*Electronic Notes in Theoretical Computer Science 57 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume57.html*](http://www.elsevier.nl/locate/entcs/volume57.html) *22 pages*

*Compact Normalisation Trace via Lazy Rewriting*

*Quang-Huy Nguyen* [*1*](#_bookmark0)*LORIA & INRIA*

*BP 101, 54602 Villers-l es-Nancy Cedex, France*

*Abstract*

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

## *1 Introduction*

*Proof assistants like PVS* [*[4],*](#_bookmark27) *KIV* [*[17*](#_bookmark40)*] or Coq* [*[13]*](#_bookmark36) *advocate the use of equa-* tional reasoning for improving eÆciency and reducing user interactions. In Coq, the proof objects are stored in each deduction step. The correctness of proofs is justi ed by type-checking these objects. This mechanism improves reliability and allows one to extract a certi ed program from the proof of its speci cation. However, an equality proof requires a lot of user interac- tions and the generated proof object is huge since it contains the contexts of rewrite steps. In [[1](#_bookmark24)], we propose an approach to deal with these problems using ELAN [[19](#_bookmark42)] as a fast oracle: Coq delegates term normalisation process to ELAN, and then replays normalisation traces provided by ELAN and which are the lists of pairs hrule label; position of contracted redexi to get normal forms (NFs). Trace replaying consists of the syntactic pattern matching be- tween redex and the left hand side (LHS) of rule and the replacement of redex

*1 Email:* [*Quang-Huy.Nguyen@loria.fr*](mailto:Quang-Huy.Nguyen@loria.fr)

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

*by the instantiated right hand side (RHS). Since the rule and the redex are* already given by ELAN, syntactic pattern matching in Coq in the worst case is linear in the size of this redex. Meanwhile, the cost of nding out a redex depends on the size of terms which can be very huge. Thus, ELAN performs the proof search and Coq checks this proof later. Coq and ELAN must work on the same canonical (con uent and terminating) TRS. In this context, ELAN should return to Coq the as compact as possible traces to minimise the time needed for replaying. This time depends not only on the number of rewrite steps but also on the positions of contracted redices since contracting inner redices creates bigger proof objects.

*Fokkink, Kamperman and Walters propose in* [*[8]*](#_bookmark31) *the lazy rewriting with* laziness annotation: every argument of function symbols is annotated lazy or eager. Only the eager arguments are eagerly reduced. A lazy argument is reduced only if this reduction creates new redexes among the active subterms which contain it. We will give a formal de nition of active subterms in sec- tion [3](#_bookmark2) but one can see them as the subterms which are 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 rewriting since lazy arguments are evaluated by need. Furthermore, lazy rewriting allows dealing with in nite structures by avoiding reductions on non-terminating branches. This property is important when working with non-terminating TRSs.

*Due to laziness annotations, some subterms of a term will not be rewritten* during lazy rewriting. These subterms are called lazy. Lazy rewriting nor- malises a term to its lazy normal forms where all active subterms are in head normal form (HNF). Some lazy subterms may be reducible, but their reduction may not be nite if the TRS is not terminating. Otherwise, all lazy subterms can recursively be normalised until HNF. In this case, lazy rewriting provides a means to get NFs.

*Also in* [*[8],*](#_bookmark31) *the authors show how to correctly simulate lazy rewriting by* innermost rewriting with respect to (w.r.t.) a new TRS obtained by transform- ing the original TRS. This transformation process is called thunki cation. A simulation is correct if it is complete, sound and termination preserving [[12]](#_bookmark35) [[7].](#_bookmark30) In other words, correctness guarantees that no information on NFs in the orig- inal TRS is lost. In addition, we are strongly concerned with the relation be- tween the normalisation derivations in order to keep lazy normalisation traces still useful for the proof assistant.

*In this paper, we show the correspondence between normalisation traces in* original and transformed TRSs and propose a normalisation procedure based on lazy rewriting. This procedure yields a NF of input terms if the TRS is terminating and so, their unique NF if the TRS is canonical. On the other hand, all normalising tasks in this procedure use leftmost-innermost strategy, that can eÆciently be performed in ELAN. Our normalisation procedure is used

*to replace leftmost-innermost normalisation in the cases where using relevant* laziness annotations we can get shorter normalising derivations. Moreover, since subterms are sequentially reduced to HNF in a top-down fashion, outer redices are usually contracted rst in this procedure.

*Thunki cation 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 rewrite rules. In [[8],](#_bookmark31) 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].](#_bookmark30) Hence, this transformation generates a fairly large number of new but simple rules and of new function symbols. The minimal TRS given by their transformation is optimal for the abstract rewriting machine (ARM) [[7]](#_bookmark30) but not for ELAN whose compiler uses an improved version of the many-to-one pattern matching algorithm presented in [[10].](#_bookmark33) Moreover, this transformation

*attens the LHSs by introducing new function symbols with new arity. This*

*fact changes the positions of redices and hence, makes the correspondence* between normalising derivations more diÆcult to establish. Therefore, we propose in this paper another transformation (preliminary transformation) to overcome the limit of thunki cation for left-linear constructor-based TRSs while keeping a good correspondence between normalising derivations.

*Since TRSs are allowed to be overlapping in this work, an order between* rewrite rules needs to be explicitly shown. Like most of functional languages, ELAN uses textual ordering and we decided to keep it instead of using speci-

*city ordering as in* [*[8].*](#_bookmark31) *On the other hand, we only consider reductions*

*(rewriting, lazy rewriting) on terms without variables (ground terms). Fur-* thermore, all rewrite rules are required to be left-linear. Completeness of thunki cation does not hold if the TRS is not left-linear. Some extensions are envisaged, for example, by checking equality between the original forms (in original signature) of the terms that instantiate the same variable. However, if the transformations become too complicated, then the gain in performance will be less clear.

*This paper is organized as follows. In section* [*2,*](#_bookmark1) *we brie y review stan-* dard de nitions on term rewriting. Section [3](#_bookmark2) gives a rule-based de nition of lazy rewriting. Thunki cation is described in section [4](#_bookmark8) where we show the correspondence between normalisation traces. In section [5](#_bookmark14) we present the nor- malisation procedure based on lazy rewriting. The preliminary transformation is described in section [6.](#_bookmark19) In section [7,](#_bookmark23) a complete example is given in order to illustrate the combination of two transformations. We close the paper by discussing some related works. All absent proofs can be found at the complete version available at <http://www.loria.fr/~nguyenqh/publication>.

## *2 Term Rewriting*

*We mostly use the notations introduced in* [*[5].*](#_bookmark28) *In particular, a signature consists of a set V of variables and a set F of function symbols. Arity of*

*function symbol f in F is denoted by ar(f ).*

*The set of terms over is denoted by T while the set of ground terms over*

*is written G. The function symbol heading a term t is denoted by Head(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 . The set of non- variable positions in a term t is denoted by FPos(t). A subterm rooted at position p of term t is denoted by tjp. By t[s]p we denote the term t whose subterm at position p is replaced by the term s. The subterm tjp1 is a context of subterm tjp2 if p1 is a pre x of p2.

*A substitution is a mapping from the variables of V to terms. If is a*

*substitution, then t denotes the result of applying on t. We write tfx 7! sg* the term t in which each occurrence of variable x is replaced by the term s. Term s overlaps the term t if there exist a non-variable subterm tjp and a substitution such that s = t jp. Notice that the variables of s and t are

*renamed before, if necessary, so that they are disjoint. By this de nition, 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 T is an ordered pair hl; ri of terms and is denoted by* l ! r. We call l and r respectively the left hand side and the right hand side of rule. Rewrite rules are 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 R over T is called a term rewriting system (TRS).* In order to identify rewrite rules in TRSs, in this paper, a rewrite rule is often denoted by [`] l ! r where ` is the label of the rule. A TRS R is called left-linear if all its rules are. A TRS is overlapping if the LHSs of two (not necessary distinct) rules are. A symbol in F is called de ned symbol of a TRS R if it is the head symbol of the LHS of some rule in R. Function symbols which are not de ned symbols are called constructor symbols of R. R is called constructor-based if no de ned symbol can appear inside a LHS. In constructor-based TRSs, only overlapping at the roots of LHSs is allowed.

*Let R be a TRS. A term s in T rewrites to a term t in T in one rewrite* step if there exist some rule [`] l ! r in R, a position p in s, and a substitution

*such that: sjp = l and t = s[r ]p.*

*We denote this rewrite step by s !R t or s !t and the re exive-transitive*

*`;p*

*closure of relation !R by ! . A derivation in R is any ( nite or in nite)*

*R*

*sequence of rewrite steps. From an operational point of view, a rewrite step* consists of two phases: the pattern matching between sjp and l giving a substi- tution , and the replacement of redex sjp in s by r . Since syntactic pattern matching yields no more than one solution, position p and label ` suÆce to memorise the rewrite step on a given term s. The pair h`; pi is called the

*trace of this rewrite step. The subterm sjp is also called a redex since it is an* instance of the LHS of some rule in R. A term is said to be in normal form

*w.r.t. R if it contains no redex. A derivation from a term to one of its NFs is* called a normalising derivation of this term.

*De nition 2.1 (Normalisation trace) If t = t1*

*`1;p1*

*t2*

*!*

*`2;p2*

*:::*

*!*

*`n!;pn t is a*

*normalisation derivation of term t w.r.t. R, then the list*

*n*

*TR = fh` ;p i; ::: ; h` ;p ig*

*t 1 1 n n*

*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 ! s. If a term is in HNF, then its head symbol cannot be modi ed in any*

*R*

*derivation issued from it. Hence, if a term t and all its subterms are in HNF,* then t is in NF.

*In this paper, we use the symbol !7! to describe the evaluation rules in the* de nitions of new operators.

## *3 Lazy Term Rewriting*

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

*De nition 3.1 (Laziness annotation) Let = (V; F) be a signature. The* laziness annotation L of is a mapping from F to fe; lg such that:

*8f 2 F; L(f ) is an ar(f )-tuple = hx1;::: ; xar(f)i where xi = l means the ith argument of f is lazy; xi = e means this argument is eager.*

*By f , we denote the ith element of L(f ). In the sequel, when speaking* about lazy rewriting, a signature always includes 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 de ne now.

*i*

*De nition 3.2 (Active and lazy positions) Let t be a term in G. We* have:

* *the root occurrence is always an active position.*
* *for any position p of t such that Head(tjp) = f and 8i = 1 ::: ar(f ): p:i is active if and only if p is and f = e; otherwise, p:i is called a lazy position.*

*i*

*The set of active positions in a term t is denoted by APos(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 symbol 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 (normal) rewriting. Lazy rewriting* only applies on active subterms and a crucial behaviour of lazy rewriting is that

*it can change the laziness property of subterms from lazy to active (subterm* activation).

*In order to apply lazy rewriting on a term t, we rst decorate it. That* is, we annotate every subterm u of t by ux where p is the position of u in t and x = a meaning that u is active while x = l meaning that u is lazy. All subterms of a lazy subterm are also lazy. The following operator decorates subterm s which is rooted at position p and occurs as an argument of the

*p*

*symbol heading an active subterm of t: (s; p; e) !7! sa and (s; p; l) !7! sl .*

*p p*

*Let t be a term in G. We associate to t a decorated term tDC = DC(ta)*

*where DC is de ned by the rule in gure* [*1.*](#_bookmark3)

*Symbol For any f 2 F:*

*DC(f (t ;::: ;t )) !7! f (DC( (t ; p:1; ));::: ; DC( (t ; p:n; )))*

*a*

*a*

*f*

*f*

*p*

*1*

*n*

*p*

*1*

*1*

*n*

*n*

*DC(f l(t1;::: ; tn)) !7! f l(DC(t1l );::: ; DC(tnl ))*

*p*

*p*

*p:1*

*p:n*

*Constant For any constant c:*

*DC(ca) !7! ca and DC(cl ) !7! cl*

*p*

*p*

*p*

*p*

*Fig. 1. Evaluation rules for term decoration*

*Let GDinit be the set of decorated terms generated by applying DC on terms*

*in G*

*: GDinit = ftj9s 2 G*

*: t = DC(sa)g. On the other hand, let us denote*

*by GDterm*

*the set of all possible decorated terms generated by decorating*

*terms in G*

*(GDinit GDterm). The mapping UD : GDterm ! G*

*removes all*

*decorations and returns the initial term.*

*Lazy rewriting at the root of a decorated term t by rule l ! r is denoted* by [l ! r](t) and is described by the rules in gure [2.](#_bookmark6) These rules transform a 4-tuple: the rst component is the term to be reduced; the second component is the set of positions of essential subterms (ES), i.e. the lazy subterms of t which correspond to a non-variable subterm of l; the third component is of the form (l1;::: ; ln ! r) where l1;::: ; ln are the subterms of l; the fourth component is a list of decorated terms to be correspondingly matched with l1;::: ; ln.

*The aim of these rules is for modelling both pattern matching and lazy* rewriting in the same process as it is done in [[3]](#_bookmark26) for term rewriting. The rule SymbolClash returns the initial term in case of con ict caused by an active subterm of t during the pattern matching phase. The lazy subterms never cause con icts. This fact di erentiates pattern matching in lazy rewrit- ing which is called pattern matching modulo laziness from (normal) 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 in-

*serts its position into ES. Decomposition is applied if a symbol which roots* an active subterm of t matches with the corresponding symbol in l. Instan- tiation 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, is the substitution returned by 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 di erent 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) re- dex 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:

*8p 2 APos(UD(t)) \ FPos(l) : Head(UD(t)jp) = Head(ljp)*

*Figure* [*3*](#_bookmark7) *describes operator LR that performs lazy rewriting inside a dec-* orated term t: 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 SH : GDterm N ! GDterm such

*that SH(s; p) adds a pre x p to the position in the decoration of s and of all*

*its subterms. We respectively denote the lazy rewriting relation w.r.t. R and*

*its re exive-transitive closure by ;R and ; . A lazy rewrite step by a rule*

*R*

*`;p*

*labelled ` at position p of term is denoted by ;.*

*De nition 3.3 (Lazy normal form) A decorated term t is said to be in* lazy normal form (LNF) w.r.t. R if there exists no decorated term t0 such that t ; t0 .

*R*

*Example 3.4* [*([15])*](#_bookmark38) *Consider the following TRS (in nite list):*

*R = <8*

*[r1] 2nd(cons(x; cons(y; z))) ! y*

*:[r2] inf (x) ! cons(x; inf (s(x)))*

*where L(2nd) = hei; L(inf ) = hei; L(cons) = he; li.*

*The term t = 2nda(inf a(0a*

*)) is derived to its LNF as follows:*

*r2;1 a*

*t*

*1*

*a a l*

*1:1*

*sl l*

*r1;*

*; 2nd (cons1(01:1; inf 1:2(*

*1:2:1(01:2:1:1)))) ;*

*2nda(consa(0a*

*; inf a*

*(sa*

*(0a*

*r2;1:2*

*)))) ;*

*1 1:1*

*1:2*

*1:2:1*

*1:2:1:1*

*2nda(consa(0a*

*; consa*

*(sa*

*(0a*

*); inf l*

*(sl*

*(sl*

*(0l*

*))))))*

*1 1:1*

*1:2*

*1:2:1*

*1:2:1:1*

*1:2:2*

*1:2:2:1*

*1:2:2:1:1*

*1:2:2:1:1:1*

*r1;*

*sa a l*

*; (0 ). In the second step, the essential subterm inf*

*(sl*

*(0l*

*)) is*

*1 1:2*

*1:2:1*

*1:2:1:1*

*activated.*

*Initialisation*

*[l ! r](t) !7! [t][;](l ! r)(t)*

*Decomposition For any f 2 F*

*[t][ES](::: ;f (t1;::: ; tn);::: ! r)(::: ;f a(s1;::: ; sn);:: :) !7!*

*p*

*[t][ES](::: ; t1;::: ; tn;::: ! r)(::: ; s1;::: ; sn;:: :)*

*SymbolClash For any f; g 2 F and f 6= g*

*[t][ES](::: ;f (t1;::: ; tn);::: ! r)(::: ; ga(s1;::: ; sm);:: :) !7! t*

*p*

*EssentialSubterm For any f 2 F, any subterm s decorated with l:*

*[t][ES](:::;f (t1;::: ; tn);::: ! r)(::: ; s;:: :) !7!*

*[t](ES [ fpg](::: ;::: ! r)(::: ;:: :)*

*Instantiation For any x 2 V, any decorated subterm s: [t][ES](:::; x;::: ! r)(::: ; s;:: :) !7!*

*[t][ES](::: ;::: ! rfx 7! U D(s)g)(::: ;:: :)*

*Replacement*

*[t][;](! r)() !7! DC(ra)*

*Activation*

*[t][ES [ fpg](! r)() !7! t[DC(UD(tjp)a)]p*

*p*

*Fig. 2. Evaluation rules for lazy rewriting*

*Application For any decorated term t, any position p in UD(t) and any*

*rule l ! r 2 R:*

*LR(t; p; l ! r) !7! t[SH([l ! r](tjp); p)]p if tjp is decorated with a*

*LR(t; p; l ! r) !7! t*

*if tjp is decorated with l*

*Fig. 3. Evaluation rules for lazy rewriting inside a term*

*Remark 3.5 Let t and t0 be two decorated terms . If t l!r t0 by applying the*

*!*

*;*

*l r*

*! UD*

*;*

*Replacement rule, then UD(t)*

*(t0 ). Otherwise, if t l!r t0 by applying*

*the Activation rule, then UD(t) = UD(t0 ).*

*The next propositions show the relation between lazy rewriting and rewrit-* ing in the same TRS.

*Proposition 3.6 If t is in LNF w.r.t. R, then UD(t) is in HNF w.r.t. 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 de nition of LNF, UD(t) is in HNF. Suppose* now that the proposition is correct for all terms of size strictly smaller than from n. The size of the subterms of t is less than or equal to n 1. Suppose that UD(t) is not in HNF. That is, there exist a term s 2 G and a rule

*l ! r 2 R such that UD(t) ! s and s matches with l (\*). Notice that the*

*R*

*derivation from UD(t) to s only contracts the redices below root. Since t is in* LNF, all its active subterms are also in LNF. By induction hypothesis, these subterms (after being removed their decoration) are in HNF and their head symbols cannot be changed by any derivation issued from 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 which contradict the hypothesis and nishes the proof. *2*

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

*Proposition 3.7 If there exists an in nite derivation t0 ;R t1 ;R :: :, then there exists k 2 N such that UD(t0) !R UD(tk).*

*Proof. Lazy rewrite steps that terminate by applying Activation strictly* decrease the number of lazy subterms. Hence, there is no in nite sequence of these lazy rewrite steps in a derivation. That is, there exists a smallest k 1 such that tk 1 ;R tk by applying Replacement. Due to remark [3.5,](#_bookmark5) we have: UD(t0) = ::: = UD(tk 1) !R UD(tk). *2*

*A direct corollary of this proposition is that if rewriting w.r.t. R is termi-* nating, then so is lazy rewriting w.r.t. R regardless of laziness annotations.

## *4 Thunki cation*

*Thunki cation has been described in* [*[8]*](#_bookmark31) *for lazy graph rewriting. We consider* lazy term rewriting and do not require the LHSs of the original TRS to be minimal [[7].](#_bookmark30) This fact requires a small generalisation in the proofs. Our thunki cation works on 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 lazy subterms always correspond to the variables of pattern. In other words, lazy rewriting steps always end by

*applying the Replacement rule and hence, lazy rewriting derivations only* include terms in GDinit.

*4.1 Thunki cation Description*

*Thunki cation extends the signature and generates a new TRS by which in-* nermost rewriting simulates lazy rewriting in the original TRS.

*The new signature 0 is built from the original signature = (V; F) by* adding new function symbols introduced during thunki cation: ; f ; vecf ; vect; t; inst for every f 2 F and for some subterms t of the RHSs of rewrite rules in the original TRS. The introduction of new function symbols allows one to mask lazy subterms. A lazy f -rooted subterm s is masked (or thunked) by a subterm in the form of ( f ; vecf (:: :)) and hence, cannot eagerly be rewritten. The structure of s is stored in this -rooted subterm so that one can recover it later.

*The thunki cation of terms is a mapping ' : G ! G0 which is de-*

*Dinit*

*ned by the rules in gure* [*4.*](#_bookmark10) *We describe now the new TRS generated by* thunki cation.

*De nition 4.1 (Lazy argument position and subterm) Let t be a term* in G. If there exist p 2 FPos(t) and i 2 N such that Head(tjp) = f and

*f = l, then p:i is called a lazy argument position in t while tj*

*i*

*p:i*

*is called a*

*lazy argument subterm of t.*

*De nition 4.2 (Migrant variable* [*[8])*](#_bookmark31) *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 subterms which instantiate migrant variables are* changed from lazy to active after the lazy rewrite step. Thus, we need to activate lazy rewriting on these subterms later.

*De nition 4.3 (Set of rules) Let R be a TRS. The set of rewrite rules S* generated by applying thunki cation on R is the union of four subsets S0, S1, S2 and S3 which are de ned as follows:

*(i) S contains the rule l ! r0 if and only if l ! r 2 R and r0 is built from* r as follows:

*0*

*In a bottom-up fashion, replace any lazy argument subterm t of the RHS r by ( t; vect(x1;::: ; xnt )) where x1;::: ; xnt are all variables of t.*

*Replace any migrant variable x of the RHS r by inst(x).*

*(ii) S1 = finst( ( f ; vecf (x1;::: ; xar(f)))) ! f (t1;::: ; tar(f)) j f 2 Fg where*

*t = inst(x ) if f = e; otherwise t = x .*

*i i i i* *i*

*(iii) S*

*= finst( ( ; vec (x ;::: ;x*

*))) ! t0 j t has been replaced in* [*(i*](#_bookmark9)*) and*

*2 t t 1 nt*

*t0 = tfxi 7! inst(xi)g8i such that xi is a migrant variable of tg.*

*(iv) S3 = finst(x) ! xg.*

*In fact, S0 contains all rewrite rules in R whose RHS has been changed (or* thunked): every lazy argument subterm t is thunked by a subterm in the form of ( t; vect(:: :)) and hence, t cannot eagerly be rewritten. A corresponding rule is then inserted into S2 in order to recover t later. The insertion of the symbol inst allows rewriting afterwards on the subterms which have instanti- ated migrant variables. The unique rule of S3 allows dealing with the direct subterms which are not thunked of symbol inst. This rule has the lowest priority and hence, is the last rule of S to be tried with terms since we use textual ordering.

*In* [*[8],*](#_bookmark31) *only non-variable lazy argument subterms of RHSs are thunked.* Since an innermost strategy will be used for rewriting by S, the subterms which instantiate variables of RHSs are always in NF before the application of any rule. In other words, the thunki cation of lazy argument subterms which are variables is unnecessary. However, in this work, we also thunk these subterms in order to ensure the correctness of lemma [5.1](#_bookmark15) in section [5](#_bookmark14).

*(i) '(f (t ;::: ;t )) !7! f ('( (t ; p:1; ));::: ; '( (t ; p:n; )))*

*a*

*f*

*f*

*p*

*1*

*n*

*1*

*1*

*n*

*n*

*(ii) '(f l(t1;::: ; tn)) !7! ( f ; vecf ('(t1l );::: ; '(tnl )))*

*p*

*p:1*

*p:n*

*(iii)*

*(iv)*

*'(ca) !7! c if c is a constant.*

*p*

*'(cl ) !7! ( c; vecc) if c is a constant.*

*p*

*Fig. 4. Evaluation rules for '*

*The set of terms B is de ned as follows:*

*Dinit*

*B = fg 2 G0 j 9g0 2 G : '(g0) !S gg*

*This de nition of B is slightly di erent from* [*[8]*](#_bookmark31) *where g0 is not thunked (by* '). The thunki cation of g0 helps to get NFs w.r.t. S more quickly. This fact is used in our normalisation procedure in section [5](#_bookmark14).

*The mapping : B ! GDinit relates terms in B and terms in GDinit and is*

*de ned by the rules in gure* [*5.*](#_bookmark11) *Actually, recovers lazy subterms using the* informations stored in their corresponding -rooted subterms.

*4.2 Correctness of Thunki cation*

*Lazy rewriting on terms in GDinit*

*w.r.t. R can correctly be simulated by*

*innermost rewriting on terms in the subset B of G0 w.r.t. S via up to* the criteria gured in [[12](#_bookmark35)]. That is, is surjective, sound, complete and termination preserving. The mapping is surjective since for every term g in

*(i) (g) !7! (g; ; e)*

*(ii) (inst(t); p; e) !7! (t; p; e)*

*(iii) (inst(t); p; l) !7! (t; p; l)*

*(iv) ( ( ; vec (t ;::: ;t )); p; e) !7! f a( (t ; p:1; f );::: ; (t ; p:n; f ))*

*f f* *1*

*n*

*p*

*1*

*1*

*n*

*n*

*(v) ( ( f ; vecf (t1;::: ; tn)); p; l) !7! f l( (t1; p:1; l);::: ; (tn; p:n; l))*

*p*

*(vi) ( ( c; vecc); p; e) !7! ca if c is a constant.*

*p*

*p*

*(viii) ( ( t; vect(t1;::: ; tnt )); p; e) !7! (tfx1 7! t1g ::: fxnt 7! tnt g; p; e)*

*(ix) ( ( t; vect(t1;::: ; tnt )); p; l) !7! (tfx1 7! t1g ::: fxnt 7! tnt g; p; l)*

*(vii) ( ( c; vecc); p; l) !7! cl if c is a constant.*

*(x) (f (t ;::: ;t ); p; e) !7! f ( (t ; p:1; );::: ; (t ; p:n; ))*

*a*

*f*

*f*

*1*

*n*

*p*

*1*

*1*

*n*

*n*

*(xi) (f (t ;::: ; t ); p; l) !7! f ( (t ; p:1; l);::: ; (t ; p:n; l))*

*l*

*1*

*n*

*p*

*1*

*n*

*(xii) (c; p; e) !7! ca if c is a constant.*

*p*

*(xiii) (c; p; l) !7! cl if c is a constant.*

*p*

*Fig. 5. Evaluation rules for*

*Dinit*

*G*

*: ('(g)) = g. In the following, !S*

*denotes the innermost rewriting*

*relation w.r.t. S.*

*Theorem 4.4 (Soundness* [*[8])*](#_bookmark31) *Let g be a term in B. If g ! g0, then*

*S*

*R*

*S0*

*(g) ;*

*R*

*(g0). More precisely: if g !*

*g0, then (g) ;*

*(g0) and if*

*g ! g0, then (g) = (g0).*

*S1[S2 [S3*

*Lemma 4.5* [*([8])*](#_bookmark31) *If g 2 B contains no symbol inst, then every active sub-* term of (g) inherits the head symbol from its corresponding subterm of g.

*Theorem 4.6 (Completeness* [*[8])*](#_bookmark31) *If g 2 B is in NF w.r.t. S, then (g) is* in LNF w.r.t. R

*Theorem 4.7 (Termination preservation* [*[8])*](#_bookmark31) *If there exists an in nite* derivation g0 !S g1 !S :: :, then there exists k 2 N such that (g0) ;R (gk).

*Corollary 4.8 If lazy rewriting w.r.t. R is terminating, then so is innermost* rewriting w.r.t. S.

*4.3 Correspondence of Trace*

*We show in this section that (lazy) normalisation traces of (g) w.r.t. R can* be extracted from normalisation traces of g w.r.t. S.

*Suppose that each rule in S0 inherits the label from its corresponding rule* in R, we have:

*Theorem 4.9 (Correspondence of trace) Assume that TS is a normal-*

*g*

*isation trace of term g 2 B w.r.t. S in an innermost reduction strategy.*

*Extracting from TS the traces of the rewrite steps performed by rewrite rules*

*g*

*(g)*

*in S0*

*yields a (lazy) normalisation trace TR*

*of (g) w.r.t. R.*

## *5 Normalisation Procedure*

*A term can be normalised by sequentially reducing all its subterms into HNF.* Suppose that we need to normalise a term t by a left-linear and terminating TRS R. The thunki cation process is rst applied on R to get the TRS S. Next, t is thunked and normalised w.r.t. S to get g as a NF. Due to the rule in S3, g contains no symbol inst. Completeness implies that (g) is in LNF

*w.r.t. R. In other words, all active subterms of (g) are in HNF w.r.t. R and*

*inherit the head symbol from the corresponding subterms of g (lemma* [*4.5*](#_bookmark12)*).* Furthermore, in (g), active subterms are never subterms of lazy subterms. In other words, (g) can be divided into two parts: the upper part contains active subterms while the lower part contains lazy subterms. Hence, the upper part of g contains the subterms which correspond to active subterms of (g) and which are in HNF w.r.t. R. The lower part of g correspond to the lazy subterms of (g). The frontier between these two parts is composed of symbols

*(lemma* [*5.1).*](#_bookmark15)

*Thus, we can unthunk (activate) -rooted subterms and reduce them into* NF w.r.t. S. By this reduction, some more subterms of g become in HNF

*w.r.t. R. Notice that if a -rooted subterm is activated, then its \active"* subterms are also unthunked. The activating procedure of -rooted subterms will be described later by operator . The process is recursively applied until all subterms of g are in HNF w.r.t. R and g is a NF of t.

*Lemma 5.1 Let g be a term in B and g contains no symbol inst. Then g is* divided into two parts. The upper part contains the subterms which correspond to active subterms of (g) while the lower part contains the subterms which correspond to lazy subterms of (g). The frontier between these two parts is composed of symbols .

*Let g be a term in G0 . We de ne the set of disjoint -ancestor positions* of g as follows:

*Pla(g) = fpj p 2 FPos(g); Head(gjp) = and Head(gjp1 ) 6=*

*for any pre x p1 of pg*

*Pla(g) can be computed by the rules in gure* [*6.*](#_bookmark16) *Intuitively, Pla(g) contains* the frontier between two parts of g. The activating operator is a mapping from G0 to G0 and is de ned by the rules in gure [7:](#_bookmark17) activates (unthunks) a -rooted term g and every -rooted subterm s of g such that (s) is an active subterm of (g). Figure [8](#_bookmark20) describes the normalisation procedure based on lazy rewriting (lazynorm(t; R)).

*Initialisation Pla(g) !7! La(g; )*

*Symbol La(f (t1;::: ; tn); p) !7! La(t1; p:1) [ ::: [ La(tn; p:n)*

*if f 2 F*

*Constant La(c; p) !7! ; if c is a constant Discovery La( (t1; t2); p) !7! fpg*

*Fig. 6. Evaluation rules for Gla(t)*

*(i) ( ( ; vec (t ;::: ;t ))) !7! f ( (t ; );::: ; (t ; ))*

*f*

*f*

*f f* *1*

*n*

*1*

*1*

*n*

*n*

*(ii) ( ( ; vec (t ;::: ;t ))) !7! tfx 7! t g ::: fx*

*t t* *1*

*nt*

*1 1*

*nt nt*

*7! t g*

*(iii)*

*(iv)*

*(v)*

*( ( ; vec )) !7! c if c is a constant*

*c c*

*(t; e) !7! (t)*

*(t; l) !7! t*

*Fig. 7. Evaluation rules for*

*Theorem 5.2 If R is terminating and ful lls all necessary conditions for* thunki cation, then lazynorm(t; R) is also terminating and yields a NF of t

*w.r.t. R.*

*Remark 5.3 The normalisation of term t by procedure lazynorm(t; R) gen-* erates a trace Tt containing the traces of all performed (leftmost-innermost) rewrite steps. Let us extract from Tt the pairs whose rst element is the label of some rule in S0. Due to theorem [4.9](#_bookmark13), this process yields a normalisation trace TR of t in R (in the sense of normal rewriting).

*t*

## *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 LHSs. Our transformation works on (left-linear) constructor-based TRSs. It is proved to be correct and to preserve a good correspondence between nor- malisation traces in original and transformed TRSs.

*procedure normalise(g 2 G0 ; S)*

*(i) g is normalised in leftmost-innermost strategy w.r.t. S to get gnf*

*(ii) if gnf contains no symbol then*

*return gnf*

*else*

*for all p 2 Pla(gnf ) do*

*s := ((g )j )*

*nf p*

*(gnf )jp := normalise(s; S)*

*end for*

*procedure lazynorm(t 2 G, R)*

*(i)*

*(ii)*

*(iii)*

*Build S = S0 [ S1 [ S2 [ S3 from R*

*g := '(DC(t))*

*tnf := normalise(g; S)*

*(iv) return tnf*

*Fig. 8. Normalisation procedure (lazynorm) based on lazy rewriting*

*6.1 Transformation Description*

*Let 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 ls ! r 2 R and Head(lsjp) = f . We activate this position by adding a new function symbol

*p p*

*f*

*f p of arity ar(f ) where e = f 8 6= while f*

*j i e*

*= e , and by transforming*

*e j j* *i*

*ls ! r which is called the source rule as follows:*

* *Replace it by the rule lt ! r where lt is ls but Head(ltjp) = f p. This rule*

*e*

*is called the transformed rule.*

* *Add a new rule ls[x]p:i ! lt[x]p:i where x is a fresh variable to R such that this rule has the lowest priority in case of overlapping. This rule is called the added rule.*

*All other rules of R are unchanged. This process is called a transformation* step that eliminates one non-variable lazy argument subterm of the LHS of a rule in R.

*Example 6.1 Consider again the TRS in example* [*3.4*](#_bookmark4)*. Applying the trans-*

*formation on the rule r1 (source rule) yields the following TRS:*

# *><8>*

*:>>*

*S =*

*[rt] 2nd(cons1(x; cons(y; z))) ! y (Transformed rule) [r2] inf (x) ! cons(x; inf (s(x)))*

*[r ] 2nd(cons(x; x0)) ! 2nd(cons1(x; x0 )) (Added rule)*

*e*

*a*

*e*

*where L(2nd) = hei; L(inf ) = hei; L(cons) = he; li; L(cons1) = he; ei.*

*e*

*Denote by S the new TRS generated by one transformation step. Let 0* be the new signature ( 0 = (V; F [ ff pg)). The set of terms B is de ned as

*e*

*follows:*

*B = fg 2 GDterm j 9g*

*0*

*0*

*2 GDterm : g*

*0*

*; gg*

*The mapping 0 : B ! GDterm relates terms in B with terms in GDterm: 0(g)*

*S*

*is built by replacing every symbol f p in g by f . Furthermore, the laziness*

*e*

*annotations of subterms of g and 0(g) are kept identical.*

*We call 0(g) the simulation of lazy rewriting on terms in GDterm w.r.t. R* by lazy rewriting on terms in B w.r.t. S. Obviously, 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 in each step, the number of non-variable lazy argument subterms of LHSs is strictly decreased.

*6.2 Correctness of Preliminary Transformation*

*The correctness of the preliminary transformation can be deduced from the* correctness of one transformation step up to the criteria gured in [[12](#_bookmark35)]. The mapping 0 is obviously surjective, since it is the identity mapping on the subset GDterm of B.

*Theorem 6.2 (Soundness) Let g be a term in B. If g ;S g0 then 0(g) ;R*

*0(g0). More precisely: if g ;S g0 by applying the added rule or the transformed* rule, then 0(g) ;R 0(g0) by applying the source rule at the same position. Otherwise, 0(g) ;R 0(g0) by applying the same rule at the same position.

*Remark 6.3 If g ;S g0 by a rewrite step using added rule, then UD( 0(g)) =* UD( 0(g0)). Hence, if we are only interested in non-decorated terms as in case of normal rewriting, then this step is redundant.

*Theorem 6.4 (Completeness) If g 2 B is in LNF w.r.t. S, then 0(g) is* in LNF w.r.t. R.

*Corollary 6.5 (Correspondence of trace) Let Tg be a (lazy) normalisa-* tion trace of term g 2 B w.r.t. S. Replacing the labels of added rule and transformed rule in Tg by the label of source rule, yields a (lazy) normalisa- tion trace of 0(g) w.r.t. R.

*Example 6.6 In example* [*6.1*](#_bookmark21)*, term t = 2nd(inf (0)) is normalised w.r.t. S as follows: 2nd(inf (0)) r2;1 2nd(cons(0; inf (s(0))) ra; 2nd(cons1(0; inf (s(0)))*

*;*

*r2;1:2* *1*

*;*

*e*

*rt;*

*; 2nd(conse(0; cons(s(0); inf (s(s(0)))))) ; s(0).*

*In the generated trace TS = fhr2; 1i; hr ; i; hr2; 1:2i; hr ; ig, replacing*

*t a t*

*R*

*rt and ra by r1 yields a (lazy) normalisation trace of t w.r.t. R : T =*

*t*

*fhr2; 1i; hr1; i; hr2; 1:2i; hr1; ig*

*Theorem 6.7 (Termination preservation) If there exists an in nite deri-* vation g ; g ; :: :, then there exists k 2 N such that 0(g ) ; 0(g ).

*0 S 1 S 0 R k*

## *7 Combining Two Transformations*

*We describe in this section, the combination of thunki cation and preliminary* transformation described above. If the LHSs of the considered TRS (R) con- tain no non-variable lazy subterms, then sole thunki cation is suÆcient. In order to get a normalisation trace of term t, we use the normalisation proce- dure described in section [5.](#_bookmark14) Otherwise, preliminary transformation is used to eliminate non-variable lazy subterms of the LHSs. The new TRS (S) gener- ated by this transformation is then, transformed by thunki cation. Suppose that the normalisation procedure yields a trace Tt . Due to remark [5.3](#_bookmark18), one

*S*

*can extract from Tt the trace T of corresponding (lazy) derivation by S. Re-*

*t*

*placing added rules and transformed rules in S by their source rules in R, one*

*gets TR which is the trace of corresponding (lazy) derivation by R.*

*t*

*Nevertheless, due to remark* [*6.3,*](#_bookmark22) *rewrite steps by added rules are redundant* since our goal is to get a normalisation trace in the sense of normal rewriting. Therefore, we need to re ne our trace by eliminating these redundant steps.

*This re nement should be done on TS before generating TR which is now the*

*t*

*t*

*normalisation trace of t w.r.t. R.*

*Example 7.1 We illustrate our method by considering the TRS (R) in ex-* ample [3.4](#_bookmark4). Thunki cation cannot directly be applied on R since the LHS of r1 contains non-variable lazy subterm cons(y; z). Using preliminary transfor- mation, we get the TRS S in example [6.1.](#_bookmark21) This TRS ful lls all necessary

*conditions for thunki cation which will give the following TRS:*

# *>8 >8*

*[rt] 2nd(cons1(x; cons(y; z))) ! y*

* *>[r2] inf (x) !*

*e*

*>U0 = <>*

*>*

*a*

* *[r*

*>*

*cons(x; ( inf(s(x); vecinf(s(x))(x))))*

*] 2nd(cons(x; x0)) ! 2nd(cons1(x; inst(x0 )))*

*e*

# *>8*

*:*

*[r11] inst( ( cons; veccons(x; y))) ! cons(inst(x); y)*

*U = <*

* *U1*

*= <>[r12] inst( (*

*[r13] inst( (*

*inf*

*; vec*

*; vec*

*inf*

*(x))) ! inf (inst(x))*

*(x))) ! 2nd(inst(x))*

*2nd*

*> >*

*2nd*

*> :>>*

*[r14] inst( ( cons1 ; veccons1 (x; y))) ! cons (inst(x); inst(y))*

*e e e*

*>*

*1*

* *U2 = n [r21] inst( ( inf(s(x)); vecinf(s(x))(x)))) ! inf (s(x))*

*:>U3 = n [r31] inst(x) ! x*

*Consider the term t = 2nd(inf (0)). We normalise '(t) = 2nd(inf (0)) w.r.t.* U by the following leftmost-innermost derivation:

*r2;1 ra;*

*2nd(inf (0)) ! 2nd(cons(0; ( inf(s(0); vecinf(s(0))(0))))) !*

*2nd(cons1(0; inst( (*

*e*

*inf (s(0)*

*; vec*

*inf (s(0))*

*r21;1:2*

*(0)))))) !*

*2nd(cons1(0; inf (s(0)))) r2;1:2*

*e !*

*rt;*

*2nd(cons1(0; cons(s(0); (*

*e*

*inf (s(0)*

*; vec*

*inf (s(0))*

*(s(0))))))) ! s(0).*

*Since s(0) contains no symbol the normalisation procedure nishes and* return this term as a NF of t w.r.t. S. Due to the soundness of preliminary transformation, s(0) is also a NF of t w.r.t. R. Thanks to theorem [4.9](#_bookmark13), one can extract from the normalising derivation above a normalisation trace

*of t w.r.t. S: TS = fhr2; 1i; hr ; i; hr2; 1:2i; hr ; ig (only the rewrite steps*

*t a t*

*S*

*performed by rules in U0 gure in T ). Finally, we eliminate the rewrite steps*

*t*

*by added rules (ra) and replace transformed rules (rt) by their source rules (r1)* to get a normalisation trace of t w.r.t. R (in the sense of normal rewriting):

*TR = fhr2; 1i; hr2; 1:2i; hr1; ig. Notice that applying an innermost strategy*

*t*

*on t using the rules in R leads to in nite reductions.*

## *8 Related Work*

*Lazy rewriting can be obtained in OBJ* [*[9]*](#_bookmark32) *and CafeOBJ* [*[6]*](#_bookmark29) *using 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:*

*(i) omit lazy arguments from local strategy of its function symbol*

*(ii) use negative integers for these arguments*

*The rst method is not well-behaved if there is some non-variable lazy* subterm in the LHS of a rule as in example [3.4,](#_bookmark4) where the second argu- ment is omitted from the local strategy of cons. However, such a strategy reduces 2nd(inf (0)) to 2nd(cons(0; inf (s(0)))) instead of s(0) since the sub- term inf (s(0)) is not allowed to be reduced and r1 cannot be applied.

*The second method is implemented in CafeOBJ using on-demand ag* [*[18].*](#_bookmark41) A negative integer i in the local strategy of function symbol f means the ith subterm of f is forced to be rewritten if and only if it causes a con ict during pattern matching. In example [3.4,](#_bookmark4) the local strategy of cons is (1 2 0) and

*r2;1*

*2nd(inf (0)) is derived as follows: 2nd(inf (0)) ! 2nd(cons(0; inf (s(0))))*

*r2;1:2*

*!*

*2nd(cons(0; (cons(s(0); inf (s(s(0)))))))*

*r1;*

*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 a con ict and hence, it is forced to be rewritten. The E-strategies that can reduce terms to their HNF is characterised in [[15]](#_bookmark38) for left-linear and constructor-based TRSs. On-demand ag is very similar to the notion of essential node and thunki cation shares the same limit with the rst method described above. Preliminary transformation allows us to overcome this limit for left-linear and constructor-based TRSs.

*Context-sensitive rewriting* [*[14*](#_bookmark37)*] can be seen as a restricted case of lazy* rewriting where subterm activation is not allowed. In order to correctly sim- ulate rewriting by context-sensitive rewriting, one needs to use canonical re- placement maps which actually require that all lazy subterms of the LHSs must be variables. In other words, context-sensitive rewriting also shares the same restriction with the rst method described above.

## *9 Conclusion*

*In this paper, we described lazy rewriting and the mechanism of thunki ca-* tion under a rule-based form. We showed the relation between normalising derivations in TRSs before and after thunki cation and proposed a normali- sation procedure based on lazy rewriting. A preliminary transformation that allows extending the application scope of thunki cation while preserving a nice correspondence between normalisation traces was also presented.

*Finding optimal derivations is undecidable in general* [*[16]*](#_bookmark39)[*[11]*](#_bookmark34) *and even* when it is decidable, the decision procedures are often diÆcult to implement. In practice, most of interesting results only involve orthogonal constructor- based TRSs [[20]](#_bookmark43) [[2]](#_bookmark25) [[21].](#_bookmark44) We think that our normalisation procedure is helpful since the normalisation procedure is reasonably eÆcient in ELAN, thanks to correct simulations, while generated traces are more compact and still useful for Coq, thanks to the nice correspondences between normalising derivations before and after each transformation. Moreover, TRSs are allowed to be over- lapping.

*A natural question may arise: which arguments should be marked lazy in*

*each function symbol ? There is not already 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

*fif (true; x; y) ! x; if (f alse; x; y) ! yg*

*the two last arguments of if should be lazy. Such TRSs form a class where* lazy rewriting can provide more compact normalisation traces. 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, variables marked lazy should not appear more than once in the RHS since this duplicates reductions on terms which will instantiate these variables. 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. This requires some extensions in Coq replaying procedure and ELAN compiler that we are investigating.

## *10 Acknowledgements*

*I sincerely thank Claude Kirchner, H el ene Kirchner and some anonymous* referees for their constructive comments on the earlier versions of this paper. I am also grateful to Mark van den Brand for pointing out [[8]](#_bookmark31) to me.

## *References*

*[1] C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in Coq. In J. Despeyroux, editor, Proc. of 2nd Workshop on Logical Frameworks and Metalanguages. Institut National de Recherche en Informatique et en* *Automatique, ISBN 2-7261-1166-1, June 2000.*

*[2] S. Antoy. De nitional trees. In H. Kirchner and G. Levi, editors, Proc. of the 3rd International Conference on Algebraic and Logic Programming, volume 632 of Lecture Notes in Computer Science, pages 143{157. Springer-Verlag, September* *1992.*

*[3] P. Borovansky , 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*](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, Proc. of 2nd International Workshop on Rewriting Logic*

*and its Applications, volume 15 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B. V. (North-Holland), 2000. Available at* [*http://www.elsevier.nl/locate/volume15.html*](http://www.elsevier.nl/locate/volume15.html)*.*

*[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, J.M. Winkler, J. Meseguer, K. Futatsugi, and J-P. Jouannaud. An introduction to OBJ. In J A. Goguen and G. Malcolm, editors, Software engineering with OBJ, Advances in Formal Methods. Kluwer Academic* *Publishers, 2000.*

*[10] A. Graf. Left-to-right tree pattern matching. In Ronald V. Book, editor, Proc. 4th Int. Conf. RTA, volume 488 of Lecture Notes in Computer Science, pages* *323{334. Springer-Verlag, 1991.*

*[11] G. Huet and J-J. L evy. 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 Speci cation, volume 1130 of Lecture Notes in Computer Science, pages 274{*

*290. Springer-Verlag, 1995.*

*[13] LogiCal/INRIA. The Coq homepage.* [*http://coq.inria.fr*](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 ag. In K. Futatsugi, editor, Proc. of 3rd International Workshop on Rewriting Logic and its Applications, volume 36 of Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers B. V. (North-Holland), 2000. Available at* [*http://www.elsevier.nl/locate/volume36.html*](http://www.elsevier.nl/locate/volume36.html)*.*

*[16] M.J. O'Donnell. Equational logic programming. In D. Gabbay, editor, Handbook of Logic in Arti cial 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*](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 Proc. of ACM Symposium on Applied Computing, pages 756{763, 2000.*

*[19] PROTHEO/LORIA. The ELAN homepage.* [*http://elan.loria.fr*](http://elan.loria.fr/)*.*

*[20] R I. Strandh. Classes of equational programs that compile into eÆcient machine code. In N. Dershowitz, editor, Proc. of the 3rd Int. Conf. RTA, volume* *355*

*of Lecture Notes in Computer Science, pages 449{461. Springer-Verlag, April* *1989.*

*[21] S. Thatte. A re nement of strong sequentiality for term rewriting with constructors. Information and Computation, 72(1):46{65, January 1987.*