots,\hat{s}_{ar(f)})$ does not match modulo laziness the root node of g. The root node of g has label h and children $\nu_1 \ldots \nu_{ar(h)}$, so we can distinguish two cases.

4.1 $h \neq f^d$.
Then $\phi(\mathsf{g})$ does not have root label f, so $f(s_1,\ldots,s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

4.2 $\hat{s}_k$ does not match modulo laziness node ν_k in g, for some eager argument k of h.
Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, this implies that s_k does not match modulo laziness node ν_k in $\phi(\mathsf{g})$. Hence, $f(s_1,\ldots,s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

We conclude that none of the left-hand sides of rules in $\mathcal{R}$ match modulo laziness the root node of $\phi(\mathsf{g})$. So $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$. □

SOUNDNESS. If $\mathsf{g} \in D(\phi)$ and $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$, then there is a $\mathsf{g}'' \in D(\phi)$ such that $\mathsf{g}' \leadsto_{\mathcal{S}}^* \mathsf{g}''$ and either $\phi(\mathsf{g}) = \phi(\mathsf{g}'')$ or $\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}'')$.

PROOF. $\mathsf{g} \in \mathbb{B}$, so $\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ for some $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$; hence, $\mathsf{g}' \in \mathbb{B}$. Let the rightmost innermost active $\mathcal{S}$-redex of g be rooted at node ν. Rule $\mathcal{S}_4$ is most

general for f, so active descendants of ν cannot carry the label f. Hence, active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. We distinguish four cases, covering the rewrite rules in $\mathcal{S}$ that can match modulo laziness node ν in g.

1 Suppose $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ is the result of an application of $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)}) \to r$ in $\mathcal{S}_0$. This match gives rise to a (possibly empty) collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in g.
$\mathsf{g}' \in \mathbb{B}$ does not contain function symbols f_h (because g and r are free of such function symbols), so $\mathsf{g}' \in D(\phi)$. The term $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)})$ is the most specific left-hand side in $\mathcal{S}$ that matches modulo laziness node ν in g, and active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. This implies that $g(s_1, \ldots, s_{ar(g)})$ is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. Moreover, this match gives rise to the same collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in $\phi(\mathsf{g})$. Completeness of ϕ implies that ν is the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$. Hence, application of $g(s_1, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ gives rise to

$$\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}').$$

So we can take $\mathsf{g}'' = \mathsf{g}'$.

2 Suppose $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ is the result of an application of

$$\begin{array}{l} \quad f(x_1, \ldots, x_{i-1}, \hat{g}(y_1, \ldots, y_{ar(g)}), x_{i+1}, \ldots, x_{ar(f)}) \\ \to\ f_{\hat{g}}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{ar(g)}, x_{i+1}, \ldots, x_{ar(f)}) \end{array}$$

in $\mathcal{S}_1$.
ν is the rightmost innermost active $\mathcal{S}$-redex of g, so ν is also the rightmost innermost active $\mathcal{S}$-redex of g'. We distinguish two cases, covering the rewrite rules in $\mathcal{S}$ that can match modulo laziness node ν in g'.

2.1 Suppose $f_{\hat{g}}(\hat{s}_1, \ldots, \hat{s}_{i-1}, \hat{t}_1, \ldots, \hat{t}_{ar(g)}, \hat{s}_{i+1}, \ldots, \hat{s}_{ar(f)}) \to r$ in $\mathcal{S}_2$ is the most specific rule in $\mathcal{S}$ that matches modulo laziness node ν in g'. Application of this rule gives rise to

$$\mathsf{g}' \leadsto_{\mathcal{S}} \mathsf{g}''$$

where $\mathsf{g}'' \in \mathbb{B}$ does not contain function symbols f_h (because g and r are free of such function symbols), so $\mathsf{g}'' \in D(\phi)$.
$f_{\hat{g}}(\hat{s}_1, \ldots, \hat{s}_{i-1}, \hat{t}_1, \ldots, \hat{t}_{ar(g)}, \hat{s}_{i+1}, \ldots, \hat{s}_{ar(f)})$ is the most specific left-hand side in $\mathcal{S}$ that matches modulo laziness node ν in g, and active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. This implies that $f(s_1, \ldots, s_{i-1}, g(t_1, \ldots, t_{ar(g)}), s_{i+1}, \ldots, s_{ar(g)})$ is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. Completeness of ϕ implies that ν is the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$. Hence, application of $f(s_1, \ldots, s_{i-1}, g(t_1, \ldots, t_{ar(g)}), s_{i+1}, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ gives rise to

$$\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}'').$$

2.2 Suppose

$$\begin{array}{l} \quad f_{\hat{g}}(x_1, \ldots, x_{i-1}, y_1, \ldots, y_{ar(g)}, x_{i+1}, \ldots, x_{ar(f)}) \\ \to\ f^d(x_1, \ldots, x_{i-1}, \hat{g}(y_1, \ldots, y_{ar(g)}), x_{i+1}, \ldots, x_{ar(f)}) \end{array}$$

in $\mathcal{S}_3$ is the most specific rule in $\mathcal{S}$ that matches modulo laziness node ν in g'. Application of this rule gives rise to

$$\mathsf{g}' \rightsquigarrow_{\mathcal{S}} \mathsf{g}''$$

where g'' is obtained from g by renaming the label f of ν into f^d. Then $\mathsf{g}'' \in \mathbb{B}$ does not contain function symbols f_h, so $\mathsf{g}'' \in D(\phi)$, and moreover

$$\phi(\mathsf{g}) = \phi(\mathsf{g}'').$$

3 Suppose $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ is the result of an application of $f(x_1, \ldots, x_{ar(f)}) \to f^d(x_1, \ldots, x_{ar(f)})$ in $\mathcal{S}_4$.
Then $\mathsf{g}' \in \mathbb{B}$ does not contain function symbols f_h, so $\mathsf{g}' \in D(\phi)$, and moreover

$$\phi(\mathsf{g}) = \phi(\mathsf{g}').$$

So we can take $\mathsf{g}'' = \mathsf{g}'$.

4 Suppose $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ is the result of an application of $f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r$ in $\mathcal{S}_5$. This match gives rise to a (possibly empty) collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in g.
$\mathsf{g}' \in \mathbb{B}$ does not contain function symbols f_h (because g and r are free of such function symbols), so $\mathsf{g}' \in D(\phi)$. The term $f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)})$ is the most specific left-hand side in $\mathcal{S}$ that matches modulo laziness node ν in g, and active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. This implies that $f(s_1, \ldots, s_{ar(f)})$ is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. Moreover, this match gives rise to the same collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in $\phi(\mathsf{g})$. Completeness of ϕ implies that ν is the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$. Hence, application of $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ gives rise to

$$\phi(\mathsf{g}) \rightsquigarrow_{\mathcal{R}} \phi(\mathsf{g}').$$

So we can take $\mathsf{g}'' = \mathsf{g}'$. □

Termination Preservation. If $\mathsf{g}_0 \in D(\phi)$ induces an infinite reduction $\mathsf{g}_0 \rightsquigarrow_{\mathcal{S}} \mathsf{g}_1 \rightsquigarrow_{\mathcal{S}} \mathsf{g}_2 \rightsquigarrow_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $\mathsf{g}_k \in D(\phi)$ and $\phi(\mathsf{g}_0) \rightsquigarrow_{\mathcal{R}} \phi(\mathsf{g}_k)$.

Proof. Each application of rule $\mathcal{S}_4$ and each subsequent application of rules in $\mathcal{S}_1$ and $\mathcal{S}_3$ renames a label f into f^d. Since g_0 contains only finitely many nodes with the label f, there can only be a finite number of such applications in a row with respect to g_0. Hence, there exists a smallest $l \geq 1$ such that either $\mathsf{g}_{l-1} \rightsquigarrow_{\mathcal{S}} \mathsf{g}_l$ is the result of an application of a rule in $\mathcal{S}_0 \cup \mathcal{S}_5$, or $\mathsf{g}_{l-1} \rightsquigarrow_{\mathcal{S}} \mathsf{g}_l \rightsquigarrow_{\mathcal{S}} \mathsf{g}_{l+1}$ is the result of subsequent application of rules in $\mathcal{S}_1$ and $\mathcal{S}_2$.

—$\mathsf{g}_0 \rightsquigarrow^*_{\mathcal{S}} \mathsf{g}_{l-1}$ is the result of applications of the rule $\mathcal{S}_4$ or subsequent applications of rules in $\mathcal{S}_1$ and $\mathcal{S}_3$. So g_{l-1} is obtained from g_0 by renaming labels f into f^d. Then $\mathsf{g}_{l-1} \in D(\phi)$ and $\phi(\mathsf{g}_0) = \phi(\mathsf{g}_{l-1})$.

—If $\mathsf{g}_{l-1} \rightsquigarrow_{\mathcal{S}} \mathsf{g}_l$ is the result of an application of a rule in $\mathcal{S}_0$ or $\mathcal{S}_5$, then case 1 or 4 in the soundness proof of ϕ, respectively, yields $\mathsf{g}_l \in D(\phi)$ and $\phi(\mathsf{g}_{l-1}) \rightsquigarrow_{\mathcal{R}} \phi(\mathsf{g}_l)$. If $\mathsf{g}_{l-1} \rightsquigarrow_{\mathcal{S}} \mathsf{g}_l \rightsquigarrow_{\mathcal{S}} \mathsf{g}_{l+1}$ is the result of an application of rules in $\mathcal{S}_1$ and $\mathcal{S}_2$, then case 2.1 in the soundness proof of ϕ yields $\mathsf{g}_{l+1} \in D(\phi)$ and $\phi(\mathsf{g}_{l-1}) \rightsquigarrow_{\mathcal{R}} \phi(\mathsf{g}_{l+1})$.

□

A.2.2 *Second Transformation Rule.* Suppose we apply the second transformation rule to $\mathcal{R}$, with respect to $f \in \mathcal{F}$ and lazy argument i of f. The transformation introduces a fresh function symbol f^d of the same arity as f, and laziness annotation $\neg\Lambda(f^d, i)$ and $\Lambda(f^d, j) \Leftrightarrow \Lambda(f, j)$ for $j \neq i$. Let $\hat{s}$ be the term that is obtained from term s by renaming all occurrences of f into f^d. The transformed TRS $\mathcal{S}$ consists of three collections $\mathcal{S}_0$–$\mathcal{S}_2$:

—$g(\hat{s}_1, \ldots, \hat{s}_{ar(g)}) \to r \in \mathcal{S}_0$
iff $g(s_1, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ with $g \neq f$, $s_j \notin \mathcal{V}$ for some eager argument j of f, or $s_j \notin \mathcal{V}$ for some lazy argument $j > i$ of f;
—$\mathcal{S}_1 = \{f(x_1, \ldots, x_{ar(f)}) \to f^d(x_1, \ldots, x_{ar(f)})\}$;
—$f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r \in \mathcal{S}_2$
iff $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments j of f and all lazy arguments $j > i$ of f.

Let Σ' denote the original signature Σ extended with the fresh function symbol f^d, and let

$$\mathbb{B} \;=\; \{\mathsf{g} \in \mathbb{G}(\Sigma') \mid \exists \mathsf{g}_0 \in \mathbb{G}(\Sigma)\,(\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g})\}.$$

The next lemma follows by induction with respect to the length of a derivation $\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g}$ with $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$.

LEMMA A.2.3. *If* $\mathsf{g} \in \mathbb{B}$*, then the subgraphs rooted at active nodes of* g *are in* $\mathbb{B}$.

We prove that the transformation of $(\mathbb{G}(\Sigma), \leadsto_{\mathcal{R}})$ into $(\mathbb{B}, \leadsto_{\mathcal{S}})$ is correct, by means of a simulation. The lazy graph $\phi(\mathsf{g})$ is obtained from $\mathsf{g} \in \mathbb{B}$ by renaming all labels f^d into f. The mapping $\phi : \mathbb{B} \to \mathbb{G}(\Sigma)$ is surjective, because it is the identity mapping on $\mathbb{G}(\Sigma) \subseteq \mathbb{B}$. Hence, ϕ constitutes a (fully defined) simulation of $(\mathbb{G}(\Sigma), \leadsto_{\mathcal{R}})$ by $(\mathbb{B}, \leadsto_{\mathcal{S}})$. We proceed to prove that ϕ is sound, complete, and termination preserving.

COMPLETENESS. If $\mathsf{g} \in \mathbb{B}$ is a lazy normal form for $\mathcal{S}$, then $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$.

PROOF. We apply induction on the number of nodes in g. If the root node of g has a variable as label, then clearly $\phi(\mathsf{g}) = \mathsf{g}$ is a lazy normal form for $\mathcal{R}$. We focus on the case where the root node of g has a label $h \in \mathcal{F} \cup \{f^d\}$, and children $\nu_1 \ldots \nu_{ar(h)}$.

Let g_j for $j = 1, \ldots, ar(h)$ denote the subgraph of g that is rooted at ν_j. Lemma A.2.3 says that $\mathsf{g}_j \in \mathbb{B}$ for eager arguments j of h. Since g is a lazy normal form for $\mathcal{S}$, the g_j for eager arguments j of h are lazy normal forms for $\mathcal{S}$. So by induction the $\phi(\mathsf{g}_j)$ for eager arguments j of h are lazy normal forms for $\mathcal{R}$. Hence, to conclude that $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$, it suffices to show that none of the left-hand sides of rules in $\mathcal{R}$ match modulo laziness the root node of $\phi(\mathsf{g})$. We distinguish three different forms of left-hand sides in $\mathcal{R}$.

1 Let $g(s_1, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ with $g \neq f$.
Then $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)}) \to r$ is in $\mathcal{S}_0$. Since g is a lazy normal form for $\mathcal{S}$, the term $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)})$ does not match modulo laziness the root node of g. The root node of g has label h and children $\nu_1 \ldots \nu_{ar(h)}$, so we can distinguish two cases.

1.1 $h \neq g$.
Since $g \neq f$, it follows that g is not the root label of $\phi(\mathsf{g})$. So $g(s_1, \ldots, s_{ar(g)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

1.2 $\hat{s}_j$ does not match modulo laziness node ν_j in g, for some eager argument j of h.
g is a lazy normal form for $\mathcal{S}$, and $\mathcal{S}_1$ is a most general rule for f; so the active nodes in g are not labeled f. Therefore, active nodes in $\phi(\mathsf{g})$ with label f correspond to active nodes in g with label f^d. So, since $\hat{s}_j$ does not match modulo laziness node ν_j in g, and j is an eager argument h, s_j does not match modulo laziness node ν_j in $\phi(\mathsf{g})$. Hence, $g(s_1, \ldots, s_{ar(g)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

2 Let $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \notin \mathcal{V}$ for some eager argument j of f or some lazy argument $j > i$ of f.
Then $f(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r$ is in $\mathcal{S}_0$. We distinguish two cases for the root label h of g: $h = f^d$ or $h \neq f^d$.

2.1 $h \neq f^d$.
Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, the root label of $\phi(\mathsf{g})$ is then unequal to f. So $f(s_1, \ldots, s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

2.2 $h = f^d$.
Let g' be obtained from g by renaming its root label f^d into f. Since $\mathsf{g} \in \mathbb{B}$, $\mathsf{g}_0 \rightsquigarrow_{\mathcal{S}}^* \mathsf{g}$ for some $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$. The outermost function symbols of right-hand sides of rules in $\mathcal{S}_0$ and $\mathcal{S}_2$ are unequal to f^d, so the label f^d of the root node of g must have been introduced by applying rule $\mathcal{S}_1$ to the root node of g'.
$\hat{s}_j \notin \mathcal{V}$ for an eager argument j of f or a lazy argument $j > i$ of f, so the rule $f(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r \in \mathcal{S}_0$ is more specific than $\mathcal{S}_1$. Hence, its left-hand side cannot match modulo laziness the root node of g'. That is, $\hat{s}_k$ for some eager argument k of f does not match modulo laziness node ν_k in g'. So $\hat{s}_k$ does not match modulo laziness node ν_k in g. Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, this implies that s_k does not match modulo laziness node ν_k in $\phi(\mathsf{g})$. Hence, $f(s_1, \ldots, s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

3 Let $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ with $s_j \in \mathcal{V}$ for all eager arguments j of f and all lazy arguments $j > i$ of f.
Then $f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r$ is in $\mathcal{S}_5$. Since g is a lazy normal form for $\mathcal{S}$, the term $f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)})$ does not match modulo laziness the root node of g. The root node of g has label h and children $\nu_1 \ldots \nu_{ar(h)}$, so we can distinguish two cases.

3.1 $h \neq f^d$.
Then $\phi(\mathsf{g})$ does not have root label f, so $f(s_1, \ldots, s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

3.2 $\hat{s}_k$ does not match modulo laziness node ν_k in g, for some eager argument k of h.
Since labels f of active nodes in $\phi(\mathsf{g})$ correspond to labels f^d in g, this implies that s_k does not match modulo laziness node ν_k in $\phi(\mathsf{g})$. Hence, $f(s_1, \ldots, s_{ar(f)})$ does not match modulo laziness the root node of $\phi(\mathsf{g})$.

We conclude that none of the left-hand sides of rules in $\mathcal{R}$ match modulo laziness the root node of $\phi(\mathsf{g})$. So $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$. □

SOUNDNESS. If $\mathsf{g} \in \mathbb{B}$ and $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$, then $\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}')$.

PROOF. $\mathsf{g} \in \mathbb{B}$, so $\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ for some $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$; hence, $\mathsf{g}' \in \mathbb{B}$. Let the rightmost innermost active $\mathcal{S}$-redex of g be rooted at node ν. Rule $\mathcal{S}_1$ is most general for f, so active descendants of ν cannot carry the label f. Hence, active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. We distinguish three cases, covering the rewrite rules in $\mathcal{S}$ that can match modulo laziness node ν in g.

1 Suppose $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ is the result of an application of $g(\hat{s}_1, \ldots, \hat{s}_{ar(g)}) \to r$ in $\mathcal{S}_0$. This match gives rise to a (possibly empty) collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in g.
$g(\hat{s}_1, \ldots, \hat{s}_{ar(g)})$ is the most specific left-hand side in $\mathcal{S}$ that matches modulo laziness node ν in g, and active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. This implies that $g(s_1, \ldots, s_{ar(g)})$ is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. Moreover, this match gives rise to the same collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in $\phi(\mathsf{g})$. Completeness of ϕ implies that ν is the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$. Hence, application of $g(s_1, \ldots, s_{ar(g)}) \to r \in \mathcal{R}$ gives rise to

$$\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}').$$

2 Suppose $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ is the result of an application of $f(x_1, \ldots, x_{ar(f)}) \to f^d(x_1, \ldots, x_{ar(f)})$ in $\mathcal{S}_4$. The lazy graph $\phi(\mathsf{g}')$ only differs from $\phi(\mathsf{g})$ in that the ith child of ν is eager
$f(x_1, \ldots, x_{ar(f)})$ is the most specific left-hand side in $\mathcal{S}$ that matches modulo laziness node ν in g, and active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. This implies that some $f(s_1, \ldots, s_{ar(f)})$ with $s_i \notin \mathcal{V}$, $s_j \in \mathcal{V}$ for eager arguments j, and $s_j \in \mathcal{V}$ for lazy arguments $j > i$ of f, is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. This match gives rise to essential lazy nodes in $\phi(\mathsf{g})$, the rightmost of which is the ith child of ν. Completeness of ϕ implies that ν is the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$. Hence,

$$\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}').$$

3 Suppose $\mathsf{g} \leadsto_{\mathcal{S}} \mathsf{g}'$ is the result of an application of $f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)}) \to r$ in $\mathcal{S}_2$. This match gives rise to a (possibly empty) collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in g.
$f^d(\hat{s}_1, \ldots, \hat{s}_{ar(f)})$ is the most specific left-hand side in $\mathcal{S}$ that matches modulo laziness node ν in g, and active descendants of ν in $\phi(\mathsf{g})$ with label f correspond to nodes in g with label f^d. This implies that $f(s_1, \ldots, s_{ar(f)})$ is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. Moreover, this match gives rise to the same collection $\{\nu_1, \ldots, \nu_n\}$ of essential lazy nodes in $\phi(\mathsf{g})$. Completeness of ϕ implies that ν is the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$. Hence, application of $f(s_1, \ldots, s_{ar(f)}) \to r \in \mathcal{R}$ gives rise to

$$\phi(\mathsf{g}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}').$$

This finishes the soundness proof of ϕ. □

TERMINATION PRESERVATION. If $\mathsf{g}_0 \in \mathbb{B}$ induces an infinite reduction $\mathsf{g}_0 \leadsto_{\mathcal{S}} \mathsf{g}_1 \leadsto_{\mathcal{S}} \mathsf{g}_2 \leadsto_{\mathcal{S}} \cdots$, then $\phi(\mathsf{g}_0) \leadsto_{\mathcal{R}} \phi(\mathsf{g}_1)$.

PROOF. Since $\mathsf{g}_0 \leadsto_{\mathcal{S}} \mathsf{g}_1$, the soundness property that we derived for ϕ induces that $\phi(\mathsf{g}_0) \leadsto_{\mathcal{R}} \phi(\mathsf{g}_1)$. □

A.3 Thunkification

We prove that the transformation rules in Sections 4.4, 4.5, and 4.6 to thunk and unthunk lazy nodes are correct. Let $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denote the original lazy signature and $\mathbb{G}(\Sigma)$ the collection of lazy graphs over Σ. Owing to the minimization of left-hand sides in Section 4.3, the left-hand sides of rewrite rules in the original left-linear $\mathcal{R}$ contain no more than two function symbols, and only variables as lazy arguments. Note that, since lazy arguments in left-hand sides of rewrite rules in $\mathcal{R}$ are always variables, pattern matching with respect to these left-hand sides does not give rise to essential lazy nodes. $\mathcal{S}$ denotes the TRS that results after applying the transformation rules in Sections 4.4, 4.5, and 4.6 to $\mathcal{R}$. Finally, $\leadsto_{\mathcal{R}}$ represents the lazy rewrite relation $\leadsto_L$ as induced by $\mathcal{R}$, while $\leadsto_{\mathcal{S}}$ represents the rightmost innermost rewrite relation with respect to specificity ordering as induced by $\mathcal{S}$. We prove correctness of the transformation using a simulation.

The transformation introduces fresh function symbols: the thunk Θ with two eager arguments, constants τ_f for $f \in \mathcal{F}$, auxiliary function symbols $\mathbf{vec}_\alpha$ for bit strings α of arity $|\alpha|$ with $\Lambda(\mathbf{vec}_\alpha, i)$ if and only if $\alpha_i = 0$, constants λ, and the function symbol **inst** with one eager argument. The transformed TRS $\mathcal{S}$ consists of four collections $\mathcal{S}_0$–$\mathcal{S}_3$:

—$\ell \to r' \in \mathcal{S}_0$
iff $\ell \to r \in \mathcal{R}$ and r' is obtained from r as follows:
  —we replace, in a bottom-up fashion, lazy nonvariable subterms t by thunks $\Theta(\lambda, \mathbf{vec}_\alpha(x_1, \ldots, x_{|\alpha|}))$, with $x_1, \ldots, x_{|\alpha|}$ the variables in t, and $\alpha_i = 0$ if and only if x_i is a lazy argument in ℓ and does not occur at an active position in r;
  —we replace migrant variables x in r by $\mathbf{inst}(x)$.

—$\mathbf{inst}(\Theta(\tau_f, \mathbf{vec}_\alpha(x_1, \ldots, x_{ar(f)}))) \to f(s_1, \ldots, s_{ar(f)}) \in \mathcal{S}_1$
iff $f \in \mathcal{F}$, α is a string of $ar(f)$ zeros, $s_i = \mathbf{inst}(x_i)$ for eager arguments i of f, and $s_i = x_i$ for lazy arguments i of f.

—$\mathbf{inst}(\Theta(\lambda, \mathbf{vec}_\alpha(x_1, \ldots, x_{|\alpha|}))) \to t' \in \mathcal{S}_2$
iff λ and $\mathbf{vec}_\alpha(x_1, \ldots, x_{|\alpha|})$ together represent the term structure t, and t' is obtained from t by replacing migrant variables x in t by $\mathbf{inst}(x)$.

—$\mathcal{S}_3 = \{\mathbf{inst}(x) \to x\}$.

Let Σ' denote the original signature Σ extended with the fresh function symbols Θ, τ_f, $\mathbf{vec}_\alpha$, λ, and **inst**, and let

$$\mathbb{B} \;=\; \{\mathsf{g} \in \mathbb{G}(\Sigma') \mid \exists \mathsf{g}_0 \in \mathbb{G}(\Sigma)\,(\mathsf{g}_0 \leadsto_{\mathcal{S}}^* \mathsf{g})\}.$$

We prove that the transformation of $(\mathbb{G}(\Sigma), \leadsto_{\mathcal{R}})$ into $(\mathbb{B}, \leadsto_{\mathcal{S}})$ is correct, by means of a simulation ϕ. This mapping eliminates nodes with the label **inst**, making the nodes below eager, and replaces thunked graph structures $\Theta(\tau_f, \mathbf{vec}_\alpha(\nu_1, \ldots, \nu_{ar(f)}))$ and $\Theta(\lambda, \mathbf{vec}_\alpha(\nu_1, \ldots, \nu_{|\alpha|}))$ by the related graph structures $f(\nu_1, \ldots, \nu_{ar(f)})$ and

$G(t)$, respectively. The lazy graph $\phi(\mathsf{g}) \in \mathbb{G}(\Sigma)$ is obtained from $\mathsf{g} \in \mathbb{B}$ by repeated applications of the transformation rules below.

1 Let node ν have label **inst** and child ν'.
We rename, in the image of the children mapping, all occurrences of ν into ν'. Next, we eliminate node ν from the lazy graph. Finally, we make node ν' eager.

2 Let node ν have label Θ and children $\nu'\nu''$. We distinguish two cases.

2.1 Let node ν' have label τ_f. Then ν'' carries the label $\mathbf{vec}_\alpha$ where α consists of $ar(f)$ zeros, and ν'' has as children the string of lazy nodes $\nu_1 \dots \nu_{ar(f)}$.
We change the label of ν into f, and supply this node with a string of children $\nu_1 \dots \nu_{ar(f)}$. For $i = 1, \dots, ar(f)$, node ν_i is made eager if and only if i is an eager argument of f. Next, we eliminate nodes ν' and ν'' from the lazy graph.

2.2 Let node ν' have label λ. Then ν'' carries label $\mathbf{vec}_\alpha$ and children $\nu_1 \dots \nu_{|\alpha|}$. Let λ and $\mathbf{vec}_\alpha(x_1, \dots, x_{|\alpha|})$ be related to the term t. We construct the graph $G(t)$, which consists of fresh nodes, and add $G(t)$ to the lazy graph. In the image of the children mapping, all occurrences of ν are renamed into the root node of $G(t)$. For $i = 1, \dots, |\alpha|$, let ν'_i denote the node in $G(t)$ with the label x_i; occurrences of ν'_i in the image of the children mapping are renamed into ν_i. Lazy nodes ν_i for $i = 1, \dots, |\alpha|$ are made eager if x_i is an eager argument in t. Finally, we eliminate the nodes ν, ν', ν'', and ν'_i for $i = 1, \dots, |\alpha|$.

The two cases above are applied to g until there are no nodes left with the label **inst** or Θ. The resulting lazy graph is $\phi(\mathsf{g})$.

The next two lemmas follow by induction with respect to the length of a derivation $\mathsf{g}_0 \leadsto^*_{\mathcal{S}} \mathsf{g}$ with $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$.

LEMMA A.3.1. *For all $\mathsf{g} \in \mathbb{B}$, $\phi(\mathsf{g}) \in \mathbb{G}(\Sigma)$ is well-defined.*

LEMMA A.3.2. *If $\mathsf{g} \in \mathbb{B}$, $label_{\mathsf{g}}(\nu) = f$, and i is an eager argument of f, then the ith child of ν in g is an eager node in $\phi(\mathsf{g})$.*

The next lemma follows from case 1 in the definition of ϕ, by induction with respect to the length of a path in g from the root node to the active node in question.

LEMMA A.3.3. *If $\mathsf{g} \in \mathbb{B}$ does not contain labels* **inst**, *then each active node in $\phi(\mathsf{g})$ inherits its label and children from the corresponding node in g.*

The mapping $\phi : \mathbb{B} \to \mathbb{G}(\Sigma)$ is surjective, because it is the identity mapping on $\mathbb{G}(\Sigma) \subseteq \mathbb{B}$. Hence, ϕ constitutes a (fully defined) simulation of $(\mathbb{G}(\Sigma), \leadsto_{\mathcal{R}})$ by $(\mathbb{B}, \leadsto_{\mathcal{S}})$. We proceed to prove that ϕ is sound, complete, and termination preserving.

COMPLETENESS. If $\mathsf{g} \in \mathbb{B}$ is a normal form for $\mathcal{S}$, then $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$.

PROOF. Since g is a normal form for $\mathcal{S}$, and $\mathcal{S}_3$ is a most general rule for **inst**, g does not contain labels **inst**. Consider any active node ν in $\phi(\mathsf{g})$. Lemma A.3.3 implies that ν inherits its label and children from node ν in g. To conclude that $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$, we need to show that none of the left-hand sides

in $\mathcal{R}$ match modulo laziness node ν in $\phi(\mathsf{g})$. We distinguish the two possible forms of left-hand sides in $\mathcal{R}$.

1 Let $f(x_1,\ldots,x_{ar(f)}) \to r$ in $\mathcal{R}$.
Then $f(x_1,\ldots,x_{ar(f)}) \to r'$ in $\mathcal{S}_0$ for some term r'. Since g is a normal form for $\mathcal{S}$, $f(x_1,\ldots,x_{ar(f)})$ does not match node ν in g. This implies $label_{\mathsf{g}}(\nu) \neq f$, and so $label_{\phi(\mathsf{g})}(\nu) \neq f$. Hence, $f(x_1,\ldots,x_{ar(f)})$ does not match modulo laziness node ν in $\phi(\mathsf{g})$.

2 Let $f(x_1,\ldots,x_{i-1},g(y_1,\ldots,y_{ar(g)}),x_{i+1},\ldots,x_{ar(f)}) \to r$ in $\mathcal{R}$.
Then $f(x_1,\ldots,x_{i-1},g(y_1,\ldots,y_{ar(g)}),x_{i+1},\ldots,x_{ar(f)}) \to r'$ in $\mathcal{S}_0$ for some term r'. Since g is a normal form for $\mathcal{S}$, $f(x_1,\ldots,x_{i-1},g(y_1,\ldots,y_{ar(g)}),x_{i+1},\ldots,x_{ar(f)})$ does not match node ν in g. This implies either $label_{\mathsf{g}}(\nu) \neq f$ or $label_{\mathsf{g}}(\nu') \neq g$, where ν' is the ith child of ν in g. Since i is an eager argument of f, Lemma A.3.2 implies that ν is an active node in $\phi(\mathsf{g})$. So by Lemma A.3.3, ν' inherits its label from node ν' in g. Hence, either $label_{\phi(\mathsf{g})}(\nu) \neq f$ or $label_{\phi(\mathsf{g})}(\nu') \neq g$. So $f(x_1,\ldots,x_{i-1},g(y_1,\ldots,y_{ar(g)}),x_{i+1},\ldots,x_{ar(f)})$ does not match modulo laziness node ν in $\phi(\mathsf{g})$.

We conclude that $\phi(\mathsf{g})$ is a lazy normal form for $\mathcal{R}$. □

The next lemma follows by induction with respect to the length of a derivation $\mathsf{g}_0 \rightsquigarrow_{\mathcal{S}}^* \mathsf{g}$ with $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$.

LEMMA A.3.4. *If $\mathsf{g} \in \mathbb{B}$ and node ν has label Θ in g, then the left-hand sides of rules in $\mathcal{S}$ do not match any descendants of ν in g.*
SOUNDNESS. If $\mathsf{g} \in \mathbb{B}$ and $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$, then either $\phi(\mathsf{g}) = \phi(\mathsf{g}')$ or $\phi(\mathsf{g}) \rightsquigarrow_{\mathcal{R}} \phi(\mathsf{g}')$.

PROOF. $\mathsf{g} \in \mathbb{B}$, so $\mathsf{g}_0 \rightsquigarrow_{\mathcal{S}}^* \mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ for some $\mathsf{g}_0 \in \mathbb{G}(\Sigma)$; hence, $\mathsf{g}' \in \mathbb{B}$. Let the node the rightmost innermost $\mathcal{S}$-redex of g be rooted at node ν. We distinguish three cases, covering the rewrite rules in $\mathcal{S}$ that can match node ν in g.

1 Let $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ be the result of an application of $\ell \to r'$ in $\mathcal{S}_0$.
By Lemma A.3.4 the node ν does not have an ancestor Θ in g, so it is active in $\phi(\mathsf{g})$. Since ℓ is the most specific left-hand side in $\mathcal{S}$ that matches node ν in g, it follows that ℓ is the most specific left-hand side in $\mathcal{R}$ that matches modulo laziness node ν in $\phi(\mathsf{g})$. Completeness of ϕ implies that the rightmost innermost active $\mathcal{R}$-redex of $\phi(\mathsf{g})$ is rooted at ν. Hence, using the definition of ϕ, we conclude that application of $\ell \to r \in \mathcal{R}$ gives rise to

$$\phi(\mathsf{g}) \rightsquigarrow_{\mathcal{R}} \phi(\mathsf{g}').$$

2 Let $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ be the result of an application of $\mathbf{inst}(\Theta(\tau_f, \mathbf{vec}_\alpha(x_1,\ldots,x_{ar(f)}))) \to f(s_1,\ldots,s_{ar(f)})$ in $\mathcal{S}_1$.
Then, owing to the definition of ϕ, we have $\phi(\mathsf{g}) = \phi(\mathsf{g}')$.

3 Let $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ be the result of an application of $\mathbf{inst}(\Theta(\lambda, \mathbf{vec}_\alpha(x_1,\ldots,x_{|\alpha|}))) \to t'$ in $\mathcal{S}_2$.
Then, owing to the definition of ϕ, we have $\phi(\mathsf{g}) = \phi(\mathsf{g}')$.

4 Let $\mathsf{g} \rightsquigarrow_{\mathcal{S}} \mathsf{g}'$ be the result of an application of $\mathbf{inst}(x) \to x$ in $\mathcal{S}_3$.
Then, owing to the definition of ϕ, we have $\phi(\mathsf{g}) = \phi(\mathsf{g}')$.

This finishes the soundness proof of ϕ. □

TERMINATION PRESERVATION. If $\mathsf{g}_0 \in \mathbb{B}$ induces an infinite reduction $\mathsf{g}_0 \leadsto_{\mathcal{S}} \mathsf{g}_1 \leadsto_{\mathcal{S}} \mathsf{g}_2 \leadsto_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $\phi(\mathsf{g}_0) \leadsto_{\mathcal{R}} \phi(\mathsf{g}_k)$.

PROOF. First, we observe that the TRS $\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$ is terminating.

—Each application of a rule in $\mathcal{S}_1 \cup \mathcal{S}_2$ strictly decreases the number of labels Θ, and this number is not increased by applications of the rule in $\mathcal{S}_3$.

—Each application of the rule in $\mathcal{S}_3$ strictly decreases the number of labels **inst**.

So there can only be a finite number of applications of rules in $\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$ in a row. Then there exists a smallest $k \geq 1$ such that $\mathsf{g}_{k-1} \leadsto_{\mathcal{S}} \mathsf{g}_k$ is the result of an application of a rule in $\mathcal{S}_0$.

1 Since $\mathsf{g}_{i-1} \leadsto_{\mathcal{S}}^* \mathsf{g}_i$ for $i = 1, \ldots, k-1$ is the result of an application of a rule in $\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$, the cases 2, 3, and 4 in the soundness proof of ϕ yield $\phi(\mathsf{g}_{i-1}) = \phi(\mathsf{g}_i)$.

2 Since $\mathsf{g}_{k-1} \leadsto_{\mathcal{S}} \mathsf{g}_k$ is the result of an application of a rule in $\mathcal{S}_0$, case 1 in the soundness proof of ϕ yields $\phi(\mathsf{g}_{k-1}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}_k)$.

Hence, $\phi(\mathsf{g}_0) = \cdots = \phi(\mathsf{g}_{k-1}) \leadsto_{\mathcal{R}} \phi(\mathsf{g}_k)$. □

ACKNOWLEDGMENTS

Jaco van de Pol, Piet Rodenburg, and Mark van den Brand provided valuable support. We are indebted to anonymous referees for suggesting many substantial improvements. A considerable part of this research was carried out at the CWI in Amsterdam.

REFERENCES

AMTOFT, T. 1993. Minimal thunkification. In *Proceedings, 3rd Workshop on Static Analysis*, P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, Eds. Lecture Notes in Computer Science, vol. 724. Springer-Verlag, Berlin, 218–229.

AUGUSTSSON, L. AND JOHNSSON, T. 1989. Parallel graph reduction with the $\langle \nu, g \rangle$-machine. In *Proceedings, 5th Conference on Functional Programming Languages and Computer Architecture*. ACM, New York, 202–213.

BAADER, F. AND NIPKOW, T. 1998. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK.

BAETEN, J. C. M., BERGSTRA, J. A., KLOP, J. W., AND WEIJLAND, W. P. 1989. Term-rewriting systems with rule priorities. *Theor. Comput. Sci. 67,* 2/3, 283–301.

BAILEY, S. W. 1995. Hielp, a fast interactive lazy functional language system. Ph.D. thesis, University of Chicago, Chicago. Available at http://www.cs.uchicago.edu/tech-reports/.

BARENDREGT, H. P., VAN EEKELEN, M. C. J. D., GLAUERT, J. R. W., KENNAWAY, J. R., PLASMEIJER, M. J., AND SLEEP, M. R. 1987. Term graph rewriting. In *Proceedings, 1st Conference on Parallel Architectures and Languages Europe*, J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, Eds. Lecture Notes in Computer Science, vol. 259. Springer-Verlag, Berlin, 141–158.

BRUS, T., VAN EEKELEN, M. C. J. D., VAN LEER, M., PLASMEIJER, M. J., AND BARENDREGT, H. P. 1987. Clean – a language for functional graph rewriting. In *Proceedings, 3rd Conference on Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 364–384.

BURN, G. 1991. *Lazy Functional Languages: Abstract Interpretation and Compilation*. Research Monographs in Parallel and Distributed Computing. Pitman, London.

CHEW, L. P. 1980. An improved algorithm for computing with equations. In *Proceedings, 21st IEEE Symposium on Foundations of Computer Science*. IEEE, 108–117.

Cousineau, G., Curien, P.-L., and Mauny, M. 1987. The categorical abstract machine. *Sci. Comput. Program. 8,* 2, 173–202.

Durand, I. 1994. Bounded, strongly sequential and forward-branching term rewriting systems. *Journal of Symbolic Computation 18,* 4, 319–352.

Durand, I. and Salinier, B. 1993. Constructor equivalent term rewriting systems. *Inf. Process. Lett. 47,* 3, 131–137.

Fokkink, W. J., Kamperman, J. F. T., and Walters, H. R. 1998. Within ARM's reach: compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Trans. Program. Lang. Syst. 20,* 3, 679–706.

Fokkink, W. J. and van de Pol, J. C. 1997. Simulation as a correct transformation of rewrite systems. In *Proceedings, 22nd Symposium on Mathematical Foundations of Computer Science*, I. Prívara and P. Ružička, Eds. Lecture Notes in Computer Science, vol. 1295. Springer-Verlag, Berlin, 249–258.

Friedman, D. P. and Wise, D. S. 1976. CONS should not evaluate its arguments. In *Proceedings, 3rd Colloquium on Automata, Languages and Programming*, S. Michaelson and R. Milner, Eds. Edinburgh University Press, Edinburgh, 257–284.

Giesl, J. and Middeldorp, A. 1999. Transforming context-sensitive rewrite systems. In *Proceedings, 10th Conference on Rewriting Techniques and Applications*, P. Narendran and M. Rusinowitch, Eds. Lecture Notes in Computer Science, vol. 1632. Springer-Verlag, Berlin, 271–285.

Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J.-P. 1993. *Applications of Algebraic Specifications using OBJ.* Cambridge University Press, Cambridge, UK.

Gräf, A. 1991. Left-to-right tree pattern matching. In *Proceedings, 4th Conference on Rewriting Techniques and Applications*, R. V. Book, Ed. Lecture Notes in Computer Science, vol. 488. Springer-Verlag, Berlin, 323–334.

Hartel, P. H., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C. H., Grieskamp, W., van Groningen, J. H. G., Hammond, K., Hausman, B., Ivory, M. Y., Jones, R. E., Kamperman, J. F. T., Lee, P., Leroy, X., Lins, R. D., Loosemore, S., Rjemo, N., Serrano, M., Talpin, J.-P., Thackray, J., Thomas, S., Walters, H. R., Weis, P., and Wentworth, P. 1996. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *J. Funct. Program. 6,* 4, 621–655.

Henderson, P. and Morris, J. H. 1976. A lazy evaluator. In *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages.* ACM, New York, 95–103.

Hoffmann, C. M. and O'Donnell, M. J. 1982a. Pattern matching in trees. *J. ACM 29,* 1, 68–95.

Hoffmann, C. M. and O'Donnell, M. J. 1982b. Programming with equations. *ACM Trans. Program. Lang. Syst. 4,* 1, 83–112.

Huet, G. and Lévy, J.-J. 1991. Computations in orthogonal rewriting systems, parts i and ii. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. MIT Press, Cambridge, Mass., 395–443.

Ingerman, P. Z. 1961. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM 4,* 1, 55–58.

Johnsson, T. 1984. Efficient compilation of lazy evaluation. In Proceedings, ACM Symposium on Compiler Construction. *ACM SIGPLAN Not. 19,* 6, 58–69.

Jones, M. P. 1994. The implementation of the Gofer functional programming system. Tech. Rep. YALEU/DCS/RR-1030, Yale University, New Haven. Available at http://www.cse.ogi.edu/ mpj/pubs/goferimp.html.

Kamperman, J. F. T. 1996. Compilation of term rewriting systems. Ph.D. thesis, University of Amsterdam, Amsterdam. Available at http://www.babelfish.nl.

Kamperman, J. F. T. and Walters, H. R. 1995. Lazy rewriting on eager machinery. In *Proceedings, 6th Conference on Rewriting Techniques and Applications*, J. Hsiang, Ed. Lecture Notes in Computer Science, vol. 914. Springer-Verlag, Berlin, 147–162.

Kamperman, J. F. T. and Walters, H. R. 1996. Minimal term rewriting systems. In *Proceedings, 11th Workshop on Specification of Abstract Data Types*, M. Haveraaen, O. Owe, and O.-J. Dahl, Eds. Lecture Notes in Computer Science, vol. 1130. Springer-Verlag, Berlin, 274–290.

Kennaway, J. R. 1990. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In *Proceedings, 3rd European Symposium on Programming*, N. D. Jones, Ed. Lecture Notes in Computer Science, vol. 432. Springer-Verlag, Berlin, 256–270.

Kennaway, J. R., Klop, J. W., Sleep, M. R., and de Vries, F. J. 1994. On the adequacy of graph rewriting for simulating term rewriting. *ACM Trans. Program. Lang. Syst. 16,* 3, 493–523.

Lucas, S. 1995. Fundamentals of context-sensitive rewriting. In *Proceedings, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, M. Bartosek, J. Staudek, and J. Wiedermann, Eds. Lecture Notes in Computer Science, vol. 1012. Springer-Verlag, Berlin, 405–412.

Luttik, S. P., Rodenburg, P. H., and Verma, R. M. 1996. Correctness criteria for transformations of rewrite systems with an application to Thatte's transformation. Technical Report P9615, University of Amsterdam, Amsterdam.

Mycroft, A. 1980. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings, 4th Symposium on Programming*, B. Robinet, Ed. Lecture Notes in Computer Science, vol. 83. Springer-Verlag, Berlin, 269–281.

Nelson, G. and Oppen, D. C. 1980. Fast decision procedures based on congruence closure. *J. ACM 27,* 2, 356–364.

O'Donnell, M. J. 1977. *Computing in Systems Described by Equations*. Lecture Notes in Computer Science, vol. 58. Springer-Verlag, Berlin.

O'Donnell, M. J. 1985. *Equational Logic as a Programming Language*. MIT Press, Cambridge, Mass.

O'Donnell, M. J. 1998. Introduction: logic and logic programming languages. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, D. M. Gabbay, Ed. Vol. 5. Oxford University Press, New York, 1–67.

Ogata, K. and Futatsugi, K. 1997. Implementation of term rewritings with the evaluation strategy. In *Proceedings, 9th Symposium on Programming Languages: Implementations, Logics, and Programs*, H. Glaser, P. H. Hartel, and H. Kuchen, Eds. Lecture Notes in Computer Science, vol. 1292. Springer-Verlag, Berlin, 225–239.

Okasaki, C., Lee, P., and Tarditi, D. 1994. Call-by-need and continuation-passing style. *Lisp and Symbolic Computation 7,* 1, 57–81.

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, N.J.

Peyton Jones, S. L. and Salkild, J. 1989. The spineless tagless G-machine. In *Proceedings, 4th Conference on Functional Programming Languages and Computer Architecture*, D. B. MacQueen, Ed. Addison-Wesley, Reading, Mass., 184–201.

Plasmeijer, M. J. August 1998. Personal communication.

Plasmeijer, M. J. and van Eekelen, M. C. J. D. 1993. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, Mass.

Plotkin, G. D. 1975. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci. 1,* 1, 125–159.

Rees, J. A. and Clinger, W., Eds. 1986. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Not. 21,* 12, 37–79.

Salinier, B. and Strandh, R. I. 1996. Efficient simulation of forward-branching systems with constructor systems. *Journal of Symbolic Computation 22,* 4, 381–399.

Sekar, R., Ramakrishnan, I. V., and Mishra, P. 1997. On the power and limitations of strictness analysis. *J. ACM 44,* 3, 505–525.

Sherman, D., Strandh, R. I., and Durand, I. 1991. Optimization of equational programs using partial evaluation. In Proceedings, 1st Symposium on Partial Evaluation and Semantics-Based Program Manipulation. *ACM SIGPLAN Not. 26,* 9, 72–82.

Staples, J. 1980. Computation on graph-like expressions. *Theor. Comput. Sci. 10,* 2, 171–185.

Steckler, P. and Wand, M. 1994. Selective thunkification. In *Proceedings, 1st Symposium on Static Analysis*, B. L. Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, 162–178.

Strandh, R. I. 1987. Optimizing equational programs. In *Proceedings, 2nd Conference on Rewriting Techniques and Applications*, P. Lescanne, Ed. Lecture Notes in Computer Science, vol. 256. Springer-Verlag, Berlin, 13–24.

Strandh, R. I. 1988. Compiling equational programs into efficient machine code. Ph.D. thesis, Johns Hopkins University.

Strandh, R. I. 1989. Classes of equational programs that compile efficiently into machine code. In *Proceedings, 3rd Conference on Rewriting Techniques and Applications*, N. Dershowitz, Ed. Lecture Notes in Computer Science, vol. 355. Springer-Verlag, Berlin, 449–461.

Thatte, S. R. 1985. On the correspondence between two classes of reduction systems. *Inf. Process. Lett. 20,* 2, 83–85.

Thatte, S. R. 1988. Implementing first-order rewriting with constructor systems. *Theor. Comput. Sci. 61,* 1, 83–92.

Turner, D. A. 1979. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9,* 1, 31–49.

Verma, R. M. 1995. Transformations and confluence for rewrite systems. *Theor. Comput. Sci. 152,* 2, 269–283.

Walters, H. R. 1997. Epic and ARM—user's guide. Technical Report SEN-R9724, CWI, Amsterdam. Report and tool set available at http://www.babelfish.nl.

Walters, H. R. and Kamperman, J. F. T. 1996. EPIC: An equational language—abstract machine and supporting tools. In *Proceedings, 7th Conference on Rewriting Techniques and Applications*, H. Ganzinger, Ed. Lecture Notes in Computer Science, vol. 1103. Springer-Verlag, Berlin, 424–427.

Wilson, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings, 1st Workshop on Memory Management*, Y. Bekkers and J. Cohen, Eds. Lecture Notes in Computer Science, vol. 637. Springer-Verlag, Berlin, 1–42.

Wray, S. C. and Fairbairn, J. 1989. Non-strict languages—programming and implementation. *Comput. J. 32,* 2, 142–151.

Received May 1998; revised September 1998; accepted July 1999