Lazy Rewriting in Logic Programming[*]

Sergio Antoy[†]

Department of Computer Science
Portland State University
P.O. Box 751
Portland, Oregon 97207
`antoy@cs.pdx.edu`

Abstract

We describe a technique enabling a logic program to perform lazy rewriting. Our technique is based on a transformation of a rewrite system in a set of Horn clauses. We characterize syntactically, in terms of a hierarchical structure of the rewrite rules of an operation, the systems accepted by our transformation and outline a design technique yielding systems with these characteristics. We define our transformation, prove its correctness and other properties, and relate the efficiency of resolution to that of rewriting. Improvements over previous similar results include using a more expressive and more powerful language, generating a more efficient logic program, providing efficient operational completeness, and establishing a tight bound on the length of a resolution as a function of the length of a corresponding reduction sequence. We compare our approach with several related proposals and discuss examples which also show the integration of lazy with eager evaluation and the possibility of narrowing in addition to rewriting.

1 Introduction

Lazy evaluation is a computational strategy which ensures that expressions are evaluated only when needed. This strategy is appealing because it makes efficient use of resources and, more interesting, supports a programming style that improves modularity and allows the use of infinite structures. Hudak [24] includes lazy evaluation in a list of fundamental attributes of modern functional programming languages. This computational strategy may be convenient in logic as

---

[*] Parts of these results appear in preliminary form and without proofs in the Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming.

[†] This material is based upon work supported by the National Science Foundation Grant No. CCR-8908565.

well, and various proposal [2, 4, 5, 6, 9, 15, 18, 35, 37, 38, 39, 40, 41, 50] have been made to integrate it with logic programming.

Most proposals aiming at the integration of functional computations in logic programming are based on rewriting. We follow this approach by proposing a translation of a term rewriting system in a logic program. A property of our translation is that, for a class of rewrite systems large enough to be of practical interest, a successful refutation via the logic program is equivalent, in a sense to be made precise later, to a sequential lazy normalization via the rewrite system. This strategy, pioneered by Narain [39], is a component of several recent systems, including [9, 37, 38, 40, 41]. These systems adopt an approach referred to as *indirect*, since they first map rewrite rules to Horn clauses and then execute the resulting program via a Prolog interpreter or compiler. The *direct* approach, instead, relies on some *ad hoc* processor often based on a specialized *WAM*-like machine [56]. The indirect approach simplifies and speeds up the implementation and takes full advantage of the speed, reliability, and portability of Prolog interpreters and compilers, including commercial products.

Rewriting [12, 29] is a computational paradigm that specifies which reductions can be performed but not where and when one should execute them during a computation. Computing requires some form of control referred to as a *strategy* [25, 29, 30, 42, 43]. Without an appropriate strategy a computation may fail to terminate even when termination is possible. We are concerned with a strategy, generally called *lazy*, that ensures that a computation terminates, if at all possible. Its foundations are based on the seminal work of Huet and Lévy and O'Donnell [25, 42, 43]. In a certain class of systems, one can find in every term t, not in normal form, a redex, called *needed*, that must be eventually reduced to compute the normal form of t. Furthermore, if each needed redex is eventually reduced, the computation of a term with a normal form terminates. The strategy implicitly implemented by our approach reduces only needed redexes. Under certain additional conditions [34, 53], this strategy yields a reduction sequence of minimum length.

Our approach to lazy rewriting improves several aspects of the indirect techniques references above. In particular, we remove certain common restrictions of the indirect approaches, such as the flattening of functional nestings, which are potentially inefficient, we outline how an exception-like mechanism for incomplete operations can be easily included in the logic programs, and we establish a simple relation between the length of a reduction sequence and that of the corresponding resolution, which shows the efficiency of our method.

In Section 2 we define the class of systems for which our method is applicable and review the notation and a few basic concepts used in this note. In Section 3 we describe the translation of a set of rewrite rules into a set of Horn clauses. In Section 4 we prove some properties of this translation. In Section 5 we describe the reduction of a term via the clauses obtained by the above translation and prove some properties of this computation. In Section 6 we discuss some practical problems arising in the application of our results. In Section 7 we discuss related work and compare our method with others having similar goals. Section 8 contains our conclusions. The Appendix presents some significant examples.

2 Preliminaries and Notations

This section characterizes the class of rewrite systems we consider. The characterization is based on a strategy for the design of rewrite rules introduced in [1]. First, we recall the terminology and notation relative to rewriting used throughout this note. Then, we review the design strategy for rewrite rules at the core of our approach.

2.1 Term Rewriting Systems

We consider a term rewriting system $\mathcal{R}$ with a many-sorted signature partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{D}$ of *(defined) operations*. For each sort s, we assume an arbitrary but fixed ordering, called *standard*, among the constructors of s. The rules of $\mathcal{R}$ are characterized below.

$\mathcal{X}$ denotes a set of sorted variables. Variables are denoted by upper case letters or, when anonymous, by the symbol "_". "$\Box$" denotes a nullary, sort-overloaded symbol called *place* distinct from constants and variables. For any signature Σ, $T(\Sigma)$ is the set of terms built over Σ. Any term referred to in this note type checks. The leading symbol or principal functor of a term t is called *root* of t. The elements of $T(\mathcal{C})$, $T(\mathcal{C} \cup \mathcal{X})$, $T(\mathcal{C} \cup \mathcal{D})$, and $T(\mathcal{C} \cup \mathcal{D} \cup \mathcal{X} \cup \{\Box\})$ are respectively called *values*, *constructor terms*, *ground terms*, and *templates*. A term t of the form $f(x_1, \ldots, x_n)$, where f is an operation and the arguments are constructor terms, is called *f-rooted constructor term*. The rules in $\mathcal{R}$ are characterized by having an operation-rooted constructor term as the left side, i.e., $\mathcal{R}$ follows the constructor discipline [43]. An additional constraint on $\mathcal{R}$ is introduced at the end of this section.

An *occurrence* is a path identifying a subterm in a term. An occurrence is a possibly empty string of dot-separated integers or occurrences, i.e., the symbol "$\cdot$" is overloaded. The empty occurrence of a term t is denoted by Λ and identifies t itself. The occurrence $p{\cdot}i$, for some occurrence p and positive integer i, identifies the i-th argument of the subterm of t identified by (at) p. For terms t and t', occurrence p, and variable X, t/p denotes the subterm of t at p; $t[p \leftarrow t']$ denotes occurrence substitution; and $\{t'/X\}t$ denotes variable substitution. If α is a reduction sequence, then $|\alpha|$ denotes the number of reduction steps in α; if t is a term, then $|t|$ denotes the number of occurrences in t. More specific notation will be introduced where needed.

2.2 Binary Choice Strategy

In the next section we will be concerned with the possibility of storing the set of rules defining any operation of $\mathcal{R}$ into a tree-like structure. This structure, which may appear complicated at first, is quite natural for a large class of rewrite systems, if we regard it as the result of a well-organized, systematic approach to the design of rewrite rules. This technique is embodied by the *binary choice strategy* [1], an interactive, iterative, non-deterministic procedure that constructs a set of tuples of terms through a set of binary choices. We give first an intuitive description of the binary choice strategy, then its formal definition in the form of pseudo-code, and finally some examples of its use.

The left side of each rule defining an operation f has the form $f(\square, \ldots, \square)$, where each place is either a variable or a constructor term. A systematic approach to the generation of the set of left sides of the rules defining f consists in replacing each template's place either with a variable or with all the members of "appropriate sets" of constructor terms in turn. The problem is that we do not want to forget some combination of arguments, or include other combinations twice, in our rules. Thus, we devise an approach to achieve this goal. Our definitional process becomes a sequence of choices for replacements that terminates when all places have been replaced. This process is more easily described and analyzed if we consider tuples of constructor terms rather than rules' left sides.

Procedure *Binary Choice*

Input:	$\mathcal{S}$:	a set of sorts,
	$\mathcal{C}$:	a signature of constructors,
	$s_1, \ldots, s_k$:	a sequence of sorts.
Output:	R:	a set of k-tuples of constructor terms.

1. [Initialize] Initialize R to a set consisting of a single k-tuple whose components are all places. Let s_i be the sort of the i-th place, for $1 \leq i \leq k$.

2. [Test] If there are no occurrences of places anywhere in R, then halt with output R.

3. [Select] Select a place in some tuple t of R. Let u be the occurrence of the selected place and s its sort.

4. [Choose] Choose either "variable" or "inductive" for the place at u.

5. [Update] If the binary choice is "variable", then replace in t the place at u with a fresh variable, else remove t from R and for each constructor c, of arity $s_1 \times \ldots \times s_n \to s$, for some $s_1, \ldots, s_n$ in $\mathcal{S}$, insert in R a tuple obtained by replacing in t the place at u with the term $c(\square, \ldots, \square)$ in which there are exactly n places and the sort of the i-th place, for $1 \leq i \leq n$, is s_i.

6. [Iterate] Go to step 2.

For designing rewrite rules, the binary choice procedure is applied to a set of templates rather than to a set of tuples. The difference is only syntactic. Given an operation symbol f of arity $s_1 \times \ldots \times s_n \to s$, the initial template is $f(\square, \ldots, \square)$ where there are exactly n places and the sort of the i-th place, for $1 \leq i \leq n$, is s_i. Figure 1 shows a partial trace of execution of the binary choice procedure for the definition of *append*, an operation to concatenate lists. The constructors of the sort *list* are (in standard ordering) [] and the mixfix symbol [...|...]. Usually, as soon as we replace the last place from a term l, we use l as the left side of a rule and go on defining its right side. For simplicity's sake, this step in not shown in Figure 1.

Throughout the remainder of this note, unless otherwise stated, we assume that the operations of any rewrite system $\mathcal{R}$ that we consider are designed with the binary choice strategy. We call *inductively sequential* any such system and we will give a declarative, rather than operational, definition of this concept in the next section. Inductively sequential systems satisfy several generally

The initial template, 1, results from the domain of *append*. The cursor is set by the user.

$$append(\Box,\blacksquare) \tag{1}$$

Template 2 comes from 1 with choice "variable".

$$append(\blacksquare, Y) \tag{2}$$

Templates 3 and 4 come from 2 with choice "inductive".

$$append([\,], Y) \tag{3}$$

$$append([\blacksquare|\Box], Y) \tag{4}$$

Template 6 comes from 4 with choice "variable", 5 is 3.

$$append([\,], Y) \tag{5}$$

$$append([E|\blacksquare], Y) \tag{6}$$

Template 8 comes from 6 with choice "variable", 7 is 5.

$$append([\,], Y) \tag{7}$$

$$append([E|X], Y) \tag{8}$$

Figure 1: Partial trace of the execution of the binary choice procedure for defining the operation *append* which concatenates two lists. At every iteration of the procedure, the user selects an occurrence of $\Box$, called *cursor* and denoted with $\blacksquare$, and makes a binary choice. After every choice, the set of templates is updated. For simplicity, the assignment of names to variables and the definition of the rules' right sides are omitted.

desirable properties [1]. As far as our discussion is concerned, any inductively sequential system $\mathcal{R}$ is *orthogonal*[1] [25, 30] and in particular $\mathcal{R}$ is *confluent* [26], and any ground operation-rooted constructor term is matched by one and only one rule of $\mathcal{R}$. The last two conditions are referred to as *completeness* and *parsimony* of $\mathcal{R}$ throughout this note.

3 Translation

In this section we describe an algorithm which generates a set $\mathcal{H}_{\mathcal{R}}$ of Horn clauses from the set of rules of an inductively sequential rewrite system $\mathcal{R}$. We begin with the definition of a particular representation of the rules of $\mathcal{R}$ that makes the algorithm's description terse and its execution efficient. The representation of the rules consists of a tree-like structure containing the selection and choices made during the design of the rules with the binary choice procedure. Except for a few additional clauses that we will introduce later, $\mathcal{H}_{\mathcal{R}}$ is the major component of a logic program implementing computations in $\mathcal{R}$ with a lazy strategy.

---

[1] In earlier work either "regular" or "left-linear and non-ambiguous" were used instead of "orthogonal."

3.1 Definitional Trees

If an operation f is designed using the binary choice strategy, then f's defining rules are naturally represented in a tree-like structure in which branches represent design decisions and leaves represent rewrite rules. This structure, called definitional tree [3], is formally defined below.

Definition 1 d is a *partial definitional tree*, or *pdt*, if and only if one of the following cases holds:

$d = rule(l, r)$, where l is a template in which there are no occurrences of places and r is a term whose variables are in l, too.

$d = var(t, o, v, d')$, where t is a template, o is the occurrence of a place in t, v is a fresh variable, and d' is a *pdt* such that the template in its root is $t[o \leftarrow v]$.

$d = ind(t, o, d_1, \ldots, d_k)$, where t is a template, o is the occurrence of a place in t, the sort of t/o has constructors $c_1, \ldots, c_k$ in standard ordering, and for all i, $1 \leq i \leq k$, d_i is a *pdt* such that the template in its root is $t[o \leftarrow c_i(\Box, \ldots, \Box)]$.

Rule, *var*, and *ind* are uninterpreted functions. Informally, a *rule* node of a *pdt* is a leaf containing a rule, while *var* and *ind* nodes are branches respectively representing "variable" and "inductive" choices of the binary choice strategy

Definition 2 A *definitional tree* is a pair $\langle f, d\rangle$ where f is an operation and d is a *pdt* such that the template in its root is $f(\Box, \ldots, \Box)$.

If $\langle f, d\rangle$ is a definitional tree, with some abuse of terminology, we call d a definitional tree of f. Note that templates do not need explicit representation in a definitional tree. However, we will keep them around when their presence simplifies the presentation of our ideas. For example, let us consider again the operation *append* discussed in Figure 1. The rules of *append* are:

$$\begin{array}{c} append([\,], Y) \to Y \\ append([E|X], Y) \to [E|append(X, Y)] \end{array} \tag{1}$$

The definitional tree resulting from the execution of the binary choice procedure discussed in Figure 1 is:

$$var(2, Y, ind(1, rule(Y), var(1{\cdot}1, E, var(1{\cdot}2, X, rule([E|append(X, Y)]))))) \tag{2}$$

Figure 2 shows the same tree in pictorial representation.

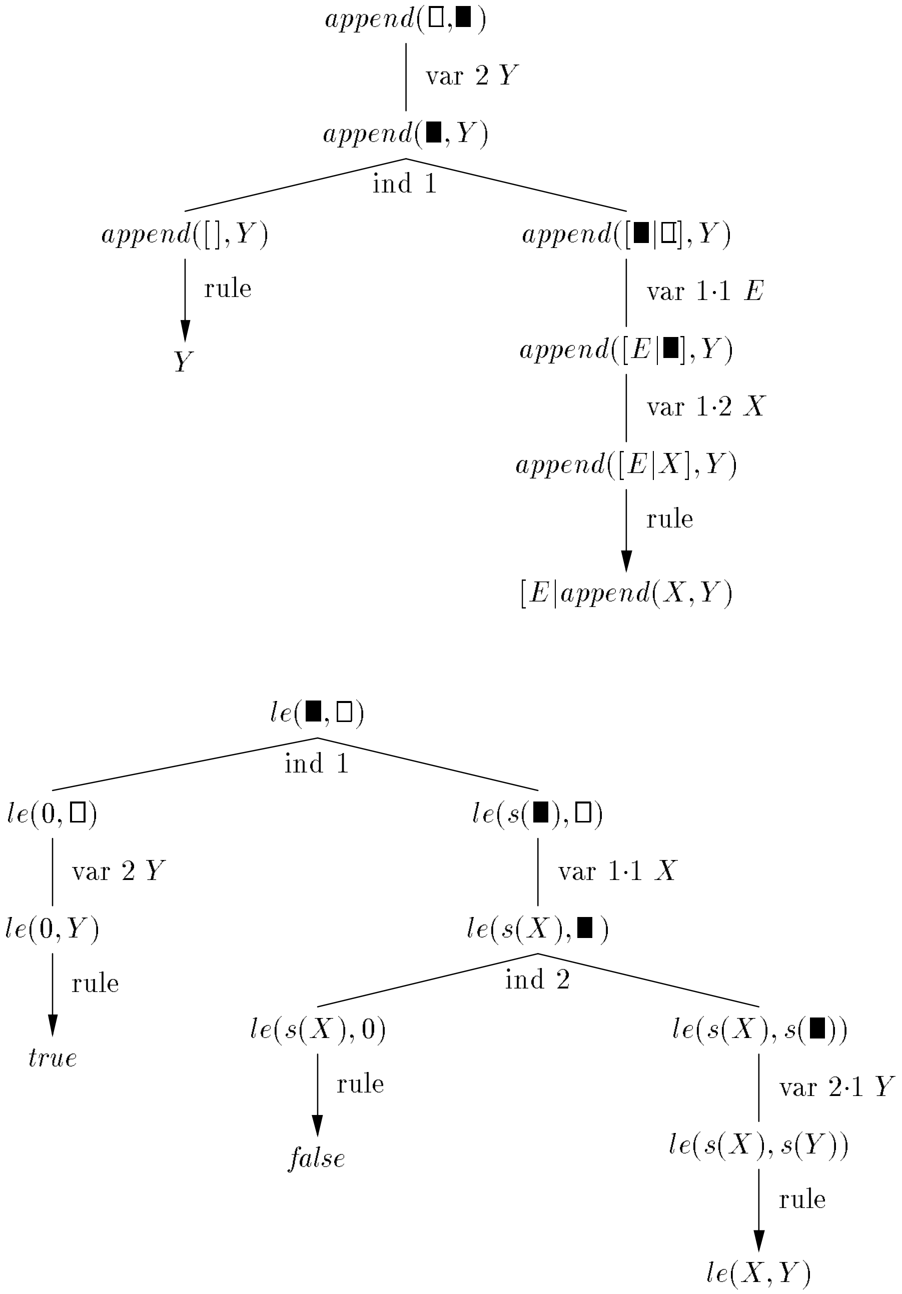

Figure 2: Pictorial representations of definitional trees of the operations *append* and *le* defined in the text. The templates are explicitly represented and the information stored in a node has been spatially rearranged to emphasize the definitional process. The *cursor* highlights the place target of the current binary choice.

Definition 3 We call *inductively sequential* any operation having a definitional tree, and by extension, we call *inductively sequential* any rewrite system such that all its operations are inductively sequential.

3.2 Translation Algorithm

We now describe the generation of a set $\mathcal{H}_{\mathcal{R}}$ of Horn clauses from the set of rewrite rules of an inductively sequential rewrite system $\mathcal{R}$. Let d_f be a definitional tree of some operation f of $\mathcal{R}$. From d_f we generate a set $\mathcal{H}_f$ of Horn clauses. $\mathcal{H}_{\mathcal{R}}$ is the union of $\mathcal{H}_f$ for all f in $\mathcal{R}$. Note that the clauses in $\mathcal{H}_f$, and hence in $\mathcal{H}_{\mathcal{R}}$, contain only one binary predicate denoted by the infix symbol $\Rightarrow$. The rewrite system $\mathcal{R}$ is a global parameter of the procedure, since it remains constant through recursive invocations.

Procedure *Translate*(d)

Input: d: a partial definitional tree of f,
$\mathcal{R}$: a term rewriting system.
Output: $\mathcal{H}$: the set of Horn clauses "contributed" by d.

Execute either of the following cases depending on d:

1. $d = rule(l, r)$: execute either of the following cases depending on r:

    1.1. r is constructor-rooted: output $l \Rightarrow r$.

    1.2. r is operation-rooted: output $l \Rightarrow X \mathrel{:-} r \Rightarrow X$, where X is a fresh variable.

    1.3. r is a variable:

        1.3.1. for each constructor c of the same sort of r output $\{c(X_1, \ldots, X_k)/r\}(l \Rightarrow r)$, where for all i, $1 \leq i \leq k$, X_i is a fresh variable, and

        1.3.2. output $l \Rightarrow X \mathrel{:-} r \Rightarrow X$, where X is a fresh variable.

2. $d = var(t, o, v, d')$: execute *Translate*(d').

3. $d = ind(t, o, d_1, \ldots, d_k)$:

    3.1. for all i, $1 \leq i \leq k$, execute *Translate*(d_i), and

    3.2. instantiate all places of t with fresh variables and output $t[o \leftarrow Z] \Rightarrow X \mathrel{:-} Z \Rightarrow Y,\ t[o \leftarrow Y] \Rightarrow X$, where X, Y, and Z are fresh variables too.

End *Translate*.

For example, the output of the procedure *Translate* when the input is the definitional tree of Figure 2 is:

```
append([],[]) => [].
append([],[A|B]) => [A|B].
append([],A) => B :- A => B.                                    (3)
append([A|B],C) => [A|append(B,C)].
append(A,B) => C :- A => D, append(D,B) => C.
```

A computation involving the above clauses is discussed in Example 1 of the Appendix. The reader may find it difficult to follow the steps of that computation before understanding the analysis, presented in the next section, of the properties of the relation $\Rightarrow$. The details of this presentation are non trivial; thus, we begin with an informal motivation and explanation of certain computational characteristics of $\Rightarrow$.

The opportunities for lazy evaluation are obvious in erasing rules, that is, rules in which some variable occurring in the left hand side does not occur in the right hand side. An example of this condition is provided by the usual "less than or equal to" operation, defined on the natural numbers, whose rules are shown below. The *successor* constructor is denoted by the symbol s. For consistency with the rest of the paper, we use functional notation.

$$\begin{aligned} le(0,Y) &\to \mathit{true} \\ le(s(X),0) &\to \mathit{false} \\ le(s(X),s(Y)) &\to le(X,Y) \end{aligned} \tag{4}$$

Given a ground term t of the form $le(x,y)$, for some terms x and y, we informally discuss when and how much x and/or y should be evaluated to determine the value of t. The value of x must be "considered" first. If x is 0, then one can fire the first rule of display (4) and there is no need to consider y. Furthermore, if x is not 0, but is the *successor* of some term x', then it may be unnecessary to consider x', since if y is 0, then one can fire the second rule to get the value of t. The above observations are incomplete, but suffice to show how one can avoid or limit the evaluation of certain subterms of t.

The word "consider" was informally used as a synonym of "evaluate until a *head normal form* is obtained," that is, a term whose root is a constructor. The relation $\Rightarrow$ is defined for this purpose. This can be seen, for example, in the clauses of display (5), which are generated by the procedure *Translate* on input a definitional tree, shown in Fig. 2, of the above rules. It is relatively easy to verify that resolving a goal of the form `le(`x`,`y`) => T` against these clauses either directly instantiates `T` to a head normal form term, i.e., either `true` or `false`, or generates subgoals that "eventually demand" the evaluation of the appropriate subterm of `le(`x`,`y`)` according to the observations presented earlier.

```
le(0,A) => true.
le(s(A),0) => false.
le(s(A),s(B)) => C :- le(A,B) => C.                    (5)
le(s(A),B) => C :- B => D, le(s(A),D) => C.
le(A,B) => C :- A => D, le(D,B) => C.
```

A computation involving the above clauses is discussed in Example 3 of the Appendix.

4 Analysis of $\Rightarrow$

As already mentioned, $\mathcal{H}_\mathcal{R}$ is not the complete logic program implementing lazy computations in $\mathcal{R}$. Certain properties of the relation $\Rightarrow$, discussed in this section, motivate the introduction of a few additional clauses. For this reason, we analyze $\Rightarrow$ first. This analysis is not straightforward and requires the introduction of two new concepts: *state of a computation* and *ℓ-reduction*.

4.1 State of Computation

A state of a computation is a tree which generalizes *proof trees* as defined, for example, in [49]. We need this generalization, in the proof of Theorem 3, to reason about non-terminating computations. Goal clauses are denoted by prefixing the compound symbol "$:-$" to a sequence of atoms.

Definition 4 Let $:-\mathcal{N}$ be a goal clause, for some atom $\mathcal{N}$, and $\mathcal{L}$ be a logic program. If there exists a clause ξ in $\mathcal{L}$ such that $:-\mathcal{N}$ and ξ resolve with resolvent $:-\mathcal{N}_1,\ldots,\mathcal{N}_k$, for some $k \geq 0$, we call *expansion* of $:-\mathcal{N}$ by ξ the ordered tree $\mathcal{T}$ which has $:-\mathcal{N}$ as root and $:-\mathcal{N}_1,\ldots,:-\mathcal{N}_k$ as leaves. If $k = 0$, then $\mathcal{T}$ has just one leaf, the empty clause.

Definition 5 Let $\mathcal{L}$ be a logic program and $\mathcal{N}$ an atom. An ordered tree $\mathcal{T}$ is *state of a computation* of $:-\mathcal{N}$ in $\mathcal{L}$ if either $\mathcal{T}$ consists of the single node $:-\mathcal{N}$, or $\mathcal{T}$ is obtained by expanding the leftmost leaf different from the empty clause of some state of a computation of $:-\mathcal{N}$ in $\mathcal{L}$.

The nodes of a state of a computation are labeled according to their expansion state as follows: a node is *empty* if it is the empty clause, *unexpanded* if it is a leaf different from the empty clause, and *expanded* if it is not a leaf. Furthermore, a node $\mathcal{N}$ is *solved* if the subtree rooted at $\mathcal{N}$ is finite and all its leaves are empty, otherwise it is *unsolved.* A proof tree is a state of a computation whose root is solved and represents a computation successfully terminated. If $\mathcal{T}$ is a state of a computation, then $|\mathcal{T}|$ denotes the number of non-empty nodes in $\mathcal{T}$.

4.2 ℓ-reduction

A computation of $:- t \Rightarrow u$ in $\mathcal{H}_{\mathcal{R}}$, where t and u are respectively a ground term and a term in $\mathcal{R}$, computes a reduction sequence of t in $\mathcal{R}$. This reduction sequence, crucial to our approach, is called *ℓ-reduction* and is defined in this section.

The reduction sequence we want to compute is one that reduces only needed redexes, because these and only these are all the redexes that really matter in a computation. A ℓ-reduction is one such sequence with the further characterization that the reduced redexes are maximal, too. Although elegant, this definition is operationally unsatisfactory because of the subtlety of the concept of *need.* With some imprecision, a redex r in a term t is *needed* if and only if r is reduced in any reduction sequence of t to normal form. Computing every reduction sequence of t to normal form may be impossible, since it is undecidable whether a term has a normal form [25], but more important, it would completely defy the reasons why we want to determine whether a redex is needed in the first place, i.e., to avoid performing reduction steps that do not contribute to the computation.

Thus, our definition of a ℓ-reduction is operational through a procedure, defined below, executing a traversal of a state of a computation. Theorem 2 proves the equivalence of our operational definition with the more elegant, but less practical, definition proposed earlier. This *modus operandi* has two advantages. It allows us to actually compute a reduction sequence in which only needed redexes are reduced without the cost of computing all the reduction sequences of a term to normal form, and it also allows us to state and prove certain invariant properties of ℓ-reductions on which later results are based. These invariant properties take the form of two assertions, referred to as *input* and *output assertion* respectively, which are stated at the beginning and at the end of the high-level pseudo-code of the procedure defining a ℓ-reduction. Theorem 1 proves that these assertions hold under appropriate conditions.

The *classification* of a clause is a labeling of the clauses of $\mathcal{H}_{\mathcal{R}}$ resulting from the procedure

Translate. We call *n-clause* any clause generated by the execution of instruction n of the procedure *Translate*, and *n-goal* (or *n-node*) any goal of a state of a computation expanded by an n-clause.

Procedure *Reduce*$(\mathcal{T}, \mu, o, \ell)$

Input: $\mathcal{T}$: a state of a computation,
μ: a reduction sequence in $\mathcal{R}$,
o: an occurrence.
Output: ℓ: a reduction sequence in $\mathcal{R}$.

Let $\mathcal{N}$ denote the root of $\mathcal{T}$ and let t denote the term reduced by μ.

If $\mathcal{N}$ is empty, then abort; else $\mathcal{N}$ must be equal to $:\!-\, t_l \Rightarrow t_r$, for some terms t_l and t_r.

If $\mathcal{N}$ is unexpanded, then abort; else $\mathcal{N}$ has been expanded by some clause ξ of $\mathcal{H}_{\mathcal{R}}$ with match σ.

Input assertion: {There exists a ground term t' such that $\mu = t \overset{*}{\rightarrow} t'$ and $t'/o = t_l$}.

Execute either of the following cases depending on the classification of $\mathcal{N}$.

1. $\mathcal{N}$ is either a 1.1-node or a 1.3.1-node: Then ξ must be equal to $l \Rightarrow r$, for some terms l and r.
   Set ℓ to extend μ with the reduction step $t'[o \leftarrow \sigma(r)]$.

2. $\mathcal{N}$ is either a 1.2-node or a 1.3.2-node: Then ξ must be equal to $l \Rightarrow X :\!-\, r \Rightarrow X$, for some terms l and r, and $\mathcal{N}$ must have only one child, say $\mathcal{T}'$.
   Set μ' to extend μ with the reduction step $t'[o \leftarrow \sigma(r)]$ and execute *Reduce*$(\mathcal{T}', \mu', o, \ell)$.

3. $\mathcal{N}$ is a 3.2-node: Then ξ must be equal to $t_Z \Rightarrow X :\!-\, Z \Rightarrow Y,\ t_Y \Rightarrow X$, for some terms t_Z and t_Y, and let p denote the occurrence of Z in t_Z. Also, $\mathcal{N}$ must have two children, say $\mathcal{T}'$ and $\mathcal{T}''$.
   Execute *Reduce*$(\mathcal{T}', \mu, o{\cdot}p, \ell')$ and execute *Reduce*$(\mathcal{T}'', \ell', o, \ell)$.

Output assertion: {There exists a ground term t' such that $\ell = t \overset{*}{\rightarrow} t'$ and $t'/o = \rho(t_r)$, for some substitution ρ}.

Return.

End *Reduce*.

The abort statements at the beginning of the procedure ensure that $\mathcal{T}$ is a state of a non-failed computation, i.e., $\mathcal{T}$ is neither the empty tree nor a tree containing unsolved nodes.

Theorem 1 Let $\mathcal{T}$ be a state of a computation of $:\!-\, t \Rightarrow u$ in $\mathcal{H}_{\mathcal{R}}$, where t and u are respectively a ground term and a term in $\mathcal{R}$. The assertions in the code of procedure *Reduce* hold throughout the execution of $\mathit{Reduce}(\mathcal{T}, t \overset{*}{\rightarrow} t, \Lambda, \ell)$.

Proof The proof is by induction on the number of times that an assertion cutpoint [19] is crossed during the execution of *Reduce*. We structure the proof in three conditions. Condition 1: The input assertion holds for the initial activation of *Reduce*. Condition 2: If the input assertion holds for an activation a, then it holds for a recursive activation b directly called by a, provided that the output assertion holds upon return from any recursive activation c directly called by a before b. Condition 3: If the input assertion holds for an activation a and the output assertion holds upon return from any recursive activations directly executed by a, then the output assertion holds for activation a. In the discussion below, when we refer to some symbol within the scope of the procedure *Reduce* we distinguish its different instances in different activations of the procedure by subscripting the symbol with activation identifiers. For example, t'_a and t'_b may denote the value of the reduct of t according to μ in activations a and b respectively. Proof of condition 1: The input assertion is a direct consequence of the values of the arguments of *Reduce* in the initial activation. Proof of condition 2: The condition is vacuously true for case 1 of the procedure. For case 2, let a be an activation and b the recursive activation directly called by a. Since $o_a = o_b$, we denote both occurrences simply with o. By the procedure *Translate*, there exists a rule $l' \rightarrow r'$ in $\mathcal{R}$ such that $l_a \rightarrow r_a = \rho(l' \rightarrow r')$ for some substitution ρ. By assumption, $t'_a/o = (t_l)_a$. By definition of state a of computation, $(t_l)_a = \sigma_a(l_a)$. Thus, $t'_a \rightarrow t'_a[o \leftarrow \sigma_a(r_a)]$ is a reduction step. By construction $\mu_b = \mu'_a$. Thus, μ_b is a reduction sequence in $\mathcal{R}$. The root of $\mathcal{T}_b$ is $:\!-\, \sigma_a(r_a) \Rightarrow X$, hence $(t_l)_b = \sigma_a(r_a)$. By construction, $t'_b = t'_a[o \leftarrow \sigma_a(r_a)]$. Thus, $t'_b/o = (t_l)_b$, t'_b is ground, and the input assertion holds for activation b. For case 3, let a be an activation, b the first recursive activation directly called by a, and c the second one if b terminates. The input assertion for activation b is a direct consequence of the fact that $t'_b = t'_a$, $(t_l)_b = (t_l)_a/p$, and $o_b = o_a{\cdot}p$. We now prove the input assertion for activation c assuming that activation b terminates and establishes the output assertion, i.e., $\ell_b = t \overset{*}{\rightarrow} t'_b$ is a reduction sequence in $\mathcal{R}$, t'_b is ground, and $t'_b/o_b = \rho_b(Y_b)$. By construction, $\mu_c = \ell_b$. Thus, μ_c is a reduction sequence in $\mathcal{R}$ and t'_c is ground. Since $o_a = o_c$, we denote both occurrences simply with o. We must prove $t'_c/o = (t_l)_c$. Upon entering activation

c, activation a has established both $t'_c = t'_a[o{\cdot}p_a \leftarrow \rho_b(Y)]$ and $(t_l)_c = (t_l)_a[p_a \leftarrow \rho_b(Y)]$. The input assertion on a is $t'_a/o = (t_l)_a$. Finally, since $t/o[p \leftarrow u] = t[o{\cdot}p \leftarrow u]/o$, for all t, o, p, and u, the condition is proved. Proof of condition 3: For case 1, the proof is analogous to that of condition 2 for case 2. For both case 2 and 3, let a be an activation and b the last recursive activation directly called by a. Since $\ell_a = \ell_b$ and $(t_r)_a = (t_r)_b$, the output assertion for activation a is a direct consequence of the assumption that the output assertion holds for activation b. □

The proof of Theorem 1 does not depend on the termination of *Reduce*. Each assertion in the procedure holds when its cutpoint is crossed regardless of whether *Reduce* will eventually abort, terminate, or run forever. *Reduce* normally terminates on a state of a computation $\mathcal{T}$ if and only if $\mathcal{T}$ is a proof tree, i.e., its root goal is solved.

Definition 6 Let $\mathcal{T}$ be a proof tree of $:\!-\ t \Rightarrow u$ in $\mathcal{H}_{\mathcal{R}}$, where t and u are respectively a ground term and a term in $\mathcal{R}$. The *ℓ-reduction* of t computed by $\mathcal{T}$ is the value of ℓ computed by *Reduce*$(\mathcal{T}, t \xrightarrow{*} t, \Lambda, \ell)$.

4.3 Properties of $\Rightarrow$

In this section we state and prove some fundamental properties of the relation $\Rightarrow$. In short, we show that, for ground terms, $\Rightarrow$ is contained in $\xrightarrow{*}$. This property is referred to as *soundness* in [2, 39, 41, 48]. We show that if $t \Rightarrow t'$, then t is not a head normal form [43], whereas t' is. This property is referred to as *simplification* in [39, 40, 41]. We also show some key properties of ℓ-reductions, namely that a ℓ-reduction is in some weak sense unique, that it reduces only needed redexes in the sense of [25], and that its length bounds the size of a proof tree.

We introduce the following notation. If α denotes the reduction sequence $t_0 \to t_1 \to \ldots \to t_k$, for some terms $t_0, t_1, \ldots, t_k$ for some $k \geq 0$, and t and o are respectively a term and an occurrence such that $t/o = t_0$, then $t[o \leftarrow \alpha]$ denotes the reduction sequence $t[o \leftarrow t_0] \to t[o \leftarrow t_1] \to \ldots \to t[o \leftarrow t_k]$. If β denotes a reduction sequence $t \xrightarrow{*} t'$, for some terms t and t', then $\beta[o \leftarrow \alpha]$ denotes the reduction sequence obtained by appending the reduction sequence $t'[o \leftarrow \alpha]$ to β provided that o is an occurrence of t'.

Many of our proofs are by *Structural Induction* on proof trees. A definition of this proof technique and a proof of its correctness can be found, for example, in [14].

Lemma 1 If $t \Rightarrow t'$, for some ground terms t and t', then let $\mathcal{T}$ denote a proof tree of $:- t \Rightarrow t'$ and ℓ the ℓ-reduction of t computed by $\mathcal{T}$. The following conditions hold:

1. t is operation-rooted;
2. ℓ reduces t to t';
3. t' is the only constructor-rooted term in ℓ;
4. $|\mathcal{T}| < 2|\ell|$.

Proof

1. If $t \Rightarrow t'$, then there exists a clause in $\mathcal{H}_{\mathcal{R}}$ with head $l \Rightarrow l'$ and a match σ such that $\sigma(l \Rightarrow l') = t \Rightarrow t'$. The constructor discipline and the translation performed by *Translate* imply that l is operation-rooted and thus it must be t.

2. The output assertion of the procedure *Reduce* implies that ℓ reduces t to some term t'' and that $t'' = \rho(t')$, for some substitution ρ. Since t' is ground, $t'' = t'$.

3. By structural induction on $\mathcal{T}$. Base case: $\mathcal{T}$ has only one non-empty node which is either a 1.1-node or a 1.3.1-node. The goal $:- t \Rightarrow t'$ resolves with some 1.1-clause or 1.3.1-clause ξ, such that $\xi = l \Rightarrow r$, for some terms l and r, and r is constructor rooted. ℓ is a single reduction step and, by condition 2, $\ell = t \to t'$. By condition 1, t is not constructor-rooted. Since $t' = \rho(r)$, for some substitution ρ, t' is constructor-rooted, and the condition is verified. Inductive case: The root of $\mathcal{T}$ is either a 1.2-node, a 1.3.2-node, or a 3.2-node. We consider two cases. Case 1: the root of $\mathcal{T}$ is either a 1.2-node or a 1.3.2-node. Then, $\ell = t \to t'' \xrightarrow{*} t'$ and $t'' \xrightarrow{*} t'$ is a ℓ-reduction. The claim is then a direct consequence of condition 1 and the inductive hypothesis. Case 2: the root of $\mathcal{T}$ is a 3.2-node. Let ℓ' and ℓ'' be the ℓ-reductions computed respectively by the left and right subtrees of $\mathcal{T}$. By construction, $\ell = t[o \leftarrow \ell'][\Lambda \leftarrow \ell'']$, for some non-empty occurrence o. Thus, there are no constructor-rooted terms in $t[o \leftarrow \ell']$, and the inductive hypothesis on ℓ'' proves the claim.

4. *Reduce* defines a bijection between the steps of ℓ and the nodes of $\mathcal{T}$ which are neither empty nor 3.2-nodes. The problem is thus reduced to counting these nodes and is left to the reader.

Note that the nodes contributing to a reduction step of ℓ have one child, while 3.2-nodes have two children. $\square$

The completeness and parsimony properties implied by the inductive sequentiality of $\mathcal{R}$ imply that for any ground operation-rooted *constructor* term t there exists *exactly* one rule $l \rightarrow r$ of $\mathcal{R}$ such that $t = \sigma(l)$, for some match σ. This property is extended by the procedure *Translate* to any ground operation-rooted (*arbitrary*) term in the sense of Lemma 2.

Lemma 2 If t is a ground operation-rooted term and T a variable, then:

1. (existence) there exists a clause in $\mathcal{H}_{\mathcal{R}}$ whose head matches $t \Rightarrow T$;
2. (uniqueness) any clause in $\mathcal{H}_{\mathcal{R}}$ witnessing condition 1, except the first in the order of generation of the procedure *Translate*, finitely fails the call $t \Rightarrow T$.

Proof Let f be the root of t and d a definitional tree of f. The template in the root of d is $f(\square, \ldots, \square)$, thus the head of the last clause output by the procedure *Translate* with input d is $f(X_1, \ldots, X_k) \Rightarrow t'$, for some term t' and variables $X_1, \ldots, X_k$, for some $k \geq 0$. This proves condition 1. Let ξ be the first clause in $\mathcal{H}_{\mathcal{R}}$ witnessing condition 1. If ξ' is a clause distinct from ξ, and ξ' witnesses condition 1, then ξ' is either a 1.3.2-clause or a 3.2-clause. In both cases there exists an occurrence o in t such that t/o is constructor-rooted and, for some variable T', $t/o \Rightarrow T'$ is the first call in the body of ξ'. By condition 1 of Lemma 1 this call cannot succeed. The number of such recursive calls that may occur in a proof tree of a computation of $t \Rightarrow T$ is bounded by the depth of t, thus the computation terminates. This proves condition 2. $\square$

Point 1 of Lemma 2 implies that a goal g of the form $:- t \Rightarrow T$, where t is operation-rooted and T is a variable, never fails in $\mathcal{H}_{\mathcal{R}}$. This simple result is proved by a straightforward induction relying on the observation that any subgoal generated by g has the same form of g. Point 2 implies that the proof tree of g, and consequently the ℓ-reduction computed by g, are unique in a sense made precise shortly. In particular, the order of the clauses of $\mathcal{H}_{\mathcal{R}}$ does not affect the result of a computation, although it may affect the efficiency with which such a result is computed. Point 2 also shows that the first clause, in the order of generation of the procedure *Translate*, matching g

is "the right one" and can be irrevocably committed as soon as it is found. Later, we will discuss two situations in which a Prolog interpreter can take advantage of this knowledge.

We now make more precise the uniqueness of ℓ-reductions implied by Lemma 2. A term may have several distinct maximal needed redexes. Since these redexes are disjoint, all the reduction sequences that reduce only these redexes can be seen as "permutations" of each other [25]. In particular, these reduction sequences have the same length, a point we will discuss again later. The non-determinism in the choice of redexes seems to be missing from our definition. The reason is that an operation may have several non-isomorphic definitional trees [3] and the non-determinism discussed earlier is removed by the choice of a particular tree. Indeed, from $:-\, t \Rightarrow T$ exactly one ℓ-reduction is computed.

Inductively sequential rewrite systems are *orthogonal* and in such systems certain redexes of a term t can be shown to be *needed* to reduce t to normal form [25]. We have already presented this concept, although in a form that sacrificed some precision to ease understanding. Now we are compelled to give a more precise definition of *need*. This definition is required only for the proof of the following theorem and could be avoided by the reader not interested in its details.

A subtlety of the definition of needed redex is that the reduction of any such redex cannot be avoided to compute a normal form, but can be delayed by first reducing other redexes. Thus, what happens to a needed redex s of a term t when some other redex of t is reduced? In the reduct of t one can identify a set of "descendants" of s, and some member of this set conserves the property of being needed. A formal definition, quite technical, of descendant can be found in [25]. A less formal, but more intuitive, formulation is proposed in [30]. Suppose we have a reduction sequence $t \xrightarrow{*} t'$ and a subterm s of t. The descendants of s in t' are computed as follows: Underline the root of s and perform the reduction sequence $t \xrightarrow{*} t'$. Then, every subterm of t' with an underlined root is a *descendant* of s[2]. Now, a redex s in a term t is *needed* if in every reduction of t to normal form a descendant of s is reduced.

An example may clarify these concepts. Consider the operation *le* of display (4) and the problem of normalizing a term t equal to $le(add(0,0), add(0,0))$, where *add* is the usual addition of natural numbers. We previously argued that the first argument of *le*, say x, is needed. In this trivial case

---

[2] Boudol [7] similarly suggests to mark occurrences with colors and credits J. Vuillemin for this idea.

one can easily verify this claim by computing every reduction sequence normalizing t. Thus, let us reduce instead the second argument of le after underlining in t the root of x. The reduct is $le(\underline{add}(0,0),0)$, where the subterm with the underlined root is the descendant of x. It is easy to verify that one must still reduce it in order to reduce t to a normal form.

The situation is slightly more complicated if one reduces either a subterm or a superterm of a needed redex. Although not a problem, this situation never happens in our approach, since the needed redexes that we find, and hence reduce, do not have proper super-redexes. The following concepts will be used to characterize this condition. Let t be a term and s a redex of t. We say that s is *maximal* in t if, for any redex r of t, s is not a proper subterm of r. Also, we say that s is *inner* in t if $s \neq t$. We will drop the qualifier "in t" when the context of s is clear. Obviously, a non inner redex has no proper superterm and consequently is maximal.

Theorem 2 If $t \Rightarrow t'$, for some ground terms t and t', then any redex reduced in a ℓ-reduction of t is maximal and needed.

Proof We prove the theorem by strong arithmetical induction on the number of inner redexes reduced in a ℓ-reduction. Let $\mathcal{T}$ denote a proof tree of $:\!-\, t \Rightarrow t'$ and ℓ denote the ℓ-reduction of t computed by $\mathcal{T}$. Note that there are inner redexes reduced in ℓ if and only if there are 3.2-nodes in $\mathcal{T}$. Base case: No inner redex is reduced in ℓ, thus every redex is maximal. We now prove the need. Let s be a redex reduced in ℓ. Condition 1 of Lemma 1 implies that s is an operation-rooted term. Since s is non inner, it has no proper superterm. A reduction of a subterm of s cannot change the root of (the resulting descendant of) s. Thus, every reduction sequence beginning with s ends in an operation-rooted term unless a descendant of s is reduced. The completeness property of $\mathcal{R}$ implies that any ground operation-rooted term is not a normal form. Consequently, if some descendant of s is not reduced, the normal form of s cannot be computed, and thus s is needed. Inductive case: We show that ℓ can be built by "pasting" together two other ℓ-reductions. For these reduction sequences the inductive hypothesis hold and the pasting operation preserves maximality and need of redexes. Without loss of generality we assume that the root of $\mathcal{T}$ is a 3.2-goal. Let ξ be the clause of $\mathcal{H}_{\mathcal{R}}$ resolving with $:\!-\, t \Rightarrow t'$, let σ be the match of the resolution, let d_ξ be the *pdt* from which ξ is generated, and let o be the occurrence in d_ξ. As a preliminary step we show that t is not a redex and that t/o is operation-rooted. The goal in the left child of $\mathcal{T}$ is $:\!-\, t/o \Rightarrow Y$, for some variable Y. Since such a goal succeeds, by condition 1 of Lemma 1, t/o is operation-rooted. The parsimony property implied by the inductive sequentiality of $\mathcal{R}$ implies that t can be reduced only by a rule defined in a leaf of d_ξ. The left side of any such rule has a constructor-rooted subterm at occurrence o. Since t/o is operation-rooted, t cannot be a redex. Now we describe the structure of ℓ. Let ℓ' and ℓ'' be the ℓ-reductions computed respectively by the left and the right subtree of $\mathcal{T}$. By construction, $\ell = t[o \leftarrow \ell'][\Lambda \leftarrow \ell'']$, thus $|\ell| = |\ell'| + |\ell''|$. By Lemma 1, $|\ell'| > 0$ and $|\ell''| > 0$. Furthermore, a redex inner in either ℓ' or ℓ'' is inner in ℓ, thus the inductive hypothesis holds for both ℓ' and ℓ''. The redexes reduced in ℓ'' are maximal and needed by the inductive hypothesis. To complete the proof, we show that the remaining redexes reduced in ℓ are maximal and that the whole reduction sequence $t[o \leftarrow \ell']$ is needed to reduce t to normal form. Let $l \Rightarrow r$, for some terms l and r, be the head of ξ. The inductive sequentiality of $\mathcal{R}$ and the procedure *Translate* imply that l is an operation-rooted constructor term. This implies that for any non-trivial prefix p of o, the root of l/p is a constructor and consequently t/p is not a redex. Since we have already proved that t is not a redex, any redex maximal in t/o is maximal in t. The inductive hypothesis implies that all redexes reduced in ℓ' are maximal and consequently all redexes reduced in $t[o \leftarrow \ell']$ are maximal too. A descendant of t must be reduced to compute a normal form of t, but no descendant s of t can be a redex until s/o is constructor-rooted. Condition 3 of Lemma 1 implies that the whole reduction sequence $t[o \leftarrow \ell']$ is needed to reduce t/o to a constructor-rooted term and, by the inductive hypothesis, all the redexes reduced in this sequence are needed. □

We are interested in reducing only needed redexes because the reduction of any other redex is wasted, as far as computing normal forms is concerned. On the other hand, the definition of needed redex involves reducing all the redexes of a term. Obviously, we do not want to reduce all these redexes only to discover which ones should not have been reduced. Theorem 2 is significant because it shows that $\Rightarrow$ finds needed redexes without any look ahead, that is, without unnecessary reductions. Point 4 of Lemma 1 strengthens this result by showing that the amount of work, measured as the size of a proof tree, involved in finding these redexes is quite modest—which in general is not the case [28, 29]. Lemma 2 further strengthens the above results by showing that the required proof tree can be computed without any failures if the clauses are selected in the order in which they are generated by the procedure *Translate*. That is, the size of the proof tree is a faithful measure of the effort that goes into the computation of the corresponding ℓ-reduction.

In the next section we show how to use ℓ-reductions to compute normal forms. Although we reduce only needed redexes, the ℓ-reduction of a term to its normal form may not be the shortest. The problem stems from delaying the reduction of a needed redex until it has generated multiple descendants, and it occurs because among the needed redexes of a term we reduce shallow ones before deep ones. The rewrite system of display (11) shows this problem in the larger context of non-determinism.

A shortest-length reduction sequence may be obtained from a ℓ-reduction by "sharing" the common subterms of a term. An early proposal in this direction is found in [53]. A comprehensive discussion within the general framework of *labelled* term rewriting system and several references on the subject are found in [34]. Implementation techniques suitable for logic programming are found in [38, 39, 40, 41] and can be applied to our approach as well. Basically, they avoid the repeated reduction of two equal redexes by binding them together by means of a logic variable. Although in some cases this feature improves dramatically the performance, its convenience for average programs is not obvious because of its overhead. Since logic programs augmented with this feature become harder to read and reason about, we do not pursue this option.

5 Execution

Huet and Lévy [25] prove that if a term t has a normal form, then repeated rewriting of needed redexes leads to that normal form. Thus, Theorem 2 suggests that $\Rightarrow$ might be used to attempt computing normal forms. In this section we introduce a new binary relation on terms, denoted by the infix symbol $\twoheadrightarrow$, for this purpose. The relation $\twoheadrightarrow$ can be thought of as a driver which guides the application of $\Rightarrow$ to appropriate subterms of a term t until the normal form of t is computed.

Definition 7 For all ground terms t and t', $t \twoheadrightarrow t'$ if and only if one of the following conditions holds:

1. $t' = c(t'_1, \ldots, t'_k)$, if $t = c(t_1, \ldots, t_k)$ for some constructor c and terms $t_1, \ldots, t_k$, for some $k \geq 0$, and for all i, $1 \leq i \leq k$, $t_i \twoheadrightarrow t'_i$.
2. $t \Rightarrow t''$ and $t'' \twoheadrightarrow t'$, for some term t''.

Condition 1 of Lemma 1 implies that the term t in case 2 of the above definition is operation-rooted, thus the cases of the definition are mutually exclusive. Intuitively, if $t \twoheadrightarrow t'$, then t' is computed from t by "skipping over" constructors and applying $\Rightarrow$ to operation-rooted terms.

The relation $\twoheadrightarrow$ is easily implemented as a logic program. We need a clause for each constructor in order to satisfy condition 1 of the definition and one clause to satisfy condition 2. All these clauses are independent of any operation, rather they depend only on the constructors of a sort. For example, the following clauses must be added to those presented in display (3).

```
[] ->> [].
[A|B] ->> [C|D] :- A ->> C, B ->> D.                    (6)
A ->> B :- A => C, C ->> B.
```

Given a rewrite system $\mathcal{R}$, from now on $\mathcal{H}_{\mathcal{R}}$ is intended to define both relations $\Rightarrow$ and $\twoheadrightarrow$. Many of the concepts and results discussed in the previous sections are extended with little modification. In particular, we extend the definition of ℓ-reduction. We show that an extended ℓ-reduction still reduces only needed redexes and that its length is a key parameter in bounding the size of the proof tree computing it. We also show that the relation $\twoheadrightarrow$ is a sound and complete (in a sense made precise later) implementation of $\xrightarrow{*}$.

We call $\Rightarrow$*-goal* a goal whose predicate is $\Rightarrow$ and $\twoheadrightarrow$*-goal* a goal whose predicate is $\twoheadrightarrow$. Now we focus on goals of the form $:\!-\ t \twoheadrightarrow u$, where t is a ground term and u is a term. If $\mathcal{T}$ is a state of a computation of such a goal, then we extend the definition of ℓ-reduction of t computed by $\mathcal{T}$ by extending the procedure *Reduce* to handle $\twoheadrightarrow$-nodes.

We classify $\twoheadrightarrow$-nodes according to the cases of Definition 7. Let $\mathcal{N}$ denote a $\twoheadrightarrow$-node. $\mathcal{N} = \ :\!-\ t \twoheadrightarrow t'$ for some terms t and t'. If t is constructor-rooted, we say that $\mathcal{N}$ is a *1-node*, otherwise t is operation-rooted and we call it a *2-node*. The case statement of the procedure *Reduce* is extended with the two following new entries.

Extension to the procedure *Reduce*

4. $\mathcal{N}$ is a 1-node: Then $\mathcal{N}$ must have has k children, say $\mathcal{T}_1, \ldots, \mathcal{T}_k$, for some $k \geq 0$.

   Let $\mu_0 = \mu$; for i from 1 to k execute *Reduce*$(\mathcal{T}_i, \mu_{i-1}, o{\cdot}i, \mu_i)$; and let $\ell = \mu_k$.

5. $\mathcal{N}$ is a 2-node: Then $\mathcal{N}$ must have two children, say $\mathcal{T}'$ and $\mathcal{T}''$.

   Execute $\mathit{Reduce}(\mathcal{T}', \mu, o, \mu')$ and execute $\mathit{Reduce}(\mathcal{T}'', \mu', o, \ell)$.

Similar to Theorem 1, the assertions in the code of the extended procedure *Reduce* hold throughout the execution of $\mathit{Reduce}(\mathcal{T}, t \overset{*}{\rightarrow} t, \Lambda, \ell)$, where $\mathcal{T}$ may now contain also $\twoheadrightarrow$-nodes. Accordingly, the definition of ℓ-reduction of a term is extended to the states of a computation of a $\twoheadrightarrow$-goal. The following lemma extends to the relation $\twoheadrightarrow$ the basic properties of the relation $\Rightarrow$ proved in Lemma 1 and Theorem 2.

Lemma 3 If $t \twoheadrightarrow t'$, for some ground terms t and t', then let $\mathcal{T}$ denote a proof tree of $:\!-\, t \twoheadrightarrow t'$ and ℓ the ℓ-reduction of t computed by $\mathcal{T}$. The following conditions hold:

1. ℓ reduces t to t';
2. t' is a value;
3. $|\mathcal{T}| \leq 2|\ell| + |t'|$;
4. Any redex reduced in ℓ is maximal and needed.

Proof

1. Analogous to condition 2 of Lemma 1.
2. By structural induction on $\mathcal{T}$.
3. By structural induction on $\mathcal{T}$. Base case: t is a constant constructor and $t = t'$. ℓ is the null reduction sequence of t and the size of t' is 1. Thus $|\mathcal{T}| = 2|\ell| + |t'|$ and the condition holds. Inductive case: Letting $\mathcal{N}$ denote the root of $\mathcal{T}$, we consider two cases. Case $\mathcal{N}$ is a 1-node: There exists a constructor c, a non-negative integer k, and terms $t_1, \ldots, t_k$ and $t'_1, \ldots, t'_k$ such that $t = c(t_1, \ldots, t_k)$ and $t' = c(t'_1, \ldots, t'_k)$. For all i, $1 \leq i \leq k$, let $\mathcal{T}_i$ denote the proof tree of $:\!-\, t_i \twoheadrightarrow t'_i$ and ℓ_i the ℓ-reduction computed by $\mathcal{T}_i$. By construction, $\ell = t[1 \leftarrow \ell_1] \ldots [k \leftarrow \ell_k]$. Hence, $|\mathcal{T}| = 1 + \sum_i |\mathcal{T}_i| \leq 1 + 2\sum_i |\ell_i| + \sum_i |t'_i| = 2|\ell| + |c(t'_1, \ldots, t'_k)| = 2|\ell| + |t'|$. Case $\mathcal{N}$ is a 2-node: Let $\mathcal{T}'$ and $\mathcal{T}''$ denote the left and right subtrees of $\mathcal{T}$ and ℓ' and ℓ'' the ℓ-reductions they respectively compute. By construction, $\ell = t[o \leftarrow \ell'][\Lambda \leftarrow \ell'']$, for some non-empty occurrence o.

By the inductive hypothesis, $|\mathcal{T}''| \leq 2|\ell''| + |t'|$. By condition 4 of Lemma 1, $|\mathcal{T}'| < 2|\ell'|$. Thus, $|\mathcal{T}| = 1 + |\mathcal{T}'| + |\mathcal{T}''| < 1 + 2|\ell''| + |t'| + 2|\ell'| = 1 + 2|\ell| + |t'|$, i.e. $|\mathcal{T}| \leq 2|\ell| + |t'|$.

4. By structural induction on $\mathcal{T}$. Base case: t is a constant constructor, ℓ is the null reduction sequence of t, and the claim is vacuously true. Inductive case: We consider the same two cases and we adopt the same notation used in the proof of the previous condition. Case $\mathcal{N}$ is a 1-node: Constructor-rooted terms are not redexes, hence the claim is a direct consequence of the inductive hypothesis. Case $\mathcal{N}$ is a 2-node: By Theorem 2, any redex in ℓ' is maximal and needed. By condition 3 of Lemma 1, ℓ' reduces t to some constructor-rooted term. By the previous case, any redex in ℓ'' is maximal and needed. Thus the claim holds. □

Point 1 of Lemma 3, which expresses our notion of soundness, i.e., the fact that the relation $\twoheadrightarrow$ is contained in the relation $\overset{*}{\rightarrow}$, deserves some comments. The implementation of a term rewriting system is a significant component of several research proposals referenced in this note. However, many of these proposals are not specifically concerned with rewriting *per se*. Rewrite rules come in as *oriented* equations, which are the focus of the discussion. The orientation is an expedient motivated by the fact that rewrite rules are operational whereas equations are declarative. Our approach is somewhat reversed. The primary focus of our work is rewriting. Equality comes in because a rewrite relation $\rightarrow$ naturally defines an associated relation called *convertibility* and denoted by $\overset{*}{\leftrightarrow}$. The convertibility relation is regarded as equality on terms when rewrite rules are regarded as equations [12, 29]. Our formulation of soundness, similar to [2, 40, 48], is inspired by rewriting and has the advantage of being compatible with regarding rewrite rules as equations and of being general enough to hold also for situations in which rewrite rules cannot be regarded as equations.

When the focus is on equations, there are goals in the form of equations. In [15] it is discussed how this form could be adopted for other goals as well. For example, a goal $p(t_1, \ldots, t_k)$, where p is a relation, can be regarded as a shorthand for $p(t_1, \ldots, t_k) = \mathit{true}$, where *true* is a suitably defined symbol. Logic programming then suggests [33, 54] to express soundness as the fact that the refutability of an equation $t = u$ implies that the equation is true, i.e., the equality of t and u is logically implied by the rewrite rules (regarded as equations) and a suitable equality theory.

There are, however, rewrite systems where regarding equations as rewrite rules is inappropriate, for example, the system presented in display (11). This is due to the lack of confluence. Regarding the rewrite rules of the operation *flip* as oriented equations would lead to $0 = 1$. In this note, we discuss inductively sequential systems, which are confluent. Our reason to reject non confluent systems, though, is not to avoid the above situation, which would not be a problem for us, but because there is no universally accepted notion of laziness for non confluent systems due to the lack of a suitable generalization of the notion of needed redex. Many results of this note, including our transformation, are extended with minor adjustments to a class of systems [2] that includes both the inductively sequential ones and some non confluent ones. In particular, our formulation of soundness holds unchanged for this larger class, whereas the more traditional formulation would clearly be unacceptable.

Our formulation of soundness is compatible with the traditional one, when rewrite rules are regarded as oriented equations. We sketch how this compatibility is a consequence of our results. We proved that the relation $\twoheadrightarrow$ is contained in $\overset{*}{\to}$. The relation $\overset{*}{\to}$ is contained in the convertibility relation $\overset{*}{\leftrightarrow}$. By transitivity, this implies that if two terms are in the relation $\twoheadrightarrow$, then the equality of these terms is implied by the rewrite rules if these are regarded as oriented equations.

The more conventional formulation of soundness expressed in terms of an equality relation is adopted by several authors, including, for example, [15, 37, 55]. However, either strong computational properties are required in addition to confluence for the simple formulation we discussed above, or a more complicated formulation is required to accommodate certain practical conditions, such as incompletely defined operations. A marginal advantage of our formulation is its simplicity, i.e., $\twoheadrightarrow \subset \overset{*}{\to}$, without placing heavy demands on the properties of $\to$.

The existence and uniqueness properties proved for $\Rightarrow$ in Lemma 2 are extended to the relation $\twoheadrightarrow$ in the next lemma.

Lemma 4 If t is a ground operation-rooted term and T a variable, then:

1. there exists a clause in $\mathcal{H}_{\mathcal{R}}$ whose head matches $t \twoheadrightarrow T$;

2. any clause in $\mathcal{H}_{\mathcal{R}}$ witnessing condition 1, except the first in the order of generation of the procedure *Translate*, finitely fails the call $t \twoheadrightarrow T$.

Proof

1. The head of the last clause defining $\twoheadrightarrow$ is $A \twoheadrightarrow B$, for A and B variables.

2. Let ξ be the first clause in $\mathcal{H}_\mathcal{R}$ witnessing condition 1. If ξ' is a clause distinct from ξ, and ξ' witnessing condition 1, then t is constructor-rooted and ξ' is $A \twoheadrightarrow B \,{:}{-}\, A \Rightarrow C,\ C \twoheadrightarrow B$, for some variables A, B, and C. The left subgoal of ξ' fails immediately, since if t is constructor-rooted, there is no clause in $\mathcal{H}_\mathcal{R}$ matching $t \Rightarrow T$. $\square$

The observations following Lemma 2 concerning the relation $\Rightarrow$ are extended by the above lemma to the relation $\twoheadrightarrow$.

We now prove our main result, that is, if a term has an "acceptable" normal form, then $\twoheadrightarrow$ computes it. In our rewrite systems, a ground term is a normal form if and only if it is a value, i.e., a constructor term. Thus we are going to prove that the relation $\overset{*}{\rightarrow}$ over $T(\Sigma) \times T(\mathcal{C})$ is contained in $\twoheadrightarrow$. As in [2, 39, 40, 41, 48] we refer to this property as the *completeness* of $\twoheadrightarrow$. The fact that we limit $\twoheadrightarrow$ to compute values is not a limitation. The relation $\twoheadrightarrow$ could be easily extended to also compute normal forms different from values. However, these normal forms represent exceptional computations in constructor-based systems and we *choose* to fail to compute them, should they exist. Our choice becomes an asset in the presence of incompleteness and/or non-determinism, since it allows us to single out undesirable answers. This feature supports a programming technique exploited, for example, in [2, 41] through the use of an *if-then* (no *else*) operation, i.e., an operation failing to produce an "acceptable" result for a false argument of the test.

We will further address this problem, within the bounds of the current discussion, later.

Theorem 3 For any ground term t and value v, if $t \overset{*}{\rightarrow} v$, then $t \twoheadrightarrow v$.

Proof The inductive sequentiality of $\mathcal{R}$ implies that values are normal forms [1], hence v is a normal form of t. Similarly, the inductive sequentiality of $\mathcal{R}$ implies that $\mathcal{R}$ is confluent, thus the normal form of t is unique. Lemma 4 implies that the goal ${:}{-}\, t \twoheadrightarrow T$, where T is a variable, cannot fail. We now prove that it succeeds. If the goal does not succeed, then there exists a state of a computation of ${:}{-}\, t \twoheadrightarrow T$, say $\mathcal{T}$, with an arbitrary large number of $\Rightarrow$-nodes. Each $\Rightarrow$-node in $\mathcal{T}$, except 3.2-nodes, contributes to a reduction step in ℓ. The chain of leftmost descendants of a

3.2-node cannot consist exclusively of 3.2-nodes, since the left argument of $\Rightarrow$ in the left child of a 3.2-node is a proper subterm of the left argument of $\Rightarrow$ in its parent node. Thus, if $:\!-\, t \twoheadrightarrow T$ does not succeed, then $|\ell|$ is unbounded. Since only needed redexes are reduced in ℓ, this would imply, contrary to the hypothesis, that t has no normal form. Thus $:\!-\, t \twoheadrightarrow T$ succeeds and T is instantiated to some value t'. Condition 1 of Lemma 3 implies that $t \overset{*}{\rightarrow} t'$. The confluence of $\mathcal{R}$ implies that $t' = v$. □

Theorem 3 implies that one can use resolution to compute reduction sequences. Such a completeness result, however, does not imply the existence of a simple and/or efficient algorithm. In fact, the choice of a redex and/or a rewrite rule involved in a reduction corresponds to a similar choice of a clause involved in a resolution. Unless appropriate choices are made, the solution of a goal may be missed. In other words, the completeness result does not necessarily suggest which choices should be made. Rather, it guarantees that if all possible choices are carried on in parallel, then a solution to a goal is eventually found, if one exists. This strategy is sometimes unacceptable for a programming language.

Lemma 4 is an important complement to our completeness result, since it suggests how to make always the "right" choice. The desired clause is the first match in the textual order of the program. This is the strategy employed by Prolog interpreters and compilers. This situation is very appealing for a programming language, since it guarantees efficient operational completeness. By contrast, some other attempts of integrations of functional and logic programming [15, 35] offer stronger completeness results, but their corresponding implementations must sacrifice completeness for the sake of efficiency or vice versa. We will further discuss this point later.

6 Practical Considerations

In this section we briefly address three problems arising in the application of the previous results: (1) how to test inductive sequentiality and translate rewrite rules rather than definitional trees, (2) how to deal with incompletely defined operations, and (3) how to improve the code generated by the procedure *Translate*.

6.1 Trees from Rules

Definitional trees are hard to write by hand and hard to read, unless displayed in pictorial representation. Often, the most natural input of our transformational approach is a set of rewrite rules. Thus, we are faced with the problems of verifying the inductive sequentiality of a rewrite system and constructing a definitional tree from the defining rules of each operation of the system.

We accomplish both tasks simultaneously. We use an algorithm that builds a definitional tree from the defining rules of an operation, if a definitional tree exists, and consequently ensures the inductive sequentiality of a rewrite system. There exist several algorithms for this task. One algorithm attempts to build definitional trees top-down. This algorithm is a variation of the algorithms described in [31, 52] for checking the completeness and/or non-ambiguity status of the rewrite rules defining an operation. For example, Kounalis' algorithm builds a tree which may not be definitional with respect to the operation's rules. Our algorithm, based on a theorem in [1] concerning *pure columns*, enforces definitionality by a different selection of the occurrences which make the definitional tree grow.

Another algorithm attempts to build definitional trees bottom-up. This algorithm, which uses backtracking, is very compact and conceptually much simpler than the previous one. Its implementation in Prolog is only a dozen lines long. Since most operations are defined by a small number of rules and the depths of their left sides are generally small as well, the amount of backtracking is acceptable. An advantage of backtracking is that all the non-isomorphic trees of an operation can be computed and, as discussed in [3], this information can be used by parallel interpreters or compilers to rewrite in parallel while preserving full laziness.

6.2 Incompleteness

An artifact of our definition of inductive sequentiality is that operations should always be complete. Occasionally this is inconvenient either because some operations are conventionally undefined on some inputs, for example the head of the empty list, or because we do want to define operations on inputs that are not going to occur, for example see the operation *show* in the Appendix.

Order-sorted algebras [16] are a plausible solution to the requirement that any operation must be completely defined. However, the considerable machinery of this approach is not necessary for our purpose. A definitional tree of an incomplete operation differs from one characterized by Definition 2 by the absence of some *rule* nodes. "Missing" *rule* nodes affect only the existence conditions of Lemmas 2 and 4. All other results of this note keep holding. In the remainder of this section we give a (less formal) proof of this claim.

The uniqueness conditions of Lemmas 2 and 4 are obviously unaffected by the incompleteness of the system. The soundness of both $\Rightarrow$ and $\twoheadrightarrow$, proved by Lemmas 1 and 3, and the laziness of both $\Rightarrow$ and $\twoheadrightarrow$, proved by Theorem 2 and Lemma 3, are unaffected by the incompleteness of some operation, since we derive these results from the existence of a proof tree. That is, the possible incompleteness of the system does not show up in the computations on which the proofs of soundness and laziness rely.

The completeness of $\twoheadrightarrow$, proved by Theorem 3, is unaffected as well. However, our proof of this result is no longer valid. In fact, we assume the lack of failures, whereas the incompleteness of the system could falsify this assumption. We need further analysis of the properties of ℓ-reductions to show the truth of our claim.

Our definition of ℓ-reduction is based on proof trees. However, the procedure *Reduce*, which defines a ℓ-reduction, and Theorem 1, which establishes its basic properties, are based on states of computations. That is, they define a mapping between states of computations and "*states* of ℓ-reductions", or in other words, between partially computed proof trees and equally partially computed ℓ-reductions. In particular, an attempt to solve some goal of a state of a computation corresponds to an attempt to reduce some term in a state of a ℓ-reduction. So far we have used only a restriction of this mapping, namely that whose domain is restricted to proof trees. Within this restriction, no attempt to solve a goal, and consequently to reduce a term, ever fails.

We need to show that at any attempt to reduce the subterm of t' at p, for some term t' and some occurrence p of t', in a *state* of a ℓ-reduction $t \overset{*}{\rightarrow} t'$, for some term t, the subterm of t' at p is operation-rooted and its reduction is needed to reduce t', and hence t, to a *value*. The subterm of t' at p must be f-rooted, for some operation f, because when f is completely defined (and we can always complete f, if it is incomplete) the attempt succeeds and the claim holds by Lemma 1. Our argument to prove the need of reducing t'/p resembles a structural induction on p. If p is the empty occurrence of t' and t'/p, that is t', cannot be reduced then, by the uniqueness conditions of Lemmas 2 and 4, t' cannot be reduced to a constructor-rooted term, hence to a value, and consequently no descendants of t will ever be a value. If p is not empty, then the reduction of t'/p (eventually to a constructor-rooted term) is needed to reduce some operation-rooted superterm of t', i.e., a term at some occurrence p' of t', where p' is a proper prefix of p, and the reduction of this superterm is eventually needed to reduce t' to a value. Thus, also in this case, unless t'/p is reducible, any descendant of t will always contain some operation symbol.

Going back to the proof of Theorem 3, we assume that t is reducible to some value v and we know that v is unique. Since $\twoheadrightarrow$ attempts to reduce only terms which must be reduced to reduce t to a value, and since the terms selected at any attempt can be reduced by at most one rule, a failure to reduce any such term would contradict the hypothesis. Thus, if $t \overset{*}{\rightarrow} v$ for some ground term t and value v, there are no failures in the computation of $:\!- \; t \twoheadrightarrow T$, where T is a variable. The rest of the proof of Theorem 3 remains unchanged.

6.3 Code Improvement

To simplify our presentation, we kept the output of the procedure *Translate* as simple as possible. An actual translator, based on the pseudo-code of the procedure *Translate*, can perform several code improvements. We have already mentioned the possibility of "sharing" the common subterms of a term to minimize the length of a ℓ-reduction. We now discuss some simple improvements assuming that Prolog [10] is the target language of the translation. One optimization concerns the introduction of cuts. By the uniqueness conditions of Lemmas 2 and 4, the choice of a program clause for a resolution can be committed as soon as the head of a clause matches a goal. Some Prolog implementations take advantage of this situation, detected by the explicit presence of cuts, to avoid choice points. When cuts are used, Lemmas 2 and 4 (point 2) imply that the order of the clauses in the Prolog program must be the textual order in which clauses are output by the procedure *Translate*.

A second improvement concerns incompleteness. This condition is easily recognized during the construction of a definitional tree from the rules of an incompletely defined operation. We encode this condition, in the definitional tree being constructed, by means of an "empty" *rule* node [3]. In a definitional tree there is enough information to know the left side of the rule which should have been present in the node if the node were not empty. In the most straightforward translation of a definitional tree empty nodes produce no output. However, useful code can be generated from empty *rule* nodes. For example, the operation returning the head of a list is defined by only one rule:

$$head([E|\_]) \to E \tag{7}$$

The operation *head* is incomplete because no rule matches the term *head*([ ]). The following code is well-suited to the incompleteness of *head*.

```
head([]) => _ :- !, fail.
head([A|_]) => A :- !.                                    (8)
head(A) => B :- !, A => C, head(C) => B.
```

The first clause is generated by the "empty" *rule* node of a definitional tree of *head*. The first clause is not strictly necessary and could be eliminated; however, its presence makes the code more versatile. In fact, any goal matching the first clause also matches the third one. However, any goal matching the first clause cannot be solved by the third one, as proved in Lemma 2. Thus, the first clause prevents the use of the third one, which would take longer to fail. Furthermore, if only deterministic computations are executed, the first clause could be augmented with instructions to raise an exception or to issue the warning that the computation fails because of an attempt to compute the head of an empty list.

7 Related work

Several aspects of our work have been investigated by others. In this section we briefly report on related research. The emphasis of our analysis is on the integration of functional and relational computing paradigms, particularly with respect to lazy rewriting.

7.1 Lazy Evaluation

Narain [39, 40, 41] describes "a technique for doing lazy evaluation in logic" similar to ours. Antoy [2] generalizes the concept of definitional tree to ambiguous rewrite systems and accordingly introduces a generalized procedure *Translate*. Naish [38] describes an approach and a corresponding implementation to the integration of functional and relational programming in which lazy evaluation is an optional feature. Other related research includes the functional-logic programming languages *K-LEAF* [4, 6, 15] and *BABEL* [32, 36, 37]. These undertakings are more ambitious than ours and our discussion is limited to their components of lazy rewriting.

The technique discussed in [40, 41] is based on a rewrite system denoted F^* and its deterministic restriction DF^*. F^* and DF^* obey the constructor discipline and two additional conditions, namely, in F^* the height of the left side of each rule must be at most 2, and in DF^* if there is a rule $l \to r$ defining some operation f and, for some integer i, l/i is a variable, then for any rule $l' \to r'$ defining f, l'/i must be a variable too. Rules satisfying the first condition are generally referred to as *flat* or *shallow*. We will refer to the second constraint as the *sub-unifiability* condition, since it implies that (the left sides of) the rewrite rules are pairwise *non strictly sub-unifiable* [13]. Any DF^* system is inductively sequential, but not vice versa. We do not prove this claim, rather we discuss its implications.

For example, the operation "less than or equal to" shown in display (4) violates the sub-unifiability condition and the operation "integer division by 2" shown below violates the flatness condition.

$$\begin{aligned} \mathit{half}(0) &\to 0 \\ \mathit{half}(s(0)) &\to 0 \\ \mathit{half}(s(s(X))) &\to s(\mathit{half}(X)) \end{aligned} \tag{9}$$

A work-around for both these limitations consists in modifying some rewrite rules and in introducing other rules defining some auxiliary operations. Moreno-Navarro et al. [36] propose a transformation of rewrite systems that yields programs satisfying both the flatness and sub-unifiability conditions. If the input to the transformation is a non-ambiguous program, then the output is in DF^*. We show in display (10) the result of this transformation applied to the rules of the operation *le* defined in display (4). It is easy to verify that the constraints imposed by DF^* are satisfied.

$$\begin{aligned} \mathit{le}(0,Y) &\to \mathit{true} \\ \mathit{le}(s(X),Y) &\to \mathit{aux}(s(X),Y) \\ \mathit{aux}(s(X),0) &\to \mathit{false} \\ \mathit{aux}(s(X),s(Y)) &\to \mathit{le}(X,Y) \end{aligned} \tag{10}$$

The sub-unifiability and flatness conditions make it straightforward to determine if and how much an argument of an f-rooted term t, for some operation f, must be evaluated, since the answer depends on f, but not on the rule used to reduce t.

The drawback of the transformation is that auxiliary operations make programs textually longer, harder to understand if they must be written or read by humans, and less efficient. Reduction sequences involving auxiliary operations contain extra reduction steps, and executing these steps also requires the creation of terms that will eventually become garbage.

Narain reports that the "main advantage [of his method] over previous methods is its considerable efficiency, both from a theoretical as well as [an] implementation point of view" [41]. Our approach is more efficient. No relation between the length of a reduction and the complexity of its logic implementation is discussed in [39, 40, 41]. Thus, a comparison with the relation stated at point 3 of Lemma 3 is impossible. There are two differences, potentially affecting efficiency, between the translation defined in [41] and ours. One concerns the need for auxiliary functions, as discussed above. The other concerns the logic programs generated from the rewrite systems. Narain's translation generates Horn clauses containing two relations, denoted *nf* and *reduce*, that roughly correspond to our relations $\twoheadrightarrow$ and $\Rightarrow$, respectively, that is *nf* computes normal forms and *reduce* computes head normal forms.

With respect to our approach, *nf* calls *reduce* also for normal forms, that is, in situations in which no reduction is necessary. These resolution steps are avoided by our method. *Reduce* applied to non head normal forms always generates three immediate subgoals, whereas in our approach fewer immediate subgoals or possibly none are sometimes generated. Thus, in some situations our resolution is shorter. The cumulative effects of these differences are difficult to quantify. Thus, we compare the proof trees generated for a simple computation. Let t be $append([a,b],[c,d])$, where a, b, c, and d are constant constructors. The normalization of t with Narain's program [41] requires a proof tree of size 27. By contrast, the normalization of t with the program generated by the procedure *Translate* requires a proof tree of size 15. Furthermore, during the construction of the proof tree with Narain's approach the number of calls, "redos", and failures executed by the logic program are respectively 54, 6, and 27. By contrast, neither "redos" nor failures ever occur with our approach. Our computation takes 1/3 of the time of Narain's. We conclude this informal comparison by remarking that this difference in performance is not due to the presence of auxiliary operations, since there are none in the program. If auxiliary operations need to be included, the difference in performance may become larger.

Our translation generates several Horn clauses per rewrite rule, between 2 and 3 for common examples. By contrast, there is one clause defining *reduce* for each symbol of the signature. The proliferation of clauses and the fact that they all define the same predicate are potential slow-downs. However, our logic programs are well-suited for Prolog interpreters and compilers that index the clauses of a predicate according to the principal functor of the first argument in the head of the clause, and for those that take advantage of the *a priori* knowledge that backtracking is never required.

Antoy [2] discusses a variation of definitional trees that accommodates ambiguous rewrite systems. This extension is achieved by storing a possibly empty set of rewrite rules, all with the same left side, in each leaf *and* branch of a generalized definitional tree. Definitional trees can be regarded as simple instances of these generalized definitional trees. The procedure *Translate* is generalized with little effort. This approach, similar to F^* [41], supports non-determinism. Both the soundness of $\Rightarrow$ and the soundness and completeness of $\twoheadrightarrow$, in the sense discussed previously, are preserved. However, many other important results proved in this note are lost. In particular, Theorem 2 cannot be formulated, since the concept of needed redex is not defined for ambiguous rewrite systems. Furthermore, there may exist distinct ℓ-reductions of a term to any of its normal forms, and the Prolog search strategy is no longer complete. Although the efficiency of this variation seems competitive, as some time measures reported in [2] show, the method is lazy only on the deterministic portions of a program. In general, some reduction steps executed during a computation may not be necessary to compute a normal form.

An additional difficulty of non-innermost evaluation for non-deterministic computations is the characterization of appropriate semantics. For example, consider the following system, where the constructors of the natural numbers are denoted 0 and s, and the symbol “1” is a shorthand for $s(0)$.

$$\begin{array}{rcl} \mathit{flip} & \rightarrow & 0 \\ \mathit{flip} & \rightarrow & 1 \\ \mathit{add}(0,Y) & \rightarrow & Y \\ \mathit{add}(s(X),Y) & \rightarrow & s(\mathit{add}(X,Y)) \\ \mathit{double}(Z) & \rightarrow & \mathit{add}(Z,Z) \end{array} \tag{11}$$

The operation *flip* non-deterministically returns either 0 or 1. The operation *double* doubles its argument using addition. The outermost evaluation of the term $\mathit{double}(\mathit{flip})$ goes through $\mathit{add}(\mathit{flip},\mathit{flip})$, and eventually yields some value in $\{0,1,2\}$, which might not be the intended behavior. The situation stems from the non-linearity of the right side of the rule defining *double*. Using innermost evaluation, the value of *double* is an even number for any argument. Narain [39, 40, 41], and for different reasons Naish [38], advocate *sharing*, i.e., when a term is reduced, all its copies are simultaneously and equally reduced. Sharing decreases the length of some computations, and without sharing most examples of generate-and-test problems discussed in [2, 39, 40, 41] become harder to code. However, sharing would make the relation $\twoheadrightarrow$ incomplete in the sense discussed in this note. Thus, it may be inappropriate in some cases and it should be an optional attribute of a computation.

Naish [38] describes an extension to NU-Prolog which allows evaluable functions to be defined using [oriented] equations. The system includes coroutining, parallelism, higher order functions, and lazy evaluation. Basically, an n-ary function is transformed in an $(n+1)$-ary predicate according to the well-known *flattening* technique which produces efficient Prolog code under the assumption of eager evaluation. Lazy evaluation is an optional feature of the system, since it makes the translation more complex and introduces some overhead even in non-lazy functions. This approach, similar to Narain's and ours, is indirect, i.e., translates rewrite rules in standard Prolog code. The implementation is efficient with respect to indexing and convenient to use, however, its semantics are not yet well-developed, for example, the conditions ensuring the soundness, completeness, and laziness of the computations.

The programming languages *K-LEAF* [4, 6, 15] and *BABEL* [32, 36, 37] integrate a rich functional component, backed-up by rigorous semantics, in a logic programming environment. Both languages follow the direct approach, since they rely on specialized *WAM*-like machines [56], however, for *K-LEAF*, Cheong [9] discusses a corresponding direct approach. The underlying computational paradigm of these languages is narrowing [46, 47]. *Narrowing* a term t means computing a minimum substitution σ such that $\sigma(t)$ becomes reducible and then computing such a reduct. Thus, rewriting can be seen as a special case of narrowing. In the appendix we briefly discuss in an example how our approach can accommodate narrowing.

The functional components of both *K-LEAF* and *BABEL* programs consist, with minor deviations, of weakly-orthogonal, conditional, constructor-based rewrite systems [29]. Systems in this class are confluent, but not necessarily sequential. This class properly contains the inductively sequential systems. The semantics of both languages are well-developed and include soundness and completeness results that are more general than ours, since the class of rewrite systems underlying these the languages is larger than ours and narrowing encompasses rewriting. However, this generality places heavy demands on the implementation.

In *BABEL*, rewrite programs are translated in a form, called *uniform* [35], which eases execution. An input example of this translation and its corresponding output are shown in displays (4) and (10) respectively. Uniform programs, except for the possibility of overlapping rules, closely resemble DF^* programs. A notion of lazy redex is provided for uniform programs, and lazy redexes are to some extent precomputed during the translation of a program to uniform form. However, rewriting lazy redexes in terms with a normal form is neither sufficient nor necessary to guarantee termination. Termination depends as well on the non-deterministic selection of the clauses used for resolution. The execution of *BABEL* programs is reasonably efficient, although the introduction of auxiliary operations and a 3-way unification algorithm used to detect lazy redexes are potentially costly.

For the execution of *K-LEAF* programs both rewrite rules and goals are translated in a form called *flat*. Some degree of laziness is achieved by detecting certain "demanded" outermost redexes. As in *BABEL*, rewriting these redexes in terms with a normal form is neither sufficient nor necessary to guarantee termination. The procedure for solving goals is non-trivial. There is some overhead to detect demanded redexes and to suspend and resume a resolution when one such redex is being reduced. Operational completeness requires a breadth-first search of the SLD-tree.

In conclusion, the generosity of the languages offered by *BABEL* and *K-LEAF* implies that, even when the goal is limited to rewriting, either the implementation sacrifices completeness to efficiency or vice versa. By contrast, we consider a more limited class of rewrite systems, but our implementation provides both completeness and efficiency.

7.2 Rewriting and Logic Computing

The implementation of rewrite systems in Prolog has received considerable attention and the large body of literature on the argument makes our discussion quite superficial. Early work, summarized in [45], was directed toward prototyping algebraic specifications through their direct implementation and/or through the use of Prolog-supported automatic theorem provers. More recent proposals aim at the integration of functional and/or equational programs into logic programs and emphasize narrowing. A collection of papers on the first goal is in [11], a comprehensive discussion on the second one is in [22], an analysis of rewriting in Prolog, including a survey of several reduction strategies and a detailed comparison of various approaches, is in [8].

Van Emden and Yukawa [55] present two approaches to the implementation of rewrite rules in Prolog. One approach, referred to as *vEM* is concerned with systems substantially similar to those we consider—any operation discussed in the *vEM* examples of [55] is inductively sequential. With respect to *vEM* we do not need *Noetherianity*, nor *strong canonicity*, nor do we place any restriction on or require any preprocessing of the terms to be reduced. As far as efficiency is concerned, the size of a *vEM* proof tree for the normalization of a term t is at least equal to the length of the implicitly computed reduction sequence, and the required preprocessing of t implies the traversal of t. Thus, in the worst case, the theoretical slow-down of our method with respect to *vEM* is dominated by a factor not greater than 2. On the other hand, *vEM* cannot handle some of our cases, for example those involving infinite structures, and there are cases accepted by both methods where *vEM* is either slower than our method by a factor arbitrarily large or, more likely, will eventually run out of available resources.

7.3 Rewrite Systems and Normalization

Tree-like structures similar to definitional trees, but without explicit representation of design decisions, are discussed in [31, 52]. These proposals, whose origins may be traced to an inductive definition of completeness [27], are concerned with the detection and/or correction of incompleteness and/or ambiguity in the defining rules of an operation. Algorithms for these tasks are based on the construction of structures similar to definitional trees. Definitional trees have the distinctive characteristic of storing occurrences, which are used to reason about reduction strategies. Definitional trees for classes of systems larger than the inductively sequential ones are discussed in [2, 3].

The binary choice strategy is introduced in [1] jointly with another strategy for designing algebraic specifications which are neither under- nor over-specified. It is shown that within certain large classes of operations, for example the unary and binary functions over the natural numbers, if an operation is completely and non-ambiguously defined, then it is inductively sequential. This seems to support the claim that inductive sequentiality is not an overly restrictive property.

Seminal studies on normalizing reduction strategies appear in [25, 42]. The problem is further analyzed and/or summarized in [8, 25, 30, 43]. The difficulty of the normalization problem led to a number of variations of the concept of sequentiality [7, 21, 30, 44, 51]. We deal with a particular class of sequential rewrite systems, which seem to be of practical interest. Our solution has two distinctive characteristics: (1) sufficient conditions for the application of our method are provided at design time or can be easily checked from syntactic characteristics of the rewrite rules, and (2) we "compile" a system only once, initially, and no computation of indices is needed at "run-time", i.e., when a term is reduced. The latter makes our approach potentially more efficient than those requiring specialized forms of pattern matching [20] or run-time operations associated to a demand-driven approach [15, 35]. Effective practical normalizing strategies are unavailable in general, as pointed out in [28, 29, 30].

Sekar and Ramakrishnan [48] address normalization in non-sequential rewrite systems through the concept of *necessary set* of redexes. This approach, which requires some form of parallel computation, may well be appropriate for logic programming.

8 Concluding Remarks

Our approach to lazy rewriting in logic improves several aspects of existing similar methods. A unique characteristic of our method is that we do not flatten functional nestings, a condition which in extreme cases makes our resolutions shorter by an arbitrarily large factor. Similar to Narain's approach, but contrary to most others, the rewrite programs underlying our approach allow us to characterize and easily detect certain redexes in a term, such that rewriting these redexes is a necessary and sufficient condition to compute the normal form of the term, if one exists. In addition to Narain's approach, however, we extend the class of acceptable rewrite programs by removing two conditions limiting the expressiveness of the rewrite rules. Perhaps surprisingly, our extension allows us to produce logic programs which are more versatile and more efficient.

With respect to approaches integrating functional and relational computations and allowing lazy rewriting we offer rigorous semantics, an expressive language, efficient operational completeness, a realistic estimate of the cost of laziness, and the ability to detect and react to exceptions occurring during a computation. With respect to the approaches which do not allow laziness we offer the capability of programming with infinite structures, an arbitrarily large increase in efficiency in the best case, and a measurable, moderate loss of efficiency in the worst case.

Acknowledgment

I wish to thank Maarten van Emden, Corrado Moiso, Sanjai Narain, Catuscia Palamidessi, and Mario Rodríguez-Artalejo for help with technical aspects of this paper.

References

[1] Sergio Antoy. Design strategies for rewrite rules. In S. Kaplan aand M. Okada, editor, *CTRS'90*, pages 333–341, Montreal, Canada, June 1990. Lect. Notes in Comp. Sci., Vol. 516.

[2] Sergio Antoy. Non-determinism and lazy evaluation in logic programming. In T. P. Clement and K.-K. Lau, editors, *LOPSTR'91*, pages 318–331, Manchester, UK, July 1991. Springer-Verlag.

[3] Sergio Antoy. Definitional trees. In *ALP'92*, pages 143–157, Volterra, Italy, September 1992. Lect. Notes in Comp. Sci., Vol. 632.

[4] R. Barbuti, M. Bellia, and G. Levi. LEAF: A language which integrates Logic, Equations and Functions. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 201–237. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] P. G. Bosco, C. Cecchi, and C. Moiso. An extension of WAM for K-LEAF. In *Sixth International Conf. on Logic Programming*, pages 318–333, Lisboa, Portugal, 1989.

[6] P. G. Bosco, E. Giovannetti, G. Levi, and Catuscia Palamidessi. A complete semantic characterization of K-LEAF, a logic language with partial functions. In *IEEE Symp. on Logic Programming*, pages 318–327, 1987.

[7] G. Boudol. Computational semantics of term rewriting systems. In Maurice Nivat and John C. Reynolds, editors, *Algebraic methods in semantics*, chapter 5. Cambridge University Press, Cambridge, UK, 1985.

[8] L. G. Bouma and H. R. Walters. Implementing algebraic specifications. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 5. Addison-Wesley, Wokingham, England, 1989.

[9] P. H. Cheong. Compiling lazy narrowing into Prolog. *New Generation Computing*, 1992. (to appear).

[10] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, second edition, 1984.

[11] Doug DeGroot and Gary Lindstrom, editors. *Logic Programming: Functions, Relations, and Equations*, Englewood Cliffs, NJ, 1986. Prentice-Hall.

[12] N. Dershowitz and J. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North Holland, Amsterdam, 1990.

[13] Rachid Echahed. On completeness of narrowing strategies. *Theoretical Computer Science*, 72:133–146, 1990.

[14] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[15] Elio Giovannetti, Giorgio Levi, Corrado Moiso, and Catuscia Palamidessi. Kernel LEAF: a logic plus functional language. *The Journal of Computer and System Sciences*, 42:139–185, 1991.

[16] J. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics of order-sorted algebras. In W. Brauer, editor, *CALP'85*, 1985. Lect. Notes in Comp. Sci., 194.

[17] Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Menlo Park, CA, 1988.

[18] A. Hansson, S. Haridi, and S.-A. Tärnlund. Properties of a logic programming language. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 267–280. Academic Press, 1982.

[19] Sidney Hantler and James King. An introduction to proving the correctness of programs. *Computing Surveys*, 8:331–353, 1976.

[20] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *JACM*, 29:68–95, 1982.

[21] C. M. Hoffmann and M. J. O'Donnell. Implementation of an interpreter for abstract equations. In *11th ACM Symposium on the Principle of Programming Languages*, Salt Lake City, 1984.

[22] S. Hölldobler. *Foundation of Equational Logic Programming.* Springer-Verlag, Berlin, 1989. Lect. Notes in Artificial Intelligence, Vol. 353.

[23] Ellis Horowitz. *Fundamentals of Programming Languages.* Computer Science Press, Rockville, MD, second edition, 1984.

[24] Paul Hudak. Conception, evolution, and application of functional programming languages. *Computing Surveys*, 21:359–411, 1989.

[25] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson.* MIT Press, Cambridge, MA, 1991. Previous version: Call by need computations in non-ambiguous linear term rewriting systems, Technical Report 359, INRIA, Le Chesnay, France, 1979.

[26] Gérard Huet. Confluent reductions: Abstract properties and applications to term-rewriting systems. *JACM*, 27:797–821, 1980.

[27] Gérard Huet and Jean-Marie Hullot. Proofs by induction in equational theories with constructors. *JCSS*, 25:239–266, 1982.

[28] J. R. Kennaway. Sequential evaluation strategies for parallel-or and related reduction systems. *Annals of Pure and Applied Logic*, 43:31–56, 1989.

[29] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992. Previous version: Term rewriting systems, Technical Report CS-R9073, Stichting Mathematisch Centrum, Amsterdam, 1990.

[30] Jan Willem Klop and Aart Middeldorp. Sequentiality in orthogonal term rewriting systems. Technical Report CS-R8932, Stichting Mathematisch Centrum, Amsterdam, The Netherlands, 1989.

[31] Emmanuel Kounalis. Completeness in data type specifications. In *EUROCAL'85*, pages 348–362. Springer-Verlag, 1985. Lect. Notes in Comp. Sci., Vol. 204.

[32] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based implementation of a functional language. In *ESOP'90*, pages 279–290, 1990. Lect. Notes in Comp. Sci., 432.

[33] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1984.

[34] Luc Maranget. Optimal derivation in weak lambda-calculi and in orthogonal terms rewriting systems. In *17th Annual Symp. on Principles of Prog. Languages*, pages 255–269. ACM, 1990.

[35] J. J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Conf. on Algebraic and Logic Progr.*, 1990. Lect. Notes in Comp. Sci., 463.

[36] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. BABEL: A functional and logic programming language based on constructor discipline and narrowing. In *Conf. on Algebraic and Logic Progr.*, pages 223–232, 1989. Lect. Notes in Comp. Sci., 343.

[37] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[38] Lee Naish. Adding equations to NU-Prolog. In *PLILP'91*, pages 15–26, Passau, Germany, August 1991. Lect. Notes in Comp. Sci., Vol. 528.

[39] Sanjai Narain. A technique for doing lazy evaluation in logic. *The Journal of Logic Programming*, 3:259–276, 1986.

[40] Sanjai Narain. *LOG(F): An optimal combination of logic programming, rewriting, and lazy evaluation.* PhD thesis, University of California, Los Angeles, CA, 1988.

[41] Sanjai Narain. Lazy evaluation in logic programming. In *Third IEEE Conference on Computer Languages*, pages 218–227, New Orleans, 1990.

[42] Michael J. O'Donnell. *Computing in Systems Described by Equations.* Springer-Verlag, 1977. Lect. Notes in Comp. Sci., Vol. 58.

[43] Michael J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, 1985.

[44] M. J. Oyamaguchi. Sufficient sequentiality: A decidable condition for call-by-need computations in term rewriting systems. Technical report, Mie University, 1987.

[45] H. Petzsch. Automatic prototyping of algebraic specifications using Prolog. In *Recent Trends in Data Type Specification*, pages 207–223. Third Workshop on Theory and Applications of Abstract Data Types, Springer-Verlag, 1985.

[46] Uday S. Reddy. Narrowing as the operational semantics of functional languages. In *International Symposium on Logic Programming*, pages 138–151. IEEE Computer Soc. Press, 1985.

[47] Uday S. Reddy. Functional logic languages, part i. In *Workshop on Graph Reduction*, pages 401–425, 1987. Lect. Notes in Comp. Sci., Vol. 516.

[48] R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 230–241, Philadelphia, PA, June 1990.

[49] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, MA, 1986.

[50] P. A. Subrahmanyam and Jia-Huai You. FUNLOG: a computational model integrating logic programming and functional programming. In Doug DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 157–198. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[51] Satish Thatte. A refinement of strong sequentiality for term rewriting with constructors. *Information and Computation*, 72:46–65, 1987.

[52] Jean-Jacques Thiel. Stop losing sleep over incomplete data type specifications. In *11th Annual Symp. on Principles of Prog. Languages*, pages 76–82. ACM, 1984.

[53] D. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[54] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.

[55] Maarten H. van Emden and Keitaro Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 4:265–288, 1987.

[56] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Menlo Park, CA, 1983.

Appendix

Example 1

The following example is adapted from [23, p.368]. Laziness saves both computation time and memory occupation. The predicate *samefringe* determines whether or not two trees have the same fringe. Lists are denoted in standard Prolog notation. The predicate *eq* determines whether or not two list elements are equal and is defined, only in the clauses, through predefined symbols.

$\mathit{and}(\mathit{false}, \_) \to \mathit{false}$
$\mathit{and}(\mathit{true}, A) \to A$
$\mathit{samefringe}(A, B) \to \mathit{eqlist}(\mathit{flatten}(A), \mathit{flatten}(B))$
$\mathit{append}([], A) \to A$
$\mathit{append}([A|B], C) \to [A|\mathit{append}(B, C)]$
$\mathit{eqlist}([], []) \to \mathit{true}$
$\mathit{eqlist}([], [\_|\_]) \to \mathit{false}$
$\mathit{eqlist}([\_|\_], []) \to \mathit{false}$
$\mathit{eqlist}([A|B], [C|D]) \to \mathit{and}(\mathit{eq}(A, C), \mathit{eqlist}(B, D))$
$\mathit{flatten}(\mathit{leaf}(A)) \to [A]$
$\mathit{flatten}(\mathit{branch}(A, B)) \to \mathit{append}(\mathit{flatten}(A), \mathit{flatten}(B))$

The following clauses, with the exception of `eq`, are the output of the procedure *Translate* on input the above rules.

```
and(false,_) => false .
and(true,false) => false .
and(true,true) => true .
and(true,A) => B  :-  A => B .
and(A,B) => C  :-  A => D , and(D,B) => C .
samefringe(A,B) => C  :-  eqlist(flatten(A),flatten(B)) => C .
append([],[]) => [] .
append([],[A|B]) => [A|B] .
append([],A) => B  :-  A => B .
append([A|B],C) => [A|append(B,C)] .
append(A,B) => C  :-  A => D , append(D,B) => C .
eqlist([],[]) => true .
eqlist([],[_|_]) => false .
eqlist([],A) => B  :-  A => C , eqlist([],C) => B .
eqlist([_|_],[]) => false .
eqlist([A|B],[C|D]) => E  :-  and(eq(A,C),eqlist(B,D)) => E .
eqlist([A|B],C) => D  :-  C => E , eqlist([A|B],E) => D .
eqlist(A,B) => C  :-  A => D , eqlist(D,B) => C .
flatten(leaf(A)) => [A] .
flatten(branch(A,B)) => C  :-  append(flatten(A),flatten(B)) => C .
flatten(A) => B  :-  A => C , flatten(C) => B .
```

```
eq(A,A) => true  :-  ! .
eq(_,_) => false .
```

We now trace the major steps of the computation of *samefringe* applied to the trees

$$branch(leaf(a), branch(leaf(c), leaf(b)))$$

and

$$branch(branch(leaf(a), leaf(b)), leaf(c)).$$

Eventually both terms are reduced to head normal forms, i.e. the trees are flattened enough to "look like" lists, so that *eqlist* can be applied. Such terms are

$$[a|append([], flatten(branch(leaf(c), leaf(b))))]$$

and

$$[a|append(append([], flatten(leaf(b))), flatten(leaf(c)))].$$

eqlist cannot yet determine whether or not these lists are equal, thus their tails are again reduced to head normal forms. The resulting lists are

$$[c|append([], flatten(leaf(b)))]$$

and

$$[b|append([], flatten(leaf(c)))].$$

Now *eqlist* can determine that the two lists differ and consequently the trees have not the same fringe.

Example 2

The following example is adapted from [17, p.52], see also [43, p.140]. Since an infinite structure is involved, laziness is needed for computing the result. The rewrite rules are incomplete, i.e. some leaves of some definitional trees are empty. We could complete these operations with arbitrary rules. The clauses generated from these arbitrary rules are unnecessary in the logic program and can be dropped, since we know that the prime numbers are infinite and the list of primes used by the computation never becomes empty. For the translation of these rules, we used a refinement of the procedure *Translate*, as discussed in section 6.2, which accepts definitional trees with "empty" *rule* nodes. We use the Prolog implementation of natural numbers, thus matching a term with an occurrence of s (successor) needs some adjustment and the predicate *factor* is defined, only in the clauses, through predefined symbols. This shows the integration of the output of *Translate* with hand coded programs and in particular of lazy with eager evaluation.

$\mathit{ints\_from}(A) \rightarrow [A|\mathit{ints\_from}(s(A))]$
$\mathit{show}(0, [\_|\_]) \rightarrow []$
$\mathit{show}(s(A), [B|C]) \rightarrow [B|\mathit{show}(A, C)]$
$\mathit{sieve}([A|B]) \rightarrow [A|\mathit{sieve}(\mathit{filter}(B, A))]$
$\mathit{primes} \rightarrow \mathit{sieve}(\mathit{ints\_from}(2))$
$\mathit{filter}([A|B], C) \rightarrow \mathit{if}(\mathit{factor}(C, A), \mathit{filter}(B, C), [A|\mathit{filter}(B, C)])$
$\mathit{if}(\mathit{false}, \_, A) \rightarrow A$
$\mathit{if}(\mathit{true}, A, \_) \rightarrow A$

The clauses resulting from the translation of the above rules follow.

```
ints_from(A) => [A|ints_from(B)]  :-  B is A+1 .
show(0,[_|_]) => [] .
show(A,[B|C]) => [B|show(D,C)]  :-  A > 0 , D is A-1 .
show(A,[B|C]) => D  :-  A => E , show(E,[B|C]) => D .
show(A,B) => C  :-  B => D , show(A,D) => C .
sieve([A|B]) => [A|sieve(filter(B,A))] .
sieve(A) => B  :-  A => C , sieve(C) => B .
primes => A  :-  sieve(ints_from(2)) => A .
filter([A|B],C) => D  :-  if(factor(C,A),filter(B,C),[A|filter(B,C)]) => D .
filter(A,B) => C  :-  A => D , filter(D,B) => C .
if(false,_,[]) => [] .
if(false,_,[A|B]) => [A|B] .
if(false,_,A) => B  :-  A => B .
if(true,[],_) => [] .
if(true,[A|B],_) => [A|B] .
if(true,A,_) => B  :-  A => B .
if(A,B,C) => D  :-  A => E , if(E,B,C) => D .
```

The remaining clauses of the program follow.

```
factor(A,B) => true  :-  0 is B mod A , ! .
factor(_,_) => false .
[] ->> [] .
[A|B] ->> [C|D]  :-  A ->> C , B ->> D .
A ->> A  :-  integer(A) , ! .
A ->> B  :-  A => C , C ->> B .
```

For example, the call $\mathit{show}(10, \mathit{primes}) \twoheadrightarrow T$ instantiates T to [2,3,5,7,11,13,17,19,23,29].

Example 3

The following example outlines how our approach is extended to accommodate narrowing. A minor change in the clauses generated by the procedure *Translate* allows us to attempt solving goals of the form $t_1 \twoheadrightarrow u_1, \ldots, t_n \twoheadrightarrow u_n$, for some $n > 0$, where for all i in $1, \ldots, n$, t_i is a term that can now contain variables and u_i is a constructor term. For example, referring to the operation *le* defined in display (4), goals that we banned in our previous discussion, but accept now, are $:\!-\, le(s(s(0)), X) \twoheadrightarrow Y$ and $:\!-\, le(X, Y) \twoheadrightarrow true,\ le(Y, X) \twoheadrightarrow true$. A Prolog program for solving these goals follows, where the predicate *nonvar* is discussed below.

```
0 ->> 0 .
s(A) ->> s(B)  :-  A ->> B .
true ->> true .
false ->> false .
A ->> B  :-  nonvar(A) , A => C , C ->> B .

le(0,_) => true .
le(s(_),0) => false .
le(s(A),s(B)) => C  :-  le(A,B) => C .
le(s(A),B) => C  :-  nonvar(B) , B => D , le(s(A),D) => C .
le(A,B) => C  :-  nonvar(A) , A => D , le(D,B) => C .
```

The only change with respect to the clauses generated by the procedure *Translate* consists in ensuring that when a goal $:\!-\, t \Rightarrow u$, for some terms t and u, is being solved, t is not an uninstantiated variable. This condition is easily established in Prolog using the metapredicate *nonvar*. The condition prevents considering substitutions mapping variables to non constructor terms, which are customarily excluded in constructor-based rewrite systems.

The behavior of a Prolog interpreter on the goals discussed above is the following. Throughout backtracking, three solutions are generated for the goal $:\!-\, le(s(s(0)), X) \twoheadrightarrow Y$, and then the program fails. The solutions are: $[0/X, \mathit{false}/Y]$, $[s(0)/X, \mathit{false}/Y]$, and $[s(s(\_))/X, \mathit{true}/Y]$. For the goal $:\!-\, le(X, Y) \twoheadrightarrow true, le(Y, X) \twoheadrightarrow true$ we obtain the following infinite sequence of solutions: $[0/X, 0/Y]$, $[s(0)/X, s(0)/Y]$, $[s(s(0))/X, s(s(0))/Y], \ldots$

The operational completeness based on Lemma 4 that we proved for rewriting is unachievable for narrowing. In general, the combination of a depth-first strategy and backtracking is insufficient to compute all the solutions of a goal,