Compact Normalisation Trace via Lazy Rewriting*

Quang-Huy Nguyen

Loria & Inria, Protheo
BP 101, 54602 Villers-lès-Nancy Cedex, France
email: Quang-Huy.Nguyen@loria.fr

Abstract. Innermost strategy is usually used in compiling term rewriting systems (TRSs) because it allows to build efficiently the result term in a bottom-up fashion. However, the innermost strategy does not always give the shortest normalising derivation. In many cases, using an appropriate laziness annotation on the arguments of the function symbols, we evaluate the lazy arguments only if it is necessary and hence, get a shorter derivation to the normal form while avoiding the non-terminating reductions. We provide in this work a transformation of the annotated TRSs, that allows to compute the normal form using an innermost strategy and to extract a lazy derivation in the original TRS from the normalising derivation in the transformed TRS. We apply our result to improve the efficiency of equational reasoning in the Coq proof assistant using ELAN as an external rewriting engine.

1 Introduction

Proof assistants like PVS [4], KIV [17] or Coq [13] advocate the use of equational reasoning for improving efficiency and reducing user interactions. In Coq, the proof objects are stored after each deduction step. The correctness of a proof is justified by type-checking these objects. This mechanism allows one to extract a certified program from the proof of its specification. However, a proof of equality requires a lot of user interactions and the generated proof object is huge since it contains the context of every rewrite step. In [1], we propose an approach to deal with these problems using ELAN [19] as a fast oracle: Coq first *delegates* a term normalisation process to ELAN and then, *replays* the normalisation trace provided by ELAN and which is a list of pairs $\langle rule\_label, position\_of\_contracted\_redex\rangle$ to get the normal form (NF) of the term. Trace replaying consists of the syntactic pattern matching between redex and the left hand side of the rule and the replacement of redex by the instantiation of the right hand side. The cost of syntactic pattern matching is linear in the size of the LHS which is relatively small. Meanwhile, the cost of finding out a redex done by ELAN depends on the size in the term to be reduced which can be

* All absent proofs can be found in the complete version available at http://www.loria.fr/~nguyenqh/publication/lrem-inria.ps.gz

very huge. Thus, ELAN performs the *proof search* and Coq *checks the proof* later. Coq and ELAN must work on the same *canonical* (confluent and terminating) TRS. Naturally, ELAN should return to Coq an as compact as possible trace to minimise the time needed for replaying. This time depends highly on the number of rewrite steps and also on the position of contracted redices since an inner one makes the proof objects for type-checking bigger.

In [8], the authors propose *lazy rewriting with laziness annotation*: each argument of a function symbol in the signature is annotated lazy or eager. Only the eager arguments are eagerly reduced. A lazy argument is reduced only if this reduction creates a new redex among the active subterms which contain it. We will give a formal definition of an active subterm in section 3 but one can see it as a subterm which is allowed to be eagerly reduced, the root being always active by default. For short, in the sequel of this paper, we denote lazy rewriting with laziness annotation by *lazy rewriting*. In many cases, lazy rewriting might give a shorter derivation to the NF than innermost strategy since the lazy arguments are evaluated by need. Furthermore, lazy rewriting allows dealing with infinite structures by avoiding reductions on the non-terminating branches. This property is important when working with non-terminating TRSs.

Due to the laziness annotations, some subterms of a term will not be rewritten during lazy rewriting. These subterms are called *lazy*. Lazy rewriting normalises a term to its lazy normal form where all *active subterms* are in head normal form (HNF). The *lazy subterms* may be reducible, but their reduction may not be finite if the TRS is not terminating. Otherwise, all lazy subterms can be normalised recursively until HNF. Thus, lazy rewriting is one way that can lead to the NF.

Also in [8], the authors show how to simulate correctly lazy rewriting by innermost rewriting with respect to (*w.r.t.*) a new TRS obtained by transforming the original TRS. This transformation process is called the *thunkification*. A simulation is correct if it is *complete, sound* and *termination preserving* [12] [7]. In other words, correctness guarantees that no information on NFs in the original TRS is lost. In order to keep the trace still useful for Coq, we also need to investigate the relation between the normalising derivations before and after any transformation.

In this paper, we show the correspondence between the normalisation traces in the original and transformed TRS and we propose a *normalisation procedure* based on lazy rewriting. This procedure yields a NF of the input term if the TRS is terminating and so, its unique NF if the TRS is canonical. On the other side, all normalising tasks in this procedure use the leftmost-innermost strategy, that can be very efficiently performed in ELAN. Our new normalisation procedure is used to replace the leftmost-innermost normalisation in the cases where by using a relevant laziness annotation, the yielded normalising derivations are shorter. Moreover, the subterms are sequentially reduced to HNF in a top-down fashion and hence, the outer redices are usually contracted first.

The thunkification only works with the TRSs where no non-variable term is put on the lazy arguments of a function symbol in the left hand sides of the rules.

In [8], the authors deal with this problem by transforming the original TRS into a *minimal* TRS (*i.e.* each LHS contains no more than two function symbols) [7]. Hence, this transformation generates a fairly large number of new but simple rules and of new function symbols. The minimal TRS given by the transformation is optimal for the *abstract rewriting machine* (ARM) [7] but not for ELAN whose compiler uses an improved version of the many-to-one pattern matching algorithm presented in [10]. Moreover, the transformation flattens the LHSs by introducing the new function symbols whose arity is different from the arity of the corresponding function symbols in the original TRS. This fact changes the position of redices and so, makes the correspondence between the normalising derivations more difficult to establish. Therefore, we propose another transformation (*preliminary transformation*) to overcome the limit of the thunkification for the *left-linear constructor-based* TRSs while keeping a good correspondence between the normalising derivations.

Since the TRSs are allowed to be overlapping, an order between the rules needs to be explicitly shown. Like most of functional languages, ELAN uses *textual ordering* and we decided to keep it instead of using specificity ordering as in [8]. On the other hand, we only consider the reductions (rewriting, lazy rewriting) on the terms without variables (*ground terms*). Furthermore, all rewrite rules are required to be *left-linear*. Completeness of the thunkification does not hold if the TRS is not left-linear. Some extensions are envisaged, for example, by checking equality between the original form (in the original signature) of the terms that instantiate the same variable. However, if the thunkification becomes too complicated, then the gain in performance will be less clear.

This paper is organized as follows: first, we review briefly the definitions on term rewriting; then, we give a rule-based formal definition of lazy rewriting and of the mechanism of thunkification presented in [8]; next, we show the correspondence between the normalisation traces and present the normalisation procedure; then, the preliminary transformation is described. A complete example is also given in order to illustrate the combination of the two transformations. We close the paper by discussing some related works.

2 Term rewriting

We mostly use the notations introduced in [5]. In particular, a *signature* Σ consists of a set $\mathcal{V}$ of variables and a set $\mathcal{F}$ of function symbols. *Arity* of a function symbol f in $\mathcal{F}$ is denoted by $ar(f)$.

The *set of terms* over Σ is denoted by $\mathcal{T}_\Sigma$ while the *set of ground terms* over Σ is denoted by $\mathcal{G}_\Sigma$. The function symbol heading a term t is denoted by $\mathcal{H}ead(t)$. A term is *linear* if no variable can occur more than once in it. A *position* within a term is represented by a sequence of natural numbers describing the path from the root of term to the head of the subterm at that position. The *position of the root of term* is an empty sequence and is denoted by ϵ. The *set of non-variable positions* in a term t is denoted by $\mathcal{FP}os(t)$. A subterm rooted at position p of a term t is denoted by $t|_p$. By $t[s]_p$ we denote the term t whose subterm at position p is replaced by a term s. The subterm $t|_{p_1}$ is a *context* of the subterm $t|_{p_2}$ if p_1 is a prefix of p_2.

A *substitution* is a mapping from the variables of $\mathcal{V}$ to terms. If σ is a substitution, then $t\sigma$ denotes the result of applying σ on t. We write $t\{x \mapsto s\}$ a term t in which each occurrence of variable x is replaced by a term s. A term s *overlaps* a term t if there exist a non-variable subterm $t|_p$ and a substitution σ such that $s\sigma = t\sigma|_p$. Notice that the variables of s and t are renamed before, if necessary, so that they are disjoint. By this definition, a term t always overlaps itself at root position. However, this case is trivial and is not considered as an overlap. Two terms s and t are overlapping if s overlaps t or vice versa.

A *rewrite rule* over a set of terms $\mathcal{T}_\Sigma$ is an ordered pair $\langle l, r\rangle$ of terms and is denoted by $l \to r$. We call l and r respectively the *left hand side* (LHS) and the *right hand side* (RHS) of rule. A rewrite rule is often restricted by two conditions: the LHS is not a variable and all variables occurred in the RHS must be contained by the LHS. A rewrite rule is called *left-linear/right-linear* if its LHS/RHS is linear.

A set of rewrite rules $\mathcal{R}$ over $\mathcal{T}_\Sigma$ is called a *term rewriting system* (TRS). In order to identify a rule in a TRS, in this paper, a rewrite rule is often denoted by $[\ell]\ l \to r$ where ℓ is the label of rule. A TRS $\mathcal{R}$ is called *left-linear* if all its rules are. A TRS is *overlapping* if the LHSs of its two (not necessary distinct) rules are. A symbol in $\mathcal{F}$ is called a *defined symbol* of a TRS $\mathcal{R}$ if it is the head symbol of the LHS of a rule in $\mathcal{R}$. A function symbol which is not a defined symbol is called a *constructor symbol* of $\mathcal{R}$. A TRS $\mathcal{R}$ is called *constructor-based* if no defined symbol can appear *inside* a LHS. In a constructor-based TRS, only the overlapping at the roots of its LHSs is possible.

Let $\mathcal{R}$ be a TRS. A term s in $\mathcal{T}_\Sigma$ rewrites to a term t in $\mathcal{T}_\Sigma$ in one *rewrite step* if there exist some rule $[\ell]\ l \to r$ in $\mathcal{R}$, a position p in s, and a substitution σ such that: $s|_p = l\sigma$ and $t = s[r\sigma]_p$. We denote this rewrite step by $s \to_{\mathcal{R}} t$ or $s \xrightarrow{\ell,p} t$ and the reflexive-transitive closure of relation $\to_{\mathcal{R}}$ by $\to^*_{\mathcal{R}}$. A *derivation* in $\mathcal{R}$ is any (finite or infinite) sequence of rewrite steps. From an operational point of view, a rewrite step consists of two phases: the pattern matching between $s|_p$ and l gives the substitution σ; the replacement of redex $s|_p$ in s by $r\sigma$. Since syntactic pattern matching yields no more than one solution, a position p and a rule ℓ suffice to memorise the rewrite step from a given term s. The pair $\langle \ell, p\rangle$ is called the *trace* of this rewrite step. A *redex* is an instance of the LHS of a rule. A term is said to be in *normal form* (NF) *w.r.t.* $\mathcal{R}$ if it contains no redex. A derivation from a term to one of its NFs is called a *normalising derivation* of this term.

Definition 1 (Normalisation trace). *If $t = t_1 \overset{\ell_1,p_1}{\to} t_2 \overset{\ell_2,p_2}{\to} \dots \overset{\ell_n,p_n}{\to} t_n$ is a normalisation derivation of term t w.r.t. $\mathcal{R}$, then $\mathbb{T}^{\mathcal{R}}_t = \{\langle \ell_1, p_1\rangle, \dots, \langle \ell_n, p_n\rangle\}$ is the corresponding normalisation trace of t.*

A term t is in *head normal form* (HNF) if there is no redex s such that $t \to^*_{\mathcal{R}} s$. If a term is in HNF, then its head symbol cannot be modified in any derivation issued from it. Hence, if a term t and all its subterms are in HNF, then t is in NF.

3 Lazy term rewriting

The signature is first given a laziness annotation that marks *lazy* or *eager* each argument of its function symbols.

Definition 2 (Laziness annotation). *Let $\Sigma = (\mathcal{V}, \mathcal{F})$ be a signature. The laziness annotation $\mathcal{L}$ of Σ is a mapping from $\mathcal{F}$ to $\{e, l\}^*$ such that:*
$\forall f \in \mathcal{F}, \mathcal{L}(f)$ is an $ar(f)$-tuple $\pi = \langle x_1, \ldots, x_{ar(f)} \rangle$ where $x_i = l$ means the i^{th} argument of f is lazy; $x_i = e$ means this argument is eager.

By π_i^f, we denote the i^{th} element of $\mathcal{L}(f)$. In the sequel, when speaking about lazy rewriting, a signature includes implicitly its laziness annotation. This laziness annotation divides the set of positions in a term into two subsets: the *active positions* and the *lazy positions*, that we define now.

Definition 3 (Active and lazy positions). *Let t be a term in $\mathcal{G}_\Sigma$. We have:*
- ϵ is always an active position.
- for any position p of t such that $\mathcal{H}ead(t|_p) = f$ and $\forall i = 1 \ldots ar(f)$: $p.i$ is active if and only if p is and $\pi_i^f = e$; otherwise, $p.i$ is called a lazy position.

The set of active positions in a term t is denoted by $\mathcal{AP}os(t)$. The subterms rooted at an active position is called active. The other subterms of the term are lazy. Thus, a subterm of t is active if and only if the path from its head to the root of t contains no edge that connects a function symbol to one of its lazy arguments.

Lazy rewriting is a restricted case of (standard) rewriting. Lazy rewriting only applies on *the active subterms* of a term and a crucial behaviour of lazy rewriting is that it can *change the laziness property of a subterm* from lazy to active (subterm activation).

In order to apply lazy rewriting on a term t, we first decorate it. That is, we annotate every subterm u of t by u_p^x where p is the position of u in t and $x = a$ meaning that u is an active subterm or $x = l$ meaning that u is lazy. All subterms of a lazy subterm are also lazy. The operator Φ decorates the subterm s which is rooted at position p and occurs as an argument of the symbol heading an active subterm of t: $\Phi(s, p, e) \mapsto\!\!\!\rightarrow s_p^a$ and $\Phi(s, p, l) \mapsto\!\!\!\rightarrow s_p^l$.

Let t be a term in $\mathcal{G}_\Sigma$. We associate to t a decorated term $t_{\mathcal{DC}} = \mathcal{DC}(t_\epsilon^a)$ where $\mathcal{DC}$ is defined by the rule system in figure 1.

Symbol For any $f \in \mathcal{F}$:

$$\mathcal{DC}(f_p^a(t_1, \ldots, t_n)) \mapsto\!\!\!\rightarrow f_p^a(\mathcal{DC}(\Phi(t_1, p.1, \pi_1^f)), \ldots, \mathcal{DC}(\Phi(t_n, p.n, \pi_n^f)))$$
$$\mathcal{DC}(f_p^l(t_1, \ldots, t_n)) \mapsto\!\!\!\rightarrow f_p^l(\mathcal{DC}({t_1}_{p.1}^{l}), \ldots, \mathcal{DC}({t_n}_{p.n}^{l}))$$

Constant For any constant c: $\mathcal{DC}(c_p^a) \mapsto\!\!\!\rightarrow c_p^a$ and $\mathcal{DC}(c_p^l) \mapsto\!\!\!\rightarrow c_p^l$

Fig. 1. Rules for term decoration

Let $\mathcal{G}_\Sigma^{\mathcal{D}init}$ be the set of decorated terms generated by applying $\mathcal{DC}$ on the terms in $\mathcal{G}_\Sigma$: $\mathcal{G}_\Sigma^{\mathcal{D}init} = \{t | \exists s \in \mathcal{G}_\Sigma : t = \mathcal{DC}(s_\epsilon^a)\}$. On the other hand, denote by $\mathcal{G}_\Sigma^{\mathcal{D}term}$ the set of all possible decorated terms generated by decorating the terms in $\mathcal{G}_\Sigma$ ($\mathcal{G}_\Sigma^{\mathcal{D}init} \subset \mathcal{G}_\Sigma^{\mathcal{D}term}$). The mapping $\mathcal{UD} : \mathcal{G}_\Sigma^{\mathcal{D}term} \to \mathcal{G}_\Sigma$ removes all decorations and returns the initial term.

The *lazy rewriting at the root of a decorated term* t by rule $l \to r$ is denoted by $[l \to r](t)$ and is described by the rules in figure 2. These rules transform a 4-tuple: the first component is the term to be reduced; the second component is the set of positions of the *essential subterms* (ES), *i.e.* the lazy subterms of t which correspond to a non-variable subterm of the pattern l; the third component is of the form $[l_1, \ldots, l_n \to r]$ where $l_1, \ldots, l_n$ are the subterms of the pattern l; the fourth component is a list of decorated terms to be matched correspondingly with $l_1, \ldots, l_n$.

The aim of these rules is for modeling both pattern matching and lazy rewriting in the same process as it is done in [3] for standard rewriting. Rule **Symbol-Clash** returns the initial term in case of conflict caused by an *active* subterm of t during pattern matching. *The lazy subterms never cause the conflict.* This fact differentiates pattern matching in lazy rewriting which is called *pattern matching modulo laziness* from (standard) pattern matching. If a subterm of t is lazy and the corresponding subterm of l is not a variable, then this lazy subterm is called *essential* and **EssentialSubterm** inserts its position into ES. **Decomposition** is applied if a symbol which roots an *active* subterm of t matches with the corresponding symbol in l. **Instantiation** instantiates a variable of the RHS with a subterm without decoration of t. **Replacement** replaces the term by the (decorated) instantiated RHS if ES is empty. In this case, no essential subterm has been revealed and pattern matching modulo laziness is identical with pattern matching. Moreover, σ is the substitution returned by the pattern matching modulo laziness. If ES is not empty, then **Activation** is applied to activate *one* essential subterm s of t and hence, all active subterms of s. One can choose s from ES using different strategies (leftmost, rightmost, ...). However, the results presented in this paper are independent of the used strategy. If **Activation** or **Replacement** is applied, then a *lazy rewrite step* is carried out and t is called a (lazy) redex since it *matches modulo laziness* with l. Formally, a (decorated) term t matches modulo laziness with a linear pattern l if and only if the symbols which root the active subterms of t match with the corresponding symbols of l:

$$\forall p \in \mathcal{APos}(\mathcal{UD}(t)) \cap \mathcal{FPos}(l) : \mathcal{H}ead(\mathcal{UD}(t)|_p) = \mathcal{H}ead(l|_p)$$

Figure 3 describes operator $\mathcal{LR}$ that performs lazy rewriting inside a decorated term t: $\mathcal{LR}$ replaces a subterm by the result of the application of lazy rewriting on it. Moreover, the decoration of this result needs to be adapted to its position in t by the shifting operator $\mathcal{SH} : \mathcal{G}_\Sigma^{\mathcal{D}term} \times \mathbb{N}^* \to \mathcal{G}_\Sigma^{\mathcal{D}term}$ such that $\mathcal{SH}(s,p)$ adds a prefix p to the position in the decoration of s and of all its subterms. We denote respectively the lazy rewriting relation *w.r.t.* $\mathcal{R}$ and its reflexive-transitive closure by $\leadsto_\mathcal{R}$ and $\leadsto_\mathcal{R}^*$. A lazy rewrite step by a rule labelled ℓ at the position p of term is denoted by $\overset{\ell,p}{\leadsto}$.

Initialisation $[l \to r](t) \;=\; [t][\emptyset][l \to r](t)$
Decomposition For any $f \in \mathcal{F}$

$$[t][ES][\ldots, f(t_1, \ldots, t_n), \ldots \to r](\ldots, f_p^a(s_1, \ldots, s_n), \ldots) \mapsto\!\!\!\rightarrow [t][ES][\ldots, t_1, \ldots, t_n, \ldots \to r](\ldots, s_1, \ldots, s_n, \ldots)$$

SymbolClash For any $f, g \in \mathcal{F}$ and $f \neq g$

$$[t][ES][\ldots, f(t_1, \ldots, t_n), \ldots \to r](\ldots, g_p^a(s_1, \ldots, s_m), \ldots) \mapsto\!\!\!\rightarrow t$$

EssentialSubterm For any $f \in \mathcal{F}$, any subterm s which is decorated with l:

$$[t][ES][\ldots, f(t_1, \ldots, t_n), \ldots \to r](\ldots, s, \ldots) \mapsto\!\!\!\rightarrow [t][ES \cup \{p\}][\ldots, \ldots \to r](\ldots, \ldots)$$

Instantiation For any $x \in \mathcal{V}$, any *decorated* subterm s:

$$[t][ES][\ldots, x, \ldots \to r](\ldots, s, \ldots) \mapsto\!\!\!\rightarrow [t][ES][\ldots, \ldots \to r\{x \mapsto \mathcal{UD}(s)\}](\ldots, \ldots)$$

Replacement $[t][\emptyset][\to r]() \;\mapsto\!\!\!\rightarrow\; \mathcal{DC}(r_\epsilon^a)$
Activation $[t][ES \cup \{p\}][\to r]() \;\mapsto\!\!\!\rightarrow\; t[\mathcal{DC}(\mathcal{UD}(t|_p)_p^a)]_p$

Fig. 2. Rules for lazy rewriting

Application For any decorated term t, any position p in $\mathcal{UD}(t)$ and any rule $l \to r \in \mathcal{R}$:

$$\mathcal{LR}(t, p, l \to r) = t[\mathcal{SH}([l \to r](t|_p), p)]_p \quad \text{if } t|_p \text{ is decorated with } a$$
$$\mathcal{LR}(t, p, l \to r) = t \quad \text{if } t|_p \text{ is decorated with } l$$

Fig. 3. Lazy rewriting inside a term

Definition 4 (Lazy normal form). *A decorated term t is said to be in lazy normal form (LNF) w.r.t. $\mathcal{R}$ if there exists no decorated term t' such that $t \rightsquigarrow_{\mathcal{R}} t'$.*

Example 5 ([15]). Consider the following rewrite system (infinite list):

$$\mathcal{R} = \begin{cases} [\text{r1}] \; 2nd(cons(x, cons(y, z))) \to y \\ [\text{r2}] \qquad\qquad inf(x) \to cons(x, inf(s(x))) \end{cases}$$

where $\mathcal{L}(2nd) = \langle e \rangle; \mathcal{L}(inf) = \langle e \rangle; \mathcal{L}(cons) = \langle e, l \rangle$.

The term $t = 2nd_\epsilon^a(inf_1^a(0_{1.1}^a))$ is derived to its LNF as follows:
$t \overset{r2,1}{\rightsquigarrow} 2nd_\epsilon^a(cons_1^a(0_{1.1}^a, inf_{1.2}^l(s_{1.2.1}^l(0_{1.2.1.1}^l)))) \overset{r1,\epsilon}{\rightsquigarrow}$
$2nd_\epsilon^a(cons_1^a(0_{1.1}^a, inf_{1.2}^a(s_{1.2.1}^a(0_{1.2.1.1}^a)))) \overset{r2,1.2}{\rightsquigarrow}$
$2nd_\epsilon^a(cons_1^a(0_{1.1}^a, cons_{1.2}^a(s_{1.2.1}^a(0_{1.2.1.1}^a), inf_{1.2.2}^l(s_{1.2.2.1}^l(s_{1.2.2.1.1}^l(0_{1.2.2.1.1.1}^l))))))$
$\overset{r1,\epsilon}{\rightsquigarrow} s_\epsilon^a(0_1^a)$. In the second step, the essential subterm $inf_{1.2}^l(s_{1.2.1}^l(0_{1.2.1.1}^l))$ is activated.

Remark 6. Let t and t' be two decorated terms . If $t \overset{l \to r}{\rightsquigarrow} t'$ by applying the **Replacement** rule, then $\mathcal{UD}(t) \overset{l \to r}{\to} \mathcal{UD}(t')$. Otherwise, if $t \overset{l \to r}{\rightsquigarrow} t'$ by applying the **Activation** rule, then $\mathcal{UD}(t) = \mathcal{UD}(t')$.

The next propositions show the relation between lazy rewriting and standard rewriting in the same TRS.

Proposition 7. *If the term t is in LNF w.r.t. $\mathcal{R}$, then $\mathcal{UD}(t)$ is in HNF w.r.t. $\mathcal{R}$.*

Proof. By induction on the size of t. If the size of t is 1, then t is a constant or a variable: t is active and t has no lazy subterm. Due to the definition of LNF, $\mathcal{UD}(t)$ is in HNF. Suppose that the proposition is correct for all terms of size from 1 to $n-1$. We prove that it is also correct for a term t of size n. The size of the subterms of t is less than or equal to n-1. Suppose that $\mathcal{UD}(t)$ is not in HNF. That is, *there exist a term $s \in \mathcal{G}_\Sigma$ and a rule $l \to r \in \mathcal{R}$ such that $\mathcal{UD}(t) \to^*_{\mathcal{R}} s$ and s matches with l* (*). Notice that the derivation from $\mathcal{UD}(t)$ to s only contracts the redices below the root. Since t is in LNF, all its active subterms are also in LNF. By induction hypothesis, these subterms (after being removed the decoration) are in HNF and *their head symbols cannot be changed by any derivation issued from $\mathcal{UD}(t)$* (**).

(*)(**) imply that the symbols which root the active subterms of t match with the corresponding symbols of l. In other words, t matches modulo laziness with l and t is not in LNF (absurdity).

Since the active subterms of a LNF are also in LNF, *all active subterms (without decoration) of a LNF are in HNF.*

Proposition 8. *If there exists an infinite derivation $t_0 \rightsquigarrow_{\mathcal{R}} t_1 \rightsquigarrow_{\mathcal{R}} \ldots$, then there exists $k \in N$ such that $\mathcal{UD}(t_0) \to_{\mathcal{R}} \mathcal{UD}(t_k)$.*

Proof. A lazy rewrite step that terminates by applying the **Activation** rule decreases strictly the number of lazy subterms in the term. Hence, there is no infinite sequence of these lazy rewrite steps in a derivation. So, there exists a smallest $k \geq 1$ such that $t_{k-1} \rightsquigarrow_{\mathcal{R}} t_k$ by applying the **Replacement** rule. Due to remark 6, we have: $\mathcal{UD}(t_0) = \ldots = \mathcal{UD}(t_{k-1}) \to_{\mathcal{R}} \mathcal{UD}(t_k)$.

A direct corollary of this proposition is that if standard rewriting *w.r.t.* $\mathcal{R}$ is terminating, then so is lazy rewriting *w.r.t.* $\mathcal{R}$ for any laziness annotation of the signature.

4 Thunkification

The *thunkification* of a TRS has been described in [8] for lazy graph rewriting. Here we consider lazy term rewriting and we do not require the LHSs of the original TRS to be *minimal.* This fact requires a small generalisation in the proofs. Our thunkification works on the left-linear but *possibly overlapping* TRSs where *all lazy subterms of the LHSs must be a variable.* In this case, no subterm activation is possible in a lazy rewriting step since the lazy subterms always correspond to a variable of the pattern. Thus, a lazy rewriting step finishes by applying the **Replacement** rule and hence, a lazy rewriting derivation only includes the terms in $\mathcal{G}_\Sigma^{\mathcal{D}init}$.

4.1 Thunkification description

The thunkification extends the signature and generates a new TRS by which innermost rewriting simulates lazy rewriting by the original TRS.

The *new signature* Σ' is built from the original signature $\Sigma = (\mathcal{V}, \mathcal{F})$ by adding new function symbols introduced during the thunkification: $\Theta, \tau_f, vec_f, vec_t, \lambda_t, inst$ for every $f \in \mathcal{F}$ and for some subterms t of the RHSs of the rules in the original TRS. The introduction of the new function symbols allows one to *mask* the lazy subterms of a term. A lazy $f-$rooted subterm s is masked (or *thunked*) by a subterm of form $\Theta(\tau_f, vec_f(\ldots))$ and hence, cannot eagerly be rewritten. The structure of s is stored in this Θ-rooted subterm so that one can recover it later.

The *thunkification of terms* is a mapping $\varphi : \mathcal{G}_{\Sigma}^{\mathcal{D}init} \rightarrow \mathcal{G}_{\Sigma'}$ which is defined by the rules in figure 4. We now describe the new TRS generated by the thunkification.

Definition 9 (Lazy argument position and subterm). *Let t be a term in $\mathcal{G}_{\Sigma}$. If there exist $p \in \mathcal{FP}os(t)$ and $i \in N$ such that $\mathcal{H}ead(t|_p) = f$ and $\pi_i^f = l$, then $p.i$ is called a lazy argument position in t while $t|_{p.i}$ is called a lazy argument subterm of t.*

Definition 10 (Migrant variable [8]). *A variable that appears at a lazy argument position in the LHS of a rewrite rule and at an active position in a subterm t of the RHS is called migrant in t.*

The laziness property of a subterm which instantiates a migrant variable is changed from lazy to active after the lazy rewrite step. Hence, we need to activate lazy rewriting on such a subterm later.

Definition 11 (Set of rules). *Let $\mathcal{R}$ be a TRS. The set of rewrite rules $\mathcal{S}$ generated by applying the thunkification on $\mathcal{R}$ is the union of four subsets $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$ which are defined as follows:*

1. *$\mathcal{S}_0$ contains the rule $l \rightarrow r'$ if and only if $l \rightarrow r \in \mathcal{R}$ and r' is built from r as follows:*
   - *In a bottom-up fashion, replace any lazy argument subterm t of the RHS r by* *$\Theta(\lambda_t, vec_t(x_1, \ldots, x_{n_t}))$ where $x_1, \ldots, x_{n_t}$ are all variables of t.*
   - *Replace any migrant variable x of the RHS r by $inst(x)$.*
2. *$\mathcal{S}_1 = \{inst(\Theta(\tau_f, vec_f(x_1, \ldots, x_{ar(f)}))) \rightarrow f(t_1, \ldots, t_{ar(f)}) \mid f \in \mathcal{F}\}$ where $t_i = inst(x_i)$ if $\pi_i^f = e$; otherwise $t_i = x_i$.*
3. *$\mathcal{S}_2 = \{inst(\Theta(\lambda_t, vec_t(x_1, \ldots, x_{n_t}))) \rightarrow t' | t$ has been replaced in 1 and $t' = t\{x_i \mapsto inst(x_i)\} \forall i$ such that x_i is a migrant variable of $t\}$.*
4. *$\mathcal{S}_3 = \{inst(x) \rightarrow x\}$.*

In fact, $\mathcal{S}_0$ contains all the rules in $\mathcal{R}$ whose RHSs have been changed (or *thunked*): every lazy argument subterm t is thunked by a subterm of form $\Theta(\lambda_t, vec_t(\ldots))$ and hence, t cannot eagerly be rewritten. A corresponding rule is inserted into $\mathcal{S}_2$ in order to recover t later. The insertion of a symbol *inst* allows to rewrite afterwards on the subterm which has instantiated a migrant variable. The unique rule of $\mathcal{S}_3$ allows to deal with a direct subterm which is not thunked of a symbol *inst*. This rule has the *lowest priority* and hence, is the last rule of $\mathcal{S}$ since we use the textual order.

In [8], only the non-variable lazy argument subterms of the RHS are thunked. Since an innermost strategy will be used for rewriting by $\mathcal{S}$, the subterms which instantiate a variable of the RHS are in NF before the application of the rule. In other words, the thunkification of the lazy argument subterms which are a variable is unnecessary. However, in this work, we also thunk these subterm in order to ensure the correctness of lemma 18 in section 5.

1. $\varphi(f_p^a(t_1, \dots, t_n)) \mapsto\!\!\!\!\!\twoheadrightarrow f(\varphi(\varPhi(t_1, p.1, \pi_1^f)), \dots, \varphi(\varPhi(t_n, p.n, \pi_n^f)))$
2. $\varphi(f_p^l(t_1, \dots, t_n)) \mapsto\!\!\!\!\!\twoheadrightarrow \varTheta(\tau_f, vec_f(\varphi(t_1{}^l_{p.1}), \dots, \varphi(t_n{}^l_{p.n})))$
3. $\varphi(c_p^a) \mapsto\!\!\!\!\!\twoheadrightarrow c$ and $\varphi(c_p^l) \mapsto\!\!\!\!\!\twoheadrightarrow \varTheta(\tau_c, vec_c)$ if c is a constant.

Fig. 4. Rules for φ

The set of terms $\mathcal{B}$ is defined as follows:

$$\mathcal{B} = \{g \in \mathcal{G}_{\Sigma'} \mid \exists g_0 \in \mathcal{G}_{\Sigma}^{\mathcal{D}init} \ : \ \varphi(g_0) \to_{\mathcal{S}}^{*} g\}$$

This definition of $\mathcal{B}$ is slightly different from [8] where g_0 is not thunked (by φ). The thunkification of g_0 helps to get the NF *w.r.t.* $\mathcal{S}$ more quickly. This fact is used in our normalisation procedure in section 5.

1. $\phi(g) = \varUpsilon(g, \epsilon, e)$
2. $\varUpsilon(inst(t), p, e) \mapsto\!\!\!\!\!\twoheadrightarrow \varUpsilon(t, p, e)$
3. $\varUpsilon(inst(t), p, l) \mapsto\!\!\!\!\!\twoheadrightarrow \varUpsilon(t, p, l)$
4. $\varUpsilon(\varTheta(\tau_f, vec_f(t_1, \dots, t_n)), p, e) \mapsto\!\!\!\!\!\twoheadrightarrow f_p^a(\varUpsilon(t_1, p.1, \pi_1^f), \dots, \varUpsilon(t_n, p.n, \pi_n^f))$
5. $\varUpsilon(\varTheta(\tau_f, vec_f(t_1, \dots, t_n)), p, l) \mapsto\!\!\!\!\!\twoheadrightarrow f_p^l(\varUpsilon(t_1, p.1, l), \dots, \varUpsilon(t_n, p.n, l))$
6. $\varUpsilon(\varTheta(\tau_c, vec_c), p, e) \mapsto\!\!\!\!\!\twoheadrightarrow c_p^a$ if c is a constant.
7. $\varUpsilon(\varTheta(\tau_c, vec_c), p, l) \mapsto\!\!\!\!\!\twoheadrightarrow c_p^l$ if c is a constant.
8. $\varUpsilon(\varTheta(\lambda_t, vec_t(t_1, \dots, t_{n_t})), p, e) \mapsto\!\!\!\!\!\twoheadrightarrow \varUpsilon(t\{x_1 \mapsto t_1\} \dots \{x_{n_t} \mapsto t_{n_t}\}, p, e)$
9. $\varUpsilon(\varTheta(\lambda_t, vec_t(t_1, \dots, t_{n_t})), p, l) \mapsto\!\!\!\!\!\twoheadrightarrow \varUpsilon(t\{x_1 \mapsto t_1\} \dots \{x_{n_t} \mapsto t_{n_t}\}, p, l)$
10. $\varUpsilon(f(t_1, \dots, t_n), p, e) \mapsto\!\!\!\!\!\twoheadrightarrow f_p^a(\varUpsilon(t_1, p.1, \pi_1^f), \dots, \varUpsilon(t_n, p.n, \pi_n^f))$
11. $\varUpsilon(f(t_1, \dots, t_n), p, l) \mapsto\!\!\!\!\!\twoheadrightarrow f_p^l(\varUpsilon(t_1, p.1, l), \dots, \varUpsilon(t_n, p.n, l))$
12. $\varUpsilon(c, p, e) \mapsto\!\!\!\!\!\twoheadrightarrow c_p^a$ if c is a constant.
13. $\varUpsilon(c, p, l) \mapsto\!\!\!\!\!\twoheadrightarrow c_p^l$ if c is a constant.

Fig. 5. Rules for ϕ

The mapping $\phi : \mathcal{B} \to \mathcal{G}_{\Sigma}^{\mathcal{D}init}$ relates the terms in $\mathcal{B}$ and the terms in $\mathcal{G}_{\Sigma}^{\mathcal{D}init}$ and is defined by the rules in figure 5. In reality, ϕ recovers the lazy subterms using the informations stored in their corresponding $\varTheta-$rooted subterms.

4.2 Correctness of thunkification

The lazy rewriting of the terms in $\mathcal{G}_{\Sigma}^{\mathcal{D}init}$ *w.r.t.* $\mathcal{R}$ can correctly be simulated by the innermost rewriting in a subset $\mathcal{B}$ of $\mathcal{G}_{\Sigma'}$ *w.r.t.* $\mathcal{S}$ via ϕ up to the criteria figured in [12]. That is, ϕ is *surjective, sound, complete* and *termination preserving.* ϕ is surjective since for every term g in $\mathcal{G}_{\Sigma}^{\mathcal{D}init}$: $\phi(\varphi(g)) = g$. In the following, $\rightarrow_{\mathcal{S}}$ denotes the innermost rewriting relation *w.r.t.* $\mathcal{S}$.

Theorem 12 (Soundness [8]). *Let g be a term in $\mathcal{B}$. If $g \rightarrow_{\mathcal{S}} g'$ then $\phi(g) \rightsquigarrow^*_{\mathcal{R}} \phi(g')$. More precisely: $g \rightarrow_{\mathcal{S}_0} g' \Rightarrow \phi(g) \rightsquigarrow_{\mathcal{R}} \phi(g')$ and $g \rightarrow_{\mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3} g' \Rightarrow \phi(g) = \phi(g')$.*

Lemma 13 ([8]). *If $g \in \mathcal{B}$ contains no symbol inst, then each active subterm of $\phi(g)$ inherits the head symbol from its corresponding subterm of g.*

Theorem 14 (Completeness [8]). *If $g \in \mathcal{B}$ is in NF w.r.t. S, then $\phi(g)$ is in LNF w.r.t. $\mathcal{R}$*

Theorem 15 (Termination preservation [8]). *If there exists an infinite derivation $g_0 \rightarrow_{\mathcal{S}} g_1 \rightarrow_{\mathcal{S}} \ldots$, then there exists $k \in N$ such that $\phi(g_0) \rightsquigarrow_{\mathcal{R}} \phi(g_k)$.*

Corollary 16. *If lazy rewriting w.r.t. $\mathcal{R}$ is terminating, then so is innermost rewriting w.r.t. $\mathcal{S}$.*

4.3 Correspondence of trace

We show in this section that a (lazy) normalisation trace of $\phi(g)$ *w.r.t.* $\mathcal{R}$ can be extracted from a normalisation trace of g *w.r.t.* $\mathcal{S}$. Suppose that each rule in $\mathcal{S}_0$ inherits the label from its corresponding rule in $\mathcal{R}$, we have:

Theorem 17 (Correspondence of trace). *Assume that $\mathbb{T}_g^{\mathcal{S}}$ is the normalisation trace of a term $g \in \mathcal{B}$ w.r.t. $\mathcal{S}$ in an innermost reduction strategy. Extracting from $\mathbb{T}_g^{\mathcal{S}}$ the traces of the rewrite steps performed by a rule in $\mathcal{S}_0$ yields a (lazy) normalisation trace $\mathbb{T}_{\phi(g)}^{\mathcal{R}}$ of $\phi(g)$ w.r.t. $\mathcal{R}$.*

5 Normalisation procedure

A term can be normalised by reducing sequentially all its subterms into HNF. Suppose that we need to normalise a term t by a *left-linear* and *terminating* TRS $\mathcal{R}$. The thunkification process is first applied on $\mathcal{R}$ to get the TRS $\mathcal{S}$. Next, t is thunked and normalised *w.r.t.* $\mathcal{S}$ to get g as a NF. Due to the rule in $\mathcal{S}_3$, g contains no symbol *inst*. Completeness implies that $\phi(g)$ is in LNF *w.r.t.* $\mathcal{R}$. In other words, all active subterms of $\phi(g)$ are in HNF *w.r.t.* $\mathcal{R}$ and inherit the head symbol from the corresponding subterms of g (lemma 13). Furthermore, in $\phi(g)$, an active subterm is never a subterm of a lazy subterm. In other words, $\phi(g)$ can be divided into two parts: the upper part contains the active subterms while the lower part contains the lazy subterms. Hence, the upper part of g contains the subterms which correspond to an active subterm of $\phi(g)$ and which are in

HNF *w.r.t.* $\mathcal{R}$. The lower part of g correspond to the lazy subterms of $\phi(g)$. The frontier between these two parts is composed by the symbols Θ (lemma 18).

Thus, we can unthunk (activate) the Θ-rooted subterms and reduce them into NF *w.r.t.* $\mathcal{S}$. By this reduction, some more subterms of g become in HNF *w.r.t.* $\mathcal{R}$. Notice that if a Θ-rooted subterm is activated, then its "active" subterms are also unthunked. The activating procedure of a Θ-rooted subterm will be described later by the operator ϕ^*. The process is recursively applied until all subterm of g are in HNF *w.r.t.* $\mathcal{R}$ and g is a NF of t.

Lemma 18. *Let g be a term in $\mathcal{B}$ and g contains no symbol inst. Then g is divided into two parts. The upper part contains the subterms which correspond to an active subterm of $\phi(g)$ while the lower part contains the subterms which correspond to a lazy subterm of $\phi(g)$. The frontier between these two parts is composed by the symbols Θ.*

Let g be a term in $\mathcal{G}_{\Sigma'}$. We define the set of disjoint *Θ-ancestor positions* of g as follows:

$\mathcal{P}_{la}(g) = \{p |\ p \in \mathcal{FP}os(g), \mathcal{H}ead(g|_p) = \Theta$ and $\mathcal{H}ead(g|_{p_1}) \neq \Theta$ for every prefix p_1 of $p\}$

$\mathcal{P}_{la}(g)$ can be computed by rules in figure 6. Intuitively, $\mathcal{P}_{la}(g)$ contains the frontier between two parts of g. The *activating operator* ϕ^* is a mapping from $\mathcal{G}_{\Sigma'}$ to $\mathcal{G}_{\Sigma'}$ and is defined by the rules in figure 7: ϕ^* activates (or *unthunks*) a $\Theta-$rooted term g and every $\Theta-$rooted subterm s of g such that $\phi(s)$ is an active subterm of $\phi(g)$. Figure 8 describes the normalisation procedure based on lazy rewriting ($norm(t, \mathcal{R})$).

Initialisation $\mathcal{P}_{la}(g) = \mathcal{L}a(g, \epsilon)$
Symbol $\mathcal{L}a(f(t_1, \ldots, t_n), p) \mapsto\!\!\!\!\rightarrow \mathcal{L}a(t_1, p.1) \cup \ldots \cup \mathcal{L}a(t_n, p.n)$ if $f \in \mathcal{F}$.
Constant $\mathcal{L}a(c, p) \mapsto\!\!\!\!\rightarrow \emptyset$ if c is a constant.
Discovery $\mathcal{L}a(\Theta(t_1, t_2), p) \mapsto\!\!\!\!\rightarrow \{p\}$

Fig. 6. Rules for $\mathcal{G}_{la}(t)$

1. $\phi^*(\Theta(\tau_f, vec_f(t_1, \ldots, t_n))) \mapsto\!\!\!\!\rightarrow f(\Psi(t_1, \pi_1^f), \ldots, \Psi(t_n, \pi_n^f))$
2. $\phi^*(\Theta(\lambda_t, vec_t(t_1, \ldots, t_{n_t}))) \mapsto\!\!\!\!\rightarrow t\{x_1 \mapsto t_1\} \ldots \{x_{n_t} \mapsto t_{n_t}\}$
3. $\phi^*(\Theta(\tau_c, vec_c)) \mapsto\!\!\!\!\rightarrow c$ if c is a constant.
4. $\Psi(t, e) \mapsto\!\!\!\!\rightarrow \phi^*(t)$ and $\Psi(t, l) \mapsto\!\!\!\!\rightarrow t$

Fig. 7. Rules for ϕ^*

Theorem 19. *If $\mathcal{R}$ is terminating and fulfills all necessary conditions for thunkification, then the procedure $norm(t, \mathcal{R})$ is also terminating and yields a NF of t w.r.t. $\mathcal{R}$.*

procedure $normalise(g \in \mathcal{G}_{\Sigma'}, \mathcal{S})$

1. g is normalised in the *leftmost-innermost strategy w.r.t.* $\mathcal{S}$ to get its NF g_{nf}
2. **if** there is no symbol Θ in g_{nf} **then return** g_{nf} **else** for all $p \in \mathcal{P}_{la}(g_{nf})$: $s{:=}\phi^*((g_{nf})|_p)$; $(g_{nf})|_p{:=}normalise(s, \mathcal{S})$

procedure $norm(t \in \mathcal{G}_\Sigma, \mathcal{R})$

1. Build $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_3$ from $\mathcal{R}$
2. $g := \varphi(\mathcal{DC}(t))$
3. $t_{nf} := normalise(g, \mathcal{S})$
4. **return** t_{nf}

Fig. 8. The normalisation procedure of t *w.r.t.* $\mathcal{R}$

Remark 20. The normalisation of a term t by procedure $norm(t, \mathcal{R})$ generates a trace $\mathbb{T}_t$ which is the list of the traces of all performed (leftmost-innermost) rewrite steps. Let us extract from $\mathbb{T}_t$ the pairs whose the first element is the label of a rule in $\mathcal{S}_0$. Due to theorem 17, this process yields a normalisation trace $\mathbb{T}_t^{\mathcal{R}}$ of t in $\mathcal{R}$ (in the sense of standard rewriting).

6 Preliminary transformation

In this section, we present a transformation that allows to eliminate all non-variable lazy argument subterms and hence, all non-variable lazy subterms of the LHSs. Our transformation works on the (left-linear) *constructor-based* TRSs. It is proved to be *correct* and to *preserve a good correspondence* between the normalisation traces in the original and transformed TRSs.

6.1 Transformation description

Let $\mathcal{R}$ be a left-linear constructor-based TRS. Suppose that $p.i$ is a non-variable lazy argument position in the LHS of a rule $l_s \to r \in \mathcal{R}$ and $\mathcal{H}ead(l_s|_p) = f$. We activate this position by adding a new function symbol f_e^p of arity $ar(f)$ where $\pi_j^{f_e^p} = \pi_j^f$ if $j \neq i$ and $\pi_i^{f_e^p} = e$ and by transforming $l_s \to r$ which is called the *source rule* as follows:

- Replace it by the rule $l_t \to r$ where l_t is l_s but $\mathcal{H}ead(l_t|_p) = f_e^p$. This rule is called the *transformed rule.*

- Add a new rule $l_s[x]_{p.i} \to l_t[x]_{p.i}$ where x is a fresh variable to $\mathcal{R}$ such that this rule *has the lowest priority* in case of overlapping. This rule is called the *added rule.*

All other rules of $\mathcal{R}$ are unchanged. This process is called a *transformation step* that eliminates *one* non-variable lazy argument subterm of a LHS of $\mathcal{R}$.

Example 21. Consider again the TRS in example 5. Applying the transformation on the rule $r1$ (*source rule*) yields the following TRS:

$$\mathcal{S} = \begin{cases} [r_t] \; 2nd(cons_e^1(x, cons(y,z))) \to y & (\textit{Transformed rule}) \\ [r2] \qquad\qquad\qquad inf(x) \to cons(x, inf(s(x))) & \\ [r_a] \qquad\quad 2nd(cons(x,x')) \to 2nd(cons_e^1(x,x')) & (\textit{Added rule}) \end{cases}$$

where $\mathcal{L}(2nd) = \langle e \rangle; \mathcal{L}(inf) = \langle e \rangle; \mathcal{L}(cons) = \langle e, l \rangle; \mathcal{L}(cons_e^1) = \langle e, e \rangle$.

Denote by $\mathcal{S}$ the new TRS generated by one transformation step. Σ' is the new signature ($\Sigma' = (\mathcal{V}, \mathcal{F} \cup \{f_e^p\})$). The set of terms $\mathcal{B}$ is defined as follows:

$$\mathcal{B} = \{g \in \mathcal{G}_{\Sigma'}^{\mathcal{D}term} \mid \exists g_0 \in \mathcal{G}_{\Sigma}^{\mathcal{D}term} : \; g_0 \rightsquigarrow_{\mathcal{S}}^* g\}$$

The mapping $\phi' : \mathcal{B} \to \mathcal{G}_{\Sigma}^{\mathcal{D}term}$ relates the terms in $\mathcal{B}$ with the terms in $\mathcal{G}_{\Sigma}^{\mathcal{D}term}$: $\phi'(g)$ is built by replacing every symbol f_e^p in g by f. Furthermore, the *laziness annotations of the subterms of* g *and* $\phi'(g)$ *are kept identical.*

We call $\phi'(g)$ the simulation of the lazy rewriting of the terms in $\mathcal{G}_{\Sigma}^{\mathcal{D}term}$ *w.r.t.* $\mathcal{R}$ by the lazy rewriting of the terms in $\mathcal{B}$ *w.r.t.* $\mathcal{S}$. Obviously, $\mathcal{S}$ is also constructor-based and left-linear. That is, the transformation can be repeated until the LHSs contain no non-variable lazy argument subterm. Our transformation is terminating since after each step, the number of non-variable lazy argument subterms of the LHSs is strictly decreased.

6.2 Correctness of transformation

The correctness of the overall transformation process can be deduced from the correctness of each transformation step up to the criteria figured in [12]. ϕ' is obviously surjective, since it is the identity mapping on the subset $\mathcal{G}_{\Sigma}^{\mathcal{D}term}$ of $\mathcal{B}$.

Theorem 22 (Soundness). *Let* g *be a term in* $\mathcal{B}$. *If* $g \rightsquigarrow_{\mathcal{S}} g'$ *then* $\phi'(g) \rightsquigarrow_{\mathcal{R}} \phi'(g')$. *More precisely: if* $g \rightsquigarrow_{\mathcal{S}} g'$ *by applying the added rule or the transformed rule, then* $\phi'(g) \rightsquigarrow_{\mathcal{R}} \phi'(g')$ *by applying the source rule at the same position. Otherwise,* $\phi'(g) \rightsquigarrow_{\mathcal{R}} \phi'(g')$ *by applying the same rule at the same position.*

Remark 23. If $g \rightsquigarrow_{\mathcal{S}} g'$ by a rewrite step using added rule, then $\mathcal{UD}(\phi'(g)) = \mathcal{UD}(\phi'(g'))$. Hence, if we only interest in the non-decorated terms as in standard rewriting, then this step is redundant.

Theorem 24 (Completeness). *If* $g \in \mathcal{B}$ *is in LNF w.r.t.* $\mathcal{S}$, *then* $\phi'(g)$ *is in LNF w.r.t.* $\mathcal{R}$.

Corollary 25 (Correspondence of trace). *Let* $\mathbb{T}_g$ *be a (lazy) normalisation trace of a term* $g \in \mathcal{B}$ *w.r.t.* $\mathcal{S}$. *Replacing the labels of the added rule and the transformed rule in* $\mathbb{T}_g$ *by the label of the source rule, yields a (lazy) normalisation trace of* $\phi'(g)$ *w.r.t.* $\mathcal{R}$.

Example 26. In example 21, term $t = 2nd(inf(0))$ is normalised *w.r.t.* $\mathcal{S}$ as follows : $2nd(inf(0)) \overset{r2,1}{\leadsto} 2nd(cons(0, inf(s(0)))) \overset{r_a,\epsilon}{\leadsto} 2nd(cons_e^1(0, inf(s(0))) \overset{r2,1.2}{\leadsto} 2nd(cons_e^1(0, cons(s(0), inf(s(s(0)))))) \overset{r_t,\epsilon}{\leadsto} s(0)$.

In the generated trace $\mathbb{T}_t^{\mathcal{S}} = \{\langle r2, 1\rangle; \langle r_a, \epsilon\rangle; \langle r2, 1.2\rangle; \langle r_t, \epsilon\rangle\}$, replacing r_t and r_a by $r1$ yields a (lazy) normalisation trace of term *w.r.t.* $\mathcal{R}$: $\mathbb{T}_t^{\mathcal{R}} = \{\langle r2, 1\rangle; \langle r1, \epsilon\rangle; \langle r2, 1.2\rangle; \langle r1, \epsilon\rangle\}$

Theorem 27 (Termination preservation). *If there exists an infinite derivation $g_0 \leadsto_{\mathcal{S}} g_1 \leadsto_{\mathcal{S}} \dots$, then there exists $k \in N$ such that $\phi'(g_0) \leadsto_{\mathcal{R}} \phi'(g_k)$.*

7 Combining two transformations

We describe in this section, the combination of the thunkification and the preliminary transformation above. If the LHSs of the considered TRS ($\mathcal{R}$) has no non-variable lazy subterms, then the sole thunkification is sufficient. In order to get a normalisation trace of a term t, we use the normalisation procedure described in section 5. Otherwise, the preliminary transformation is used to eliminate the non-variable lazy subterms of the LHSs. The new TRS ($\mathcal{S}$) generated by this transformation is then, transformed by the thunkification. Suppose that the normalisation procedure yields a trace $\mathbb{T}_t$. Due to remark 20, one can extract from $\mathbb{T}_t$ the trace $\mathbb{T}_t^{\mathcal{S}}$ of the corresponding (lazy) derivation by $\mathcal{S}$. Replacing the added rules and the transformed rules in $\mathcal{S}$ by their source rules in $\mathcal{R}$, one gets $\mathbb{T}_t^{\mathcal{R}}$ which is the trace of the corresponding (lazy) derivation by $\mathcal{R}$.

Nevertheless, due to remark 23, the rewrite steps by the added rules are redundant since our goal is to get a *normalisation trace in the sense of standard rewriting.* Therefore, we need to refine our trace by eliminating these redundant steps. This refinement should be done on $\mathbb{T}_t^{\mathcal{S}}$ before generating $\mathbb{T}_t^{\mathcal{R}}$ which is now the normalisation trace of t *w.r.t.* $\mathcal{R}$ in the sense of standard rewriting.

Example 28. We illustrate our method by considering the TRS ($\mathcal{R}$) in example 5. The thunkification cannot be directly applied on $\mathcal{R}$ since the LHS of $r1$ contains the non-variable lazy subterm $cons(y, z)$. Using the preliminary transformation, we get the TRS $\mathcal{S}$ in example 21. This TRS fulfills all necessary conditions for the thunkification which will give the following TRS:

$$\mathcal{U} = \begin{cases} \mathcal{U}_0 = \begin{cases} [r_t]\ 2nd(cons_e^1(x, cons(y, z))) \to y \\ [r2]\ inf(x) \to cons(x, \Theta(\lambda_{inf(s(x)}, vec_{inf(s(x))}(x)))) \\ [r_a]\ 2nd(cons(x, x')) \to 2nd(cons_e^1(x, inst(x'))) \end{cases} \\ \mathcal{U}_1 = \begin{cases} [r11]\ inst(\Theta(\tau_{cons}, vec_{cons}(x, y))) \to cons(inst(x), y) \\ [r12]\ inst(\Theta(\tau_{inf}, vec_{inf}(x))) \to inf(inst(x)) \\ [r13]\ inst(\Theta(\tau_{2nd}, vec_{2nd}(x))) \to 2nd(inst(x)) \\ [r14]\ inst(\Theta(\tau_{cons_e^1}, vec_{cons_e^1}(x, y))) \to cons_e^1(inst(x), inst(y)) \end{cases} \\ \mathcal{U}_2 = \{ [r21]\ inst(\Theta(\lambda_{inf(s(x))}, vec_{inf(s(x))}(x)))) \to inf(s(x)) \\ \mathcal{U}_3 = \{ [r31]\ inst(x) \to x \end{cases}$$

Consider the term $t = 2nd(inf(0))$. We normalise $\varphi(t) = 2nd(inf(0))$ *w.r.t.* $\mathcal{U}$ by the following leftmost-innermost derivation:

$2nd(inf(0)) \stackrel{r2,1}{\rightarrow} 2nd(cons(0, \Theta(\lambda_{inf(s(0)}, vec_{inf(s(0))}(0))))) \stackrel{r_a,\epsilon}{\rightarrow}$
$2nd(cons^1_e(0, inst(\Theta(\lambda_{inf(s(0)}, vec_{inf(s(0))}(0)))))) \stackrel{r21,1.2}{\rightarrow} 2nd(cons^1_e(0, inf(s(0))))$
$\stackrel{r2,1.2}{\rightarrow} 2nd(cons^1_e(0, cons(s(0), \Theta(\lambda_{inf(s(0)}, vec_{inf(s(0))}(s(0))))))) \stackrel{r_t,\epsilon}{\rightarrow} s(0)$.

$s(0)$ contains no symbol Θ and hence, the normalisation procedure finishes and return this term as a NF of t *w.r.t.* $\mathcal{S}$. Due to Soundness of the preliminary transformation, $s(0)$ is also a NF of t *w.r.t.* $\mathcal{R}$. Thanks to theorem 17, one can extract from the normalising derivation above a normalisation trace of t *w.r.t.* $\mathcal{S}$: $\mathbb{T}^{\mathcal{S}}_t = \{\langle r2, 1\rangle; \langle r_a, \epsilon\rangle; \langle r2, 1.2\rangle; \langle r_t, \epsilon\rangle\}$ (only the rewrite steps performed by a rule in $\mathcal{U}_0$ figure in $\mathbb{T}^{\mathcal{S}}_t$). Finally, we eliminate the steps by the added rule (r_a) and replace the transformed rule (r_t) by its source rule (r_1) to get a normalisation trace of t *w.r.t.* $\mathcal{R}$ (in the sense of standard rewriting): $\mathbb{T}^{\mathcal{R}}_t = \{\langle r2, 1\rangle; \langle r2, 1.2\rangle; \langle r1, \epsilon\rangle\}$. Notice that applying an innermost strategy on t using the rules in $\mathcal{R}$ leads to an infinite reduction.

8 Related works

Lazy rewriting can be obtained in OBJ [9] and CafeOBJ [6] using the *operator evaluation strategy* (E-strategy) where each operator (function symbol) has its own evaluation order. There are two suggested ways to simulate lazy rewriting by E-strategy: (1) omit the lazy arguments from the local strategy of its function symbol *or* (2) use the negative integers for these arguments. The *first method* does not behave well if there are some non-variable lazy subterms in LHSs as in example 5, where the second argument is omitted from the local strategy for *cons*. However, such a strategy reduces $2nd(inf(0))$ to $2nd(cons(0, inf(s(0))))$ instead of $s(0)$ since the subterm $inf(s(0))$ is not allowed to be reduced and $r1$ cannot be applied.

The *second method* is implemented in CafeOBJ using the *on-demand flag* [18]. A negative integer $-i$ in the local strategy given to a function symbol f means the i^{th} subterm of f is forced to be rewritten *if and only if it causes the conflict during pattern matching*. In example 5, the local strategy of *cons* is (1 -2 0) and $2nd(inf(0))$ is derived as follows: $2nd(inf(0)) \stackrel{r2,1}{\rightarrow} 2nd(cons(0, inf(s(0)))) \stackrel{r2,1.2}{\rightarrow}$ $2nd(cons(0, (cons(s(0), inf(s(s(0))))))) \stackrel{r1,\epsilon}{\rightarrow} s(0)$. In the second rewrite step, $r1$ is tried with the term $2nd(cons(0, inf(s(0))))$. The subterm $inf(s(0))$ causes the conflict and hence, it is forced to be rewritten. The E-strategies that can reduce a term to its to HNF is characterised in [15] for the *left-linear* and *constructor-based* TRSs. The on-demand flag in CafeOBJ is similar to the "essential node" notion and the thunkification has the same limit with the *first method* above. The preliminary transformation allows us to overcome this limitation for the left-linear and constructor-based TRSs.

Context-sensitive rewriting [14] can be seen as a restricted case of lazy rewriting where the subterm activation is not allowed. In order to simulate correctly standard rewriting by context-sensitive rewriting, one needs to use a *canonical replacement map* which is equivalent to the condition that all lazy subterms of the LHSs must be a variable. In other words, context-sensitive rewriting shares the same limit with the first method above.

9 Conclusion

In this paper, we described lazy rewriting and the mechanism of thunkification under a rule-based form. We showed the relation between the normalising derivations in the TRSs before and after the thunkification and proposed a normalisation procedure based on lazy rewriting. A preliminary transformation that allows to extend the application scope of the thunkification method while preserving the correspondence between the normalisation traces was also presented.

The optimal derivation is undecidable in general [16] [11] and even when it is decidable, the decision procedure is complicated to implement. In practice, most interesting results only involve the orthogonal constructor-based TRSs [20] [2] [21]. We think that our normalisation procedure is helpful since the normalisation procedure is reasonably efficient in ELAN, thanks to the correct simulations, while the generated trace is more compact and still useful for Coq, thanks to the nice correspondence between the normalising derivations before and after each transformation. Moreover, the TRSs are allowed to be overlapping.

A natural question may rise: which arguments should be marked lazy in each function symbol ? There is no general answer, but intuitively, the variables that appear in the LHS but not in the RHS of the same rule should be lazy. Thus, in an *if-then-else* construction like $\{if(true, x, y) \to x;\ if(false, x, y) \to y\}$, the two last arguments of if should be lazy. If all variables in the LHS also appear in the RHS, then all redices are necessary and lazy or outermost strategies do not give a shorter derivation than innermost strategies. Furthermore, the variable marked lazy should not appear more than once in the RHS since this duplicates the reduction of the terms which will instantiate this variable. In such cases, sharing is required with lazy rewriting. In our work, sharing is only helpful if it is implemented in both Coq and ELAN sides. This requires some extensions in Coq replaying procedure and ELAN compiler that we are investigating.

10 Acknowledgements

I sincerely thank Claude Kirchner, Hélène Kirchner and the anonymous referees for their constructive comments on the earlier version of this paper. I am also grateful to Mark van den Brand for pointing [8] out to me.

References

1. C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in Coq. In J. Despeyroux, editor, *Proceeding of 2nd Workshop on Logical Frameworks and Metalanguages*, June 2000.

2. S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer-Verlag LNCS 632, September 1992.
3. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 2001.
4. CSL/SRI. The PVS homepage. http://pvs.csl.sri.com.
5. N. Dershowitz and J-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.
6. R. Diaconescu and K. Futatsugi. An overview of CafeOBJ. In C. Kirchner and H. Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier Science Publishers, 2000.
7. W. Fokkink, J. Kamperman, and P. Walters. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Transactions on Programming Languages and Systems*, 20(3):679–706, May 1998.
8. W. Fokkink, J. Kamperman, and P Walters. Lazy rewriting on eager machinery. *ACM Transactions on Programming Languages and Systems*, 2(1):45–86, January 2000.
9. J A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J-P. Jouannaud. Introducing OBJ. Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, Menlo Park , CA , USA, March 1992.
10. A. Graf. Left-to-right tree pattern matching. In Ronald V. Book, editor, *Rewriting Techniques and Applications '91*, pages 323–334. Springer-Verlag LNCS 488, 1991.
11. G. Huet and J-J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J-L. Lassez and G. D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1991.
12. J. Kamperman and P. Walters. Minimal term rewriting systems. In M. Haveraaen, O. Owe, and O. J. Dahl, editors, *Recent Trends in Data Type Specification*, pages 274–290. Springer-Verlag LNCS 1130, 1995.
13. LogiCal/INRIA. The Coq homepage. INRIA, http://coq.inria.fr.
14. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), January 1998.
15. M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flag. In K. Futatsugi, editor, *Proceeding of 3rd International Workshop on Rewriting Logic and its Applications*, pages 211–227, 2000.
16. M.J. O'Donnell. Equational logic programming. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, *Logic Programming*, chapter 2. Oxford University Press, 1995. Preprint.
17. University of Karlsruhe. The KIV homepage. http://i11www.ira.uka.de/ kiv/KIV-KA.html.
18. K. Ogata and K. Futatsugi. Operational semantics of rewriting with the on-demand evaluation strategy. In *Proceeding of ACM Symposium on Applied Computing*, pages 756–763, 2000.
19. PROTHEO/LORIA. The ELAN homepage. LORIA, http://elan.loria.fr.
20. R I. Strandh. Classes of equational programs that compile into efficient machine code. In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 449–461. Springer-Verlag LNCS 355, April 1989.
21. S. Thatte. A refinement of strong sequentiality for term rewriting with constructors. *Information and Computation*, 72(1):46–65, January 1987.