Centrum voor Wiskunde en Informatica
**REPORT***RAPPORT*

# Lazy rewriting on eager machinery

J.F.Th. Kamperman and H.R. Walters

Computer Science/Department of Software Technology

**CS-R9461 1994**

Report CS-R9461
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

# Lazy Rewriting on Eager Machinery

J.F.Th. Kamperman (jasper@cwi.nl)
H.R. Walters (pum@cwi.nl)

*CWI*
*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

**Abstract**

We define *Lazy Term Rewriting Systems* and show that they can be realized by local adaptations of an *eager* implementation of conventional term rewriting systems. The overhead of lazy evaluation is only incurred when lazy evaluation is actually performed.

Our method is modelled by a transformation of term rewriting systems, which concisely expresses the intricate interaction between pattern matching and lazy evaluation. The method easily extends to term *graph* rewriting.

*CR Subject Classification (1991):* D.3.4 [Programming languages]: Processors – Compilers, Optimization; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.6: Logic Programming.

*AMS Subject Classification (1991):* 68N20: Compilers and generators; 68Q05: Models of Computation; 68Q42: Rewriting Systems

*Keywords & Phrases:* lazy term rewriting, program transformation.

*Note:* Partial support received from the European Communities under the ESPRIT project 5399 (Compiler Generation for Parallel Machines – COMPARE).

## 1. Introduction

It is well-known that outermost rewriting strategies often have a better termination behaviour than innermost rewriting strategies [O'D77]. Innermost strategies (also called *eager evaluation*) can be implemented much more efficiently, however. We propose to solve this dilemma by transforming a term rewriting system (TRS) in such a way that the termination behaviour of innermost rewriting is improved. At the core of our transformation are established ideas of Ingermann [Ing61] and Plotkin [Plo75].

The bad termination behaviour of innermost rewriting is illustrated by the TRS in Figure 1 on page 2. In this system, the term $nth(0, inf(0))$ has an infinite reduction sequence $inf(0) \stackrel{(1)}{\rightarrow} cons(0, inf(s(0))) \stackrel{(1)}{\rightarrow} cons(0, cons(s(0), inf(s(s(0))))) \stackrel{(1)}{\rightarrow} \ldots$

This can be avoided by applying rule (1.1) only once, before applying rule (2): $nth(0, inf(0)) \stackrel{(1)}{\rightarrow} nth(0, cons(0, inf(s(0)))) \stackrel{(2)}{\rightarrow} 0$. By postponing some reductions, outermost rewriting may succeed in avoiding them altogether. Consider, however, the term $cons(0, inf(0))$, which does not terminate under an outermost reduction strategy.

$$
\begin{array}{lrcl}
(1.1) & inf(x) & \rightarrow & cons(x, inf(s(x))) \\
(1.2) & nth(0, cons(x, y)) & \rightarrow & x \\
(1.3) & nth(s(x), cons(y, z)) & \rightarrow & nth(x, z)
\end{array}
$$

Figure 1

Therefore, rather than simulating a pure outermost strategy, our transformation simulates a variant of *lazy* evaluation, which is used to implement *lazy functional programming languages* [PvE93]. We will briefly discuss this. During lazy evaluation, non-outermost redexes are contracted in order to establish that the outermost function symbol will never become part of a redex. The resulting term is said to be in Weak Head Normal Form (WHNF). E.g., the term $cons(0, inf(0))$ in the example above is in WHNF. The (implicit) routine which produces output for this term recursively causes reduction to WHNF of the arguments. The term $cons(0, inf(0))$ still leads to an infinite computation, but meaningful output is produced during this computation. Because rewriting of outermost redexes is expensive, it is usually avoided as much as possible. Arguments that can be rewritten eagerly without affecting termination behaviour, are called *strict*. Strictness analysis (initiated by Mycroft, [Myc80]) attempts to identify these arguments statically.

Now consider the TRS in Figure 2, which differs only slightly from the one in figure 1.

$$
\begin{array}{lrcl}
(1.4) & inf(x) & \rightarrow & cons(x, thunk(x)) \\
(1.5) & inst(thunk(x)) & \rightarrow & inf(s(x)) \\
(1.6) & nth(0, cons(x, y)) & \rightarrow & x \\
(1.7) & nth(s(x), cons(y, z)) & \rightarrow & nth(x, inst(z))
\end{array}
$$

Figure 2

The term $nth(0, inf(0))$ still rewrites to 0, but there are no infinite reduction sequences. This example illustrates that only minor changes are needed to achieve the desired effect, and that these changes can be made local to "lazy positions" (cf. the second argument of *cons*). To some extent, this explains the success of strictness analysis. In many cases, only a few positions need a lazy treatment in order to preserve termination.

The example also demonstrates the common observation that a good implementation of a lazy language spends most time in "eager mode". Given the locality of the changes above, it is worthwhile to investigate how an *eager* implementation can be adapted to do some *lazy* evaluation, rather than adapting a *lazy* implementation to do (a lot of) eager evaluation.

We use laziness annotations to indicate argument positions where rewriting should be postponed if possible. These annotations could be provided by the programmer or by a strictness analyzer. In the latter case, all arguments that are not found to be

strict, will get the annotation *lazy*, and the reductions performed by our implementation will correspond closely to the reductions performed by an implementation of a lazy functional programming language using the same strictness analyzer.

Even though Figure 2 is a simplified version of the result of our transformation, applied to the TRS of Figure 1 (with only the second argument of *cons* annotated with *lazy*), it exhibits a peculiarity of our scheme. The term $inf(0)$ rewrites to the normal form $cons(0, thunk(0))$, which is not a term in the original signature. However, the term $thunk(0)$ (called a "thunk" after Ingermann [Ing61]) represents a (possibly infinite) term in the original system, which can be further approximated by repeatedly replacing terms $thunk(x)$ by the normal form of $inst(thunk(x))$. Our lazy normal forms (LNFs) generalize the notion of WHNF, and the approximation process corresponds to what is done by the output routine of an implementation of a lazy functional language. We do not assume such an output routine, because leaving the thunk in place offers the possibility of preventing uninteresting work, and yields a larger class of terminating systems.

We give definitions and notations pertaining to term (graph) rewriting in Section 2, our definition of *lazy* term rewriting in Section 3, and a complete version of the transformation sketched above in Section 4. We make some remarks on a realistic implementation in Section 5. We end with a discussion of related work and conclusions.

## 2. Term (graph) rewriting

We mostly repeat definitions and results from [Klo92] and [DJ90].

A Term Rewriting System (TRS) is a pair $(\Sigma, R)$ of a *signature* $\Sigma$ and a set of *rewrite rules* $R$. The signature $\Sigma$ consists of:

- A countably infinite set $Var$ of *variables* $x_1, x_2, \ldots$
- A non-empty set of *function symbols* $F, G, \ldots$, each with an *arity*, which is the number of arguments it requires. Function symbols with arity 0 are called *constant symbols*.

The set of terms over $\Sigma$ is the smallest set $Ter(\Sigma)$ such that

- $x_1, x_2, \ldots \in Ter(\Sigma)$,
- if $F$ is an $n$-ary function symbol and $t_1, \ldots t_n \in Ter(\Sigma)$ $(n \geq 0)$, then $F(t_1, \ldots, t_n) \in Ter(\Sigma)$. The $t_i$ $(i = 1, \ldots, n)$ are called the *arguments*.

Terms in which no variable occurs more than once are called *linear*.

A path in a term is represented as a sequence of positive integers. By $t|_p$, we denote the *subterm* of $t$ at path $p$. For example, if $t = push(0, pop(push(y, z)))$, then $t|_{2.1}$ is the first subterm of $t$'s second subterm, which is $push(y, z)$. We will say $p \in t$ if the path $p$ is defined in $t$, i.e., $p$ leads to a subterm of $t$. The empty

path (denoting the root) is written $\varepsilon$. We will call a set of paths $P$ *prefix-reduced* if there are no pairs $p, p' \in P$ such that $p$ is a prefix of $p'$. We will call a set $S$ of subterms of $s$ *prefix-reduced with respect to* $s$ if there is a *prefix-reduced* set of paths $\{p_1, \ldots, p_n\}$ such that $S = \{s|_{p_1}, \ldots, s|_{p_n}\}$. We write $t[s]_p$ for the term resulting from the replacement of $t|_p$ by $s$ in $t$.

A *substitution* $\sigma$ is a map from $Ter(\Sigma)$ to $Ter(\Sigma)$ satisfying $\sigma(F(t_1, \ldots, t_n)) = F(\sigma(t_1), \ldots, \sigma(t_n))$ for every function symbol $F$. By convention, we write $t^\sigma$ instead of $\sigma(t)$.

A *rewrite rule* is a pair $(t, s)$ of terms $\in Ter(\Sigma)$. It will be written as $t \to s$. Often a rewrite rule will get a name, e.g. $r$, and we write $r : t \to s$. Two conditions are imposed:

- the LHS (left-hand side) $t$ is not a variable,
- the variables in the RHS (right-hand side) $s$ already occur in $t$.

A rewrite rule $r : t \to s$ determines a relation: the set of *rewrites* $t^\sigma \to_r s^\sigma$ for all substitutions $\sigma$. The LHS $t^\sigma$ is called a *redex* (from "reducible expression"), and the RHS $s^\sigma$ is called the *contractum*. Allowing replacement inside other terms, $\to_r$, the *one-step rewrite relation* generated by $r$, is defined by:

$$u|_p = t^\sigma \;\Rightarrow\; u \to_r u[s^\sigma]_p$$

We call the relation $\to_R = \cup_{r \in R} \to_r$ the *rewrite relation* defined by $R$. Usually, the subscript $R$ is omitted if it is clear from the context. Concatenating rewrite steps we have (possibly infinite) *rewrite sequences* $t_0 \to t_1 \to \ldots$ or *rewrites* for short. If $t_0 \to \ldots \to t_n$ $(n \geq 0)$, we also write $t_0 \stackrel{*}{\to} t_n$. If $t_0 \to \ldots \to t_n$ $(n \geq 1)$, we also write $t_0 \stackrel{+}{\to} t_n$ and call $t_n$ a *reduct* of $t_0$. A term $t \in Ter(\Sigma)$ is said to be in *normal form* if there is no $s$ such that $t \stackrel{+}{\to}_R s$. It is understood that $R$ does not contain rewrite rules that are equal up to an bijective renaming of variables.

A TRS is called *left-linear* if all left-hand sides are linear. A TRS is called *confluent* if, for all terms $t_1, t_2, t_3$, we have that $t_1 \stackrel{*}{\to} t_2$ and $t_1 \stackrel{*}{\to} t_3$ implies that there exists a term $t_4$ such that $t_2 \stackrel{*}{\to} t_4$ and $t_3 \stackrel{*}{\to} t_4$. A TRS is called *terminating* if there are no infinite rewrite sequences. In the sequel, we will only consider left-linear, confluent TRSs. However, we will not require TRSs to be terminating. Note that it is undecidable whether a TRS is confluent or terminating.

In general, a term can contain many redexes. In an implementation of a TRS, a rewriting *strategy* determines which of the many possible rewrite sequences is chosen. Confluence guarantees unique normal forms.

In this article, we will assume the existence of an implementation of the *leftmost-innermost strategy* (LI: the leftmost-innermost redex takes precedence). By a transformation, we will simulate lazy evaluation.

A typical implementation of an LI strategy for TRSs is given in [Heu88], where the rules are compiled into a Lisp function. The body of this function consists of

pattern matching code that determines which code is used for instantiation of the RHS. The former code is produced by a pattern matching compiler, the latter code is typically a number of nested function calls, with references to terms as arguments. On many architectures, this type of recursive code performs badly, which leads to several alternatives [TAL90, KW93, Bak94].

Term *graph* rewriting [BvEJ+87], where terms and rules are replaced by graphs, can be seen as a restriction of rewriting with infinite terms [KKSdV93]. An implementation of term rewriting can be turned into an implementation of graph rewriting by taking care that the sharing expressed by graphs is retained. Note that, in general, this is not easy.

## 3. Lazy term rewriting

We define *lazy term rewriting* as term rewriting with a restriction on the (one-step) rewrite relation. First, we define *lazy* signatures, which make a distinction between *eager* argument positions and *lazy* argument positions.

The choice to annotate the *arguments* rather than the *function symbols* themselves is not only motivated by compatibility with lazy functional languages, but has two additional disadvantages. First, if functions are annotated, we must expect thunks at every argument position, thus losing the locality of our transformation. Second, for functions such as `if(Bool,Exp,Exp)`, it is more natural to annotate an argument position than to annotate all function symbols that may occur there. Unfortunately, not all TRSs can be made terminating by only annotating arguments (cf. the rule $inf(x) = inf(x)$).

A lazy signature includes a predicate $\Lambda$ on function symbols and natural numbers, where $\Lambda(F, i) = \mathit{true}$ means that the $i$th argument position of $F$ ($0 \leq i \leq \mathit{arity}(F)$) is lazy, and $\Lambda(F, i) = \mathit{false}$ means that it is eager. As an abbreviation, we write $F(!, ?)$ for a function $F$ of arity 2, the first argument of which is eager and the second argument of which is lazy.

This notion is easily extended to paths in terms:

**Definition**

- *For all terms $t$, $\varepsilon$ is an* eager path *in $t$.*
- *If $p$ is an eager path in $t$ and $t|_p = F(t_1, \ldots, t_n)$ with $\neg\Lambda(F, i)$ for some $i$ ($1 \leq i \leq n$) then $p.i$ is eager.*
- *All other paths are* lazy.

In other words, a path is *eager* precisely if it passes through eager arguments only. A lazy path $p$ is called *directly lazy* if $p = p'.i$ with $p'$ eager. For example, given the signature

$$\{cons(!, ?), bin(!, !)\}$$

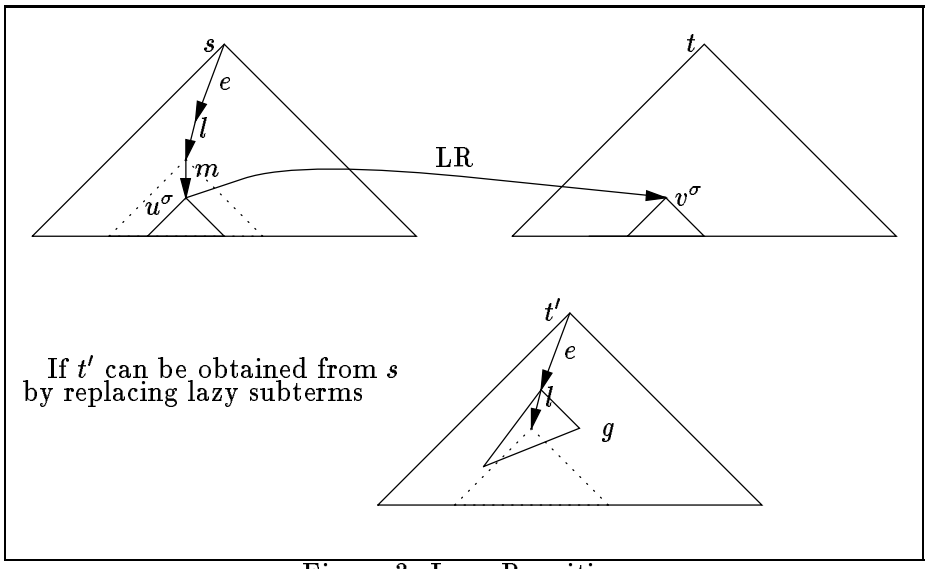


Figure 3: Lazy Rewriting

and the terms $t_1 = cons(x, cons(y, z))$ and $t_2 = \text{bin}(cons(x, y), cons(x, z))$, the paths 1 in $t_1$ and $1, 1.1, 2, 2.1$ in $t_2$ are eager; $2.1, 2.2$ in $t_1$ and $1.2, 2.2$ in $t_2$ are lazy, of which only 2.1 and 2.2 in $t_1$ are not also directly lazy. With $Lazy(t)$, we will denote the prefix reduced set of lazy paths in $t$, and a subterm at a lazy path will be called a *lazy subterm*.

With $\zeta(t)$, we will denote the term obtained by replacing every lazy subterm of $t$ with the unique constant $\zeta$. For any normal form $n$, $\zeta(n)$ is exactly the part we are interested in. We will say that terms $t_1$ and $t_2$ are $\zeta$-equal, or equal *up to* $\zeta$, when $\zeta(t_1) = \zeta(t_2)$.

Ideally, we would like to rewrite a lazy subterm at path $p$ only if this is *needed* to establish a *needed* redex at an eager prefix $e$ of $p$. Then, the termination behaviour of lazy rewriting would be at least as good as the termination behaviour of rewriting only needed redexes.

If there are overlapping LHSs, however, the notion of needed redex cannot be defined. Therefore, we give a weaker definition, which only requires that a redex at an eager prefix of $p$ can be established by replacing lazy subterms. The ideal of *needed* rewriting can be approximated by demanding a particular relation between the lazy subterms and their replacements. We will not try to achieve this, because most interesting relations seem to be either undecidable or hard to implement or have such a large bias towards a particular strategy that they are unnatural as a restriction on the rewrite relation. Instead, we try to make the restriction on the rewrite relation as weak as possible, by considering only LHSs and outermost lazy positions. The rewrite *strategy* is expected to approximate the ideal by avoiding as much rewrites at lazy paths as reasonably possible. The transformation presented in section 4 implements such a strategy.

We will first present our definition informally, using Figure 3 as illustration. Let

the lazy path $p$ consist of an eager path $e$, a lazy path $l$, and a path $m$, which may be either eager or lazy. We allow rewriting at $p = e.l.m$ in $t$ only if at the eager prefix $e$, a redex $g^\tau$ can be established by replacing some lazy subterms of $t$, such that the nonvariable part of $g$ (shown as a triangle labeled with $g$) overlaps with $l$. The endpoints of lazy paths where rewriting is allowed, are indicated by a dotted triangle. The actual rewrite at $e.l.m$ is indicated by an arrow annotated with "LR".

Formally, this is described by the following definition:

**Definition** *$s$ rewrites* lazily *to $t$, written $s \overset{LR}{\rightarrow} t$, if $\exists u \rightarrow v \in R, \sigma, p$ such that*

- $s|_p = u^\sigma$
- $t = s[v^\sigma]_p$
- – *$p$ is eager in $s$, or*
  – *$p = e.l.m$, where $e$ is eager in $s$, $e.l$ is lazy in $s$, and $\exists p_1, \ldots, p_n \in Lazy(s), r_1, \ldots, r_n, t' \in Ter(\Sigma), g \rightarrow h \in R, \tau$ such that $t' = s[r_1]_{p_1} \ldots [r_n]_{p_n}$, $t'|_e = g^\tau$ and $g|_l \notin Var$.*

This restriction of the one-step rewrite relation yields an extended class of normal forms. We will call these *lazy normal forms (LNF)*. For instance, given the TRS of Figure 1, if $\Lambda(cons, 2) = true$, then $cons(0, inf(0))$ is an LNF which is not a normal form. If $\Lambda(f, i)$ is *true* for all $f, i$ LNF coincides with WHNF. If $t$ is an LNF, we call $\zeta(t)$ a $\zeta$-LNF.

Because $\overset{LR}{\rightarrow}$ is a restriction of $\rightarrow$, it follows easily that termination is preserved. We have that lazy rewriting is both correct and complete in the following sense:

**Theorem 1** *(Completeness) For all normal forms $s$ of $t$, there is a $\zeta$-equal LNF $s'$.*

**Proof sketch** $s'$ can be constructed from the rewriting sequence $t_1 \rightarrow t_2 \ldots \rightarrow s$ by directly performing the rewrites that are allowed, and maintaining a residual map [O'D77] of the paths where rewriting is not allowed. When a path where rewriting is forbidden is mapped to a context which is either eager, or may turn into a redex by replacing lazy subterms, the suspended rewrite is also performed. Thus, all non-preformed rewrites pertain either to a term that is deleted, or are mapped (by the residual map) to a lazy subterm in $s'$. Therefore, $\zeta(s') = \zeta(s)$ ■

**Theorem 2** *(Correctness) If $t$ is an LNF, then for all normal forms $s$ of $t$, $\zeta(s) = \zeta(t)$.*

**Proof** From the definition, it follows that there are no redexes at eager paths, and no lazy path leading to a redex has an eager prefix which may become a redex by replacing lazy subterms. Therefore, all eager paths in an LNF are stable ■

**Corollary 1** *If there is a unique normal form $t$ of $s$, then all LNFs of $s$ are $\zeta$-equal to $t$.*

From the fact that an arbitrary number of "irrelevant" rewrite steps can in general be performed before the rewrite that turns a term into an LNF, it follows that confluence is not preserved. However, given the fact that we are only really interested in $\zeta(n)$ for any LNF $n$, it is fair to consider only $\zeta$-*confluence*:

**Definition** *A TRS $R$ is* $\zeta$-confluent *if for every $t_1$, $t_2$ and $t_3$, if $t_1 \to t_2$ and $t_1 \to t_3$ then there are terms $t_4, t_5$, such that $t_2 \xrightarrow{*} t_4, t_3 \xrightarrow{*} t_5$ and $\zeta(t_4) = \zeta(t_5)$.*

**Theorem 3** *Lazy rewriting preserves $\zeta$-confluence.*

**Proof** Suppose $R$ is $\zeta$-confluent, and let $t_1, t_2$ and $t_3$ be such that $t_1 \xrightarrow{LR} t_2$ and $t_1 \xrightarrow{LR} t_3$. Then there are $\zeta$-equal terms $t_4, t_5$, and rewriting sequences $s_1 : t_2 \xrightarrow{*} t_4$ and $s_2 : t_3 \xrightarrow{*} t_5$. If at some term $t'$ from the sequence $s_1$, a rewrite at a (lazy) path $p$ is forbidden by lazy rewriting, then no redex can later occur at an eager path above $p$. Therefore, at all eager paths above $p$, $t'$ has the same function symbol as $t_4$. We can thus skip the forbidden rewrite and all rewrites that occur below $p$, because they only affect subterms that do not make a difference from the viewpoint of $\zeta$-equality. Repeating our reasoning for all other forbidden rewrites, we arrive at a term that is $\zeta$-equal to $t_4$. Similarly for $s_2$ ■

Of course, $\zeta$-confluence implies uniqueness up to $\zeta$ of LNFs.

## 4. A TRANSFORMATION TO ACHIEVE LAZINESS

We will specify a transformation $\mathcal{L}$ from TRSs to TRSs and a transformation $\mathcal{T}$ from terms to terms, such that when $\mathcal{T}(t)$ is rewritten by an innermost strategy in $\mathcal{L}(R)$ to a normal form $n$, then $\zeta(n)$ is the $\zeta$-LNF of $t$ with respect to $R$. The transformed system avoids rewriting lazy subterms to a large extent (optimal avoidance is impossible in nonorthogonal TRSs). Basically, the transformation $\mathcal{T}$ replaces all lazy subterms of an input term by stable terms (*thunks*), and $\mathcal{L}$ adds rules for "unthunking" both input thunks and thunks that encode right-hand sides. Furthermore, $\mathcal{L}$ ensures that

- Lazy subterms of right-hand sides are thunked, so only stable terms are introduced at lazy paths.
- When a subterm (matched to a variable) is moved from a lazy *lhs* position into an eager *rhs* position, it is unthunked, so thunks only occur at lazy positions.
- A lazy argument is unthunked before a match overlapping with it is rejected.

We start with some definitions. A *thunk* is a term with a special function symbol $\delta$ at the top, a name of a pattern ($p$) as first argument, and a tuple of terms (denoted by $\mathbf{vec}_n(t_1, \ldots, t_n)$) as second argument:

$$\delta(p, \mathbf{vec}_n(t_1, \ldots, t_n))$$

Given a rule $s \to t$, we call a variable *migrating* if it occurs at a *directly* lazy position in $s$ and at some eager position in $t$. Because we want to keep the effect of our transformation local, rules must be added that "unthunk" migrating variables.

*4.1 The transformation $\mathcal{L}$*

$\mathcal{L}$ takes a TRS $(\Sigma, R)$, and transforms it into a system $(\Sigma \cup N \cup A, RG \cup RI \cup R')$. In the transformed system,

- $N$ is a countably infinite set of function symbols that do not occur in $\Sigma$ (they are used in thunks as names of patterns). There is a set $T \subset N$ of "tokens", such that for every function symbol $f$ in $\Sigma$, we have a unique $t_f \in T$,
- $A$ is a set of "administrative" function symbols

  $$\{\delta(!,!), \delta?(!), \mathbf{inst}(!), \pi(!,!), \mathbf{true}\} \cup_{m,n \in Nat} \{\mathbf{vec}_{mn}(l_{m1}, \ldots, l_{mn})\},$$

  where $\delta$ will be used as the top symbol of a thunk, $\delta?$ is a predicate that recognizes thunks, a function $\mathbf{vec}_{mn}(l_{m1} \ldots l_{mn})$ is used to "pack" $n$ variables in a thunk ($m$ encodes the laziness annotations: the $l_{mi}$ are either ! or ?. Most of the time, $m$ will be omitted). Finally, $\pi$ is a projection function that makes implementation of *graph rewriting* easy, which will be discussed in Section 5.1.
- $RG$ contains the general rules defining the projection $\pi$ and the thunk-recognizer $\delta?$:

  $$\pi(x, y) \to y$$

  $$\delta?(\delta(x, y)) \to \mathbf{true}$$

- $RI$ contains the rules describing selective unthunking of input terms. For every $f$ with arity $n$, of which $k$ are eager positions (with indices $e_1, \ldots, e_k$), $RI$ contains the rules (with $c_{f_i} \in N$):

$$\begin{array}{lrcl}
(4.8) & \mathbf{inst}(\delta(t_f, \mathbf{vec}_n(x_1, \ldots, x_n))) & \to & c_{f_1}(\delta?(x_{e_1}), x_1, \ldots, x_n) \\
(4.9) & c_{f_1}(\mathbf{true}, x_1, \ldots, x_n) & \to & c_{f_2}(\delta?(x_{e_2}), x_1, \ldots, \mathbf{inst}(x_{e_1}), \ldots, x_n) \\
(4.10) & c_{f_1}(\delta?(y), x_{e_1}, \ldots, x_n) & \to & c_{f_2}(\delta?(x_{e_2}), x_1, \ldots, x_{e_1}, \ldots, x_n) \\
 & & \ldots & \\
(4.11) & c_{f_k}(\mathbf{true}, x_1, \ldots, x_n) & \to & f(x_1, \ldots, inst(x_{e_k}), \ldots, x_n) \\
(4.12) & c_{f_k}(\delta?(y), x_1, \ldots, x_n) & \to & f(x_1, \ldots, x_{e_k}, \ldots, x_n)
\end{array}$$

  Here, (4.8) starts the instantiation of a delayed term with function symbol $f$, (4.9,4.11) handle the case that an argument ($x_{e_1}$ and $x_{e_k}$, respectively) is still thunked and (4.10,4.12) handle the case that an argument is already unthunked. Note that the distinction between thunked and unthunked arguments relies on the partial function $\delta?$ being evaluated eagerly.
- The rules in $R'$ are obtained by applying the three transformations below (RHS for thunk introduction, LR for left-right unthunking and LS for left-hand side matching) to $R$ as follows: RHS until fixpoint, LR once for every equation in the fixpoint, LS once for every equation in the result of LR.

**RHS (Thunk Introduction)** This transformation is applicable to all rules $r : s \to t$ where $t$ contains a directly lazy path $p$, such that $t|_p$ is neither a variable, nor a subterm already occurring in $s$, nor a thunk. Let $\{t_1, \ldots, t_n\}$ be the set of terms occurring in both $s$ and $t|_p$, and prefix-reduced with respect to $t$, then $r$ is *replaced* by two rules ($i$ unique in $N$):

$$\begin{aligned} s &\to t[\delta(i, \mathbf{vec}_n(t_1, \ldots, t_n))]_p \\ \mathbf{inst}(\delta(i, \mathbf{vec}_n(x_1, \ldots, x_n))) &\to \pi(\delta(i, \mathbf{vec}_n(x_1, \ldots, x_n)), t[t_i/x_i]) \end{aligned}$$

Here $\Lambda(\mathbf{vec}_n, i) =?$ if and only if $t_i$ is a variable occuring at a directly lazy context in $s$.

**LR (Migrating Thunk Elimination)** This transformation applies to rules $r : s \to t$ containing *migrating* variables. Supposing $\{t_1, \ldots, t_n\}$ is a set of subterms which occur both in $s$ and $t$, and which is prefix-reduced with respect to $t$, and let $e_1, \ldots, e_k$ be the indices of the *migrating* variables, then $r$ is replaced by the following rules, similar in form and intent to the rules in $RI$:

$$\begin{aligned} s &\to c_{i_1}(\delta?(x_{e_1}), t_1, \ldots, t_n) \\ c_{i_1}(\mathbf{true}, x_1, \ldots, x_n) &\to c_{i_2}(\delta?(x_{e_2}), x_1, \ldots, \mathbf{inst}(x_{e_1}), \ldots, x_n) \\ c_{i_1}(\delta?(y), x_1, \ldots, x_n) &\to c_{i_2}(\delta?(x_{e_2}), x_1, \ldots, x_{e_1}, \ldots, x_n) \\ &\ldots \\ c_{i_k}(\mathbf{true}, x_1, \ldots, x_n) &\to c_{i_k+1}(x_1, \ldots, \mathbf{inst}(x_{e_k}), \ldots, x_n) \\ c_{i_k}(\delta?(y), x_1, \ldots, x_n) &\to c_{i_k+1}(x_1, \ldots, x_{e_k}, \ldots, x_n) \\ c_{i_k+1}(x_1, \ldots, x_n) &\to t[t_i/x_i] \end{aligned}$$

**LS (Matching Thunk Elimination)** This transformation is applicable to rules $r : s \to t$ if $s$ contains nonvariable lazy positions. For every element $i = \{i_1, \ldots, i_n\}$ in the prefix-reduced powerset of lazy paths in $s$, add a rule (all $n_j$ and $v_j$ fresh):

$$s[\delta(n_1, v_1)]_{i_1} \ldots [\delta(n_n, v_n)]_{i_n}$$

$$\to s[\pi(\delta(n_1, v_1), \mathbf{inst}(n_1, v_1))]_{i_1}[\delta(n_2, v_2)]_{i_2} \ldots [\delta(n_n, v_n)]_{i_n}$$

*4.2 The transformation $\mathcal{T}$*

$\mathcal{T}$ thunks all non-variable lazy subterms of the original input term, by the token of their outermost function symbol and their thunked arguments.

$$\begin{aligned} \mathcal{T}[f(t_1, \ldots, t_n)] &= f(t'_1, \ldots, t'_n) \quad (\text{where } t'_i = \mathcal{T}[t_i] \text{ iff } \Lambda(f, i) =!, \text{ otherwise } t'_i = \mathcal{T}_l[t_i]) \\ \mathcal{T}_l[f(t_1, \ldots, t_n)] &= \delta(t_f, \mathbf{vec}_n(\mathcal{T}_l[t_1], \ldots, \mathcal{T}_l[t_n])) \\ \mathcal{T}[x] &= \mathcal{T}_l[x] = x \end{aligned}$$

*4.3 Correctness and completeness of the transformation*
First, we remark that the transformation itself terminates, because every application of RHS replaces one (non-thunked) lazy argument by a thunk, and LR and LHS terminate trivially.

**Theorem 4** *(Correctness of $\mathcal{L}$ and $\mathcal{T}$) Given a TRS $R$ and a term $t$, every step in an innermost rewriting of $\mathcal{T}(t)$ in $\mathcal{L}(R)$ is either an administrative step (checking if an argument is a thunk), or it corresponds to a legal step in $\overset{LR}{\rightarrow}_R$.*

**Proof** Note that for all terms $t$, $\mathcal{T}(t)$ has only $R$-redexes above lazy positions, because all lazy subterms are thunked by $\mathcal{T}_l$. By RHS, all rules have been transformed into rules that put stable terms at lazy paths, and LR preserves this property. The only redexes at lazy paths are $\mathcal{L}(R)$-redexes, introduced by LS, but the conditions for application of LS imply that there is a nonvariable $R$-pattern overlapping with the hole in which the redex is introduced, so the condition for lazy rewriting is fullfilled ■

**Theorem 5** *(Completeness of $\mathcal{L}$ and $\mathcal{T}$) Given a TRS $R$ and a term $t$, the normal form $t_n$ of $t$ with respect to $\mathcal{L}(R)$ (if it exists) is $\zeta$-equal to some LNF $t_l$ of $t$.*

**Proof** Suppose that $\zeta(t_n) \neq \zeta(t_l)$. Because of correctness, we have that $t \overset{LR}{\rightarrow} t'_n$, where $t'_n$ is obtained from $t_n$ by replacing thunks with the RHSs they represent. Because $t_l$ is a LNF, we have that $t \overset{LR}{\rightarrow} t_l$. Lazy rewriting preserves $\zeta$-confluence, so $t'_n$ cannot be a lazy normal form. This means that $t_n$ must either contain a normal redex, or an administrative redex (because $t_n$ only differs from $t'_n$ by having thunks at lazy paths, and LS introduces rules that remove any thunk which blocks matching of a LHS) ■

## 5. From transformation to implementation

The transformation in Section 4 is useful both as a tool for experimentation, and as a concise model of an implementation of lazy rewriting. To obtain an implementation that can compete with special-purpose lazy implementations such as TIM ([FW87]) or the Spineless Tagless G-machine (STG, [JS89]), some details have to be changed.

First, in order to prevent multiple reductions of the same term, the TRS should be interpreted as a *graph* rewriting system. We give details on this in Section 5.1.

Second, some glaring inefficiency is caused by the LS transformation. This can be overcome by simulating the effect of LS in the pattern-matching code, which is explained in Section 5.2.

*5.1 Graph rewriting by adding sharing*
By the following modifications, the advantages of graph rewriting are incorporated:

- In the implementation of $\mathcal{T}$, sharing should be retained.

- The function $\pi(!,!)$ is implemented such, that it overwrites its first argument (always a thunk) with the LNF of its second argument (always the LNF corresponding with the thunk). Note that this requires a fixed node size, or some other means to avoid overwriting smaller with bigger nodes.
- If a subterm occurs both in LHS and RHS of some rule, no copy should be made. Then it follows from the construction of the transformed system, that thunks are never duplicated, so every thunk is only evaluated once.
- For cyclic graphs, the code that is generated for the construction of a right-hand side must be modified slightly. Without loss of generality, we consider a prototypical RHS $x : f(\ldots, x, \ldots)$. For this RHS, the compiler should emit code corresponding to $\mathbf{inst}(T)$, where $T$ is a thunk for $f(\ldots, T, \ldots)$. Note that this requires that the "address" of a node under construction is available during the construction.

*5.2 Optimizations*

When implemented naively, our transformation has a large impact on the number of equations. A worst-case analysis shows that the maximal number of additional equations is

$$3 + n.r + n.2^l + 2s$$

where $n$ is the number of rules, $r$ is the maximal number of nonvariable lazy positions in a RHS, $l$ is the maximal number of nonvariable lazy positions in a LHS, and $s$ is the number of lazy positions in the signature. It should be noted that, measured in function symbols, the rules added by $RHS$ are compensated for by a reduction in size of the original rule, and $s$ is generally small compared to $n$.

Thus, the only dangerous term is the exponential term in $l$, caused by the powerset construction in transformation $LS$. We will illustrate both the problem and its solution with an example. Assuming we have a signature $\{a, b, i(!), t(?,?)\}$ and a rule $i(t(a,b)) \to a$, then $LS$ adds the rules

$$\begin{aligned}
i(t(\delta(p, \mathbf{vec}_0), b)) &\to i(t(\pi(\delta(p, \mathbf{vec}_0), \mathbf{inst}(\delta(p, \mathbf{vec}_0))), b)) \\
i(t(\delta(p, \mathbf{vec}_0), \delta(p', \mathbf{vec}_0))) &\to i(t(\pi(\delta(p, \mathbf{vec}_0), \mathbf{inst}(\delta(p, \mathbf{vec}_0))), \delta(p', \mathbf{vec}_0))) \\
i(t(a, \delta(p, \mathbf{vec}_0))) &\to i(t(a, \pi(\delta(p, \mathbf{vec}_0), \mathbf{inst}(\delta(p, \mathbf{vec}_0)))))
\end{aligned}$$

When a term $i(t(x,y))$ is rewritten, where both $x$ and $y$ are thunks which will instantiate to $a$ and $b$ respectively, this leads to the following inefficiencies:

- $i$ and $t$ are matched 3 times (2 times to discover the thunks, and the last time to find the original match),
- the function symbol $t$ is copied 2 times, because the subterm from the LHS cannot be reused.

This can be repaired by changing the pattern matching code to instantiate the thunks, such that the rules introduced by $LS$ are no longer needed (even though they give a nice model of what is happening). In pseudo code, the modified code reads as follows:

```
case x of
i(y): case y of
   t(z1,z2): case z1 of
      a: label1: case z2 of
                 b: continue(a) /* matched ! */
                 thunk: inst(z2); goto label1
                 otherwise: return(x) /* normal form */
      thunk: label2: case z2 of
                     b: inst(z1)
                        case z1 of
                        a) continue(a) /* matched ! */
                        otherwise: return(x) /* normal form */
                     thunk: inst(z2); goto label2
                     otherwise: return(x) /* normal form */
      otherwise: return(x) /* normal form */
   otherwise: ...
otherwise: ...
```

This pattern matching code is bigger than the code for the single rule in the original system, but it is a very efficient implementation of lazy pattern matching, because it only does extra work (compared to the eager implementation) if an unevaluated thunk is encountered during matching.

The implementation can be further improved by implementing $\delta$ as a tag-bit, $\delta?$ as a bit-test, and **inst** and $\mathbf{vec}_n$ as built-in functions. Finally, the effect of the LR transformation can be achieved by generating slightly different code for right-hand sides.

## 6. Related work

A very early related paper is [Plo75], which gives simulations of call-by-name by call-by-value (eager evaluation), and vice versa, in the context of the $\lambda$-calculus. Call-by-name evaluation differs from lazy evaluation (or call-by-need): Thunks are not overwritten with the result of evaluation, but evaluated on every use (which is essential in a language with side-effects).

In the context of functional programs, [Amt93] developed an algorithm to transform call-by-name programs into call-by-value equivalents. In [SW94], dataflow analysis is done in order to minimize thunkification in this context.

In [OLT94], a continuation passing style (cps) transformation of call-by-need into call-by-value equivalents is given. To their knowledge, it is the first. Apart from the fact that a particular $\lambda$-calculus is only one instance of a TRS, our transformation differs mainly by completely integrating pattern matching of algebraic datatypes in the transformation. It is unclear how much can be gained by taking pattern matching into account in a transformation for a lazy functional implementation. An abstract

approach to strictness analysis of algebraic datatypes is investigated in [Ben93]. We noted that the built-in pattern-matching (*case*) and conditional constructs (*if*) in many lazy languages are often unnecessarily assumed to be strict (cf. [Bur91]).

The effect of our transformations of rewrite systems is somewhat similar in spirit to the use of evaluation transformers in [Bur91]. Not only in theory, but also in practice, our technique does not rely on properties of built-in algebraic datatypes such as lists or trees. In [BM92], some of the techniques in [Bur91] are formulated in the context of continuation passing transformations.

Another approach to obtain better termination properties are the sequential strategies investigated by [HL91, O'D77]. In this approach, only *needed* redexes are rewritten, i.e., redexes that would be rewritten in any reduction to a normal form. Unfortunately, *neededness* is only well-defined in TRSs that do not have overlapping redexes. This restriction is hard to live with in practice.

To our knowledge, only the Clean [PvE93] and the OBJ3 [GWM+92] systems support laziness annotations. Clean supports the annotation of *strict* arguments, OBJ3 features annotations for the evaluation order of arguments which are somewhat more explicit than ours. It appears that a similar transformation can implement OBJ's annotations.

A rule occuring in the context of an $E$-unification algorithm, presented in [MMR86], is called "lazy rewriting" in [Klo92]. It might be interesting to investigate whether our technique of implementing lazy rewriting on eager machinery is useful in that context.

In CAML (Categorial ML, [CH90]) there are lazy constructors, which can be used to achieve effects similar to our transformation. However, the transformation of the program must then be carried out manually for the most part (only equivalents of **inst**, $\delta?$ and $\delta$ are supplied by the implementation).

It is obvious, that our last implementation of lazy term rewriting is similar to the implementation of modern lazy functional languages. As far as we know, these implementations are completely lazy by nature, but are optimized to perform as much eager evaluation as possible.

Therefore, it is appropriate to provide a discussion of the cost of basic datastructures and actions in our scheme, compared with the cost in those implementations. It should be noted that it is *extremely* difficult ([JS89]) to assess the effect of different design choices on performance, so we will only give a qualitative discussion.

- Only a little structure ($\delta$, a thunk constant and a vector containing references to subterms from the left-hand side) occurs below a lazy position in any rhs after the transformation. This is comparable to the frames used in TIM [FW87], or the closures in the STG. Similarly to the latter, our scheme only uses space for the subterms from the LHS that may actually be used later. In the ABC machine [PvE93], complete graphs are built for lazy arguments, which is a drawback compared to all other implementations.

- No runtime cost is incurred when all arguments in the original TRS are annotated eager. Even when all arguments are found to be strict, TIM and STG do a function call to obtain the tag of a constructor term (this is the reason they are called "tagless"), whereas our implementation only needs to dereference a pointer.

- There is no need for the dreaded indirection nodes ([O'D77, JL92], because $\delta$ fullfills this role; every term (input or rhs) is evaluated exactly once, either by immediate innermost rewriting, or later, by overwriting the $\delta$ node. In [JL92], the indirection nodes are also transformed away, but some very complicated analysis is needed to arrive at this result. In the ABC machine, the indirection nodes are indispensable.

- In the rules added by transformation LR, testing if a lazy argument is thunked, is done by rewriting. Even if this is replaced by a bit-test implementation, a subsequent call of **inst** must be done. This is less efficient than the "tagless" reduction which is done in both TIM and STG.

- Unthunking is only done if all eager pattern matching was succesfull. Because the order of pattern matching and its effects on evaluation of subterms are fixed in the semantics of lazy functional languages, this cannot be done in the other implementations. Usually, the interaction between pattern matching of algebraic datatypes and lazy evaluation is not incorporated in strictness analysis.

Taking into account these points, we expect our scheme to perform better than ABC, TIM or the STG, when there is a small number of lazy arguments.

In contrast with common opinion, we hold that laziness annotations provided by the programmer are a suitable way of indicating lazy evaluation. We observe that it is reasonable to require of a lazy functional programmer to make sure that his program terminates. This requires a thorough understanding of both the program and the operational semantics of the language. In our experience, this level of understanding is adequate to provide complete laziness annotations. The *truly lazy* programmer will of course use a strictness analyzer to assist in the process of understanding his program.

We certainly do not want to imply that our scheme renders strictness analyzers superfluous. Fine-grain strictness analysis can even be used to improve the result of our transformation.

## 7. Conclusions and acknowledgements

We have defined *lazy rewriting* and have generalized the notion of *Weak Head Normal Form* to the less operational notion of *Lazy Normal Form*.

We have modeled lazy rewriting by a transformation of term rewriting systems, which avoids rewriting of lazy subterms to a large extent (optimal avoidance is impossible in nonorthogonal TRSs), and completely integrates pattern matching of

algebraic datatypes. When all arguments are annotated, the transformed system computes WHNFs.

We derive an efficient implementation on already efficient eager machinery from this model. Our method compares favourably to existing methods.

Our notion of Lazy Normal Forms (LNFs) could also be helpful in an implementation of *abstract rewriting*, as described in [BEØ93], or in the context of *theorem proving*.

We would like to thank John Field for his very insightful comments on an earlier version of this paper, and Jan Heering for his meticulous reading of a later version.

REFERENCES

[Amt93] Torben Amtoft. Minimal thunkification. In *Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, 1993.

[Bak94] Henry G. Baker. Cons should not cons its arguments, part II: Cheney on the M.T.A. Draft Memorandum, January 1994.

[Ben93] P.N. Benton. Strictness properties of lazy algebraic datatypes. In *Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 206–217. Springer-Verlag, 1993.

[BEØ93] Didier Bert, Rachid Echahed, and Bjarte M. Østvold. Abstract rewriting. In *Third International Workshop on Static Analysis, Padova, Italy*, volume 724 of *Lecture Notes in Computer Science*, pages 178–192. Springer-Verlag, 1993.

[BM92] Geoffrey Burn and Daniel Le Métayer. Cps-translation and the correctness of optimising compilers. Technical Report DoC92/20, Imperial College, Department of Computing, 1992.

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, 1991.

[BvEJ+87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W.Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In J.W. de Bakker, A.J. Nijman, and vol. II P.C. Treleaven, editors, *Proceedings PARLE'87 Conference*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer Verlag, 1987.

[CH90] Guy Cousineau and Gérard Huet. The CAML primer. Technical report, Inria, 1990. Version 2.6.1, available by ftp from ftp.inria.fr.

[DJ90] N. Dershowitz and J.-P Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol B.*, pages 243–320. Elsevier Science Publishers, 1990.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture*

*Notes in Computer Science*, pages 34–45. Springer-Verlag, 1987.

[GWM+92] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud. Introducing OBJ. In J.A. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification Using OBJ*. Cambridge University Press, 1992. To Appear.

[Heu88] Thierry Heuillard. Compiling conditional rewriting systems. In S. Kaplan and J.P. Jouannaud, editors, *Proceedins of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 111–128. Springer-Verlag, 1988.

[HL91] G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems part I and II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic; essays in honour of Alan Robinson*, pages 395–443. MIT Press, 1991.

[Ing61] P.Z. Ingermann. Thunks – a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[JL92] Simon L Peyton Jones and David Lester. *Implementing Functional Languages – A Tutorial*. Prentice Hall, 1992.

[JS89] Simon L Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Functional Programming and Computer Architecture*, pages 184–201. ACM, 1989.

[KKSdV93] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. The adequacy of term graph rewriting for simulating term rewriting. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*. John Wiley & Sons Ltd, 1993.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[MMR86] A. Martelli, C. Moiso, and C.F. Rossi. An algorithm for unification in equational theories. In *Proceedings of the Symposium on Logic Programming*, pages 180–186. IEEE Computer Society, 1986.

[Myc80] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In B. Robinet, editor, *International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[O'D77] M.J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, 1977.

[OLT94] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and

continuation-passing style. *Lisp and Symbolic Computation*, 7:57–82, 1994.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[PvE93] M J. Plasmeijer and M C J D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[SW94] Paul Steckler and Mitchell Wand. Selective thunkification. In *First International Static Analysis Symposium*, Namur, Belgium, 28-30 September 1994. also available by ftp as sas94.ps.Z from ftp.ccs.neu.edu:/pub/people/steck.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of computer Science, Carnegie Mellon University, November 1990.