Programming in Equational Logic: Beyond Strong Sequentiality*

R.C. Sekar† and I.V. Ramakrishnan
Dept. of Computer Science,
SUNY at Stony Brook, NY 11794
E-mail: sekar@thumper.bellcore.com, ram@cs.sunysb.edu

Abstract

Orthogonal term rewriting systems (also known as *regular* systems) provide an elegant framework for programming in equational logic. O'Donnell showed that the parallel-outermost strategy, which replaces all outermost redexes in each step, is complete for such systems. Many of the reductions performed by this strategy could be wasteful in general. A lazy normalization algorithm that completely eliminated these wasteful reductions by reducing only "needed redexes" was later developed by Huet and Levy. However, this algorithm required the input programs to be restricted to the subclass of *strongly sequential systems*. This is because needed redexes do not exist for all orthogonal programs, and even when they do, they may not be computable. It is therefore quite natural to ask whether it is possible to devise a complete normalization algorithm for the entire class that minimizes (rather than eliminate) the wasteful reductions. In this paper we propose a solution to this problem using the concept of a *necessary set* of redexes. In such a set, at least one of the redexes *must* be reduced to normalize a term. We devise an algorithm to compute a necessary set for any term not in normal form and show that a strategy that repeatedly reduces all redexes in such a set is complete for orthogonal programs. We also show that our algorithm is "optimal" among all normalization algorithms that are based on left-hand sides alone. This means that our algorithm is lazy (like Huet-Levy's) on strongly sequential parts of a program and "relaxes laziness minimally" to handle the other parts and thus does not sacrifice generality for the sake of efficiency.

1 Introduction

Equational logic programming is a style of declarative programming in which systems of oriented equations (called *rewrite rules*) of the form $l \to r$ (l and r are terms with variables in them universally quantified implicitly) constitute the set of logical assertions describing a program. An implementation of an equational language takes such a program and a term t as input and produces as output a "simpler" term s called the *normal form* of t (i.e., s contains no instances of the left-hand side (lhs) of any rule). Such an implementation is said to be *complete* if it finds the normal form s whenever $s = t$ follows logically from the input rules.

*Partially supported by NSF grant CCR-8805734. A preliminary version of this paper was presented at the 1990 Symposium on Logic in Computer Science.

†Current Address: Bellcore Rm 2A-274, 445 South Street, Morristown, NJ 07962 – 1910.

An equational program is evaluated by *term rewriting*, i.e., by repeatedly replacing *redexes* (instances of lhs) in the input term by the corresponding right-hand sides (rhs) until the input term is normalized. (A good survey of term rewriting appears in [DJ90].) An implementation based on term rewriting can be complete if the input programs are guaranteed to be *confluent* or Church-Rosser. For programming purposes, it is natural to do this by requiring input programs to be *orthogonal*[1] [OD77, HL79]. In such systems, the lhs of rules are linear (i.e., no variable in an lhs occurs more than once) and no two lhs unify except possibly at the root. If they do unify, then any term that is an instance of such overlapping rules must be rewritten into the same term by each of those rules.

For orthogonal programs, O'Donnell [OD77] showed that the parallel-outermost strategy, which repeatedly replaces all the outermost redexes, is complete. Many of the reductions performed by this strategy could be wasteful in general. A lazy normalization algorithm that completely eliminated these wasteful reductions by reducing only "needed redexes" was later developed by Huet and Levy [HL79]. However, this algorithm required the input programs to be restricted to the subclass of *strongly sequential systems*. This is because it is not always possible to eliminate wasteful reductions. For instance, in $or(R_1, R_2)$ (R_1, R_2 are redexes; see parallel-or example of fig. 1) only one of R_1 or R_2 may need to be reduced, but there is no way of apriori knowing which one. So we must reduce both redexes and thereby do wasteful reductions. Even in $B(R_1, R_2, R_3)$ (see Berry's example in fig. 1) where it is adequate to reduce only two of the three redexes, we once again cannot determine which two. Therefore it appears more appropriate to *minimize* rather than completely eliminate wasteful reductions. Furthermore, if this minimization *eliminates* wasteful reductions for strongly sequential systems then we can handle truly parallel constructs such as *or*, as well as sequential (but not strongly sequential) functions such as *merge* in fig. 1 without compromising the efficiency of lazy normalization on strongly sequential components of the program. We develop such a strategy in this paper.

1.1 Overview of our Approach and Related Results

Huet and Levy [HL79] developed a lazy normalization strategy (based on lhs alone) for strongly sequential systems based on the concept of *needed redexes*. Such redexes must be reduced in every reduction sequence that normalizes a term. We encounter two problems when using the concept of needed redexes to eliminate wasteful reductions for the entire class of orthogonal systems. Firstly, needed redexes no longer exist when overlaps are permitted, e.g., the term $or(or(T, T), or(T, F))$ contains no needed redex. Secondly, even when needed redexes exist, they cannot always be computed, e.g., two of the three redexes R_1, R_2, and R_3 are needed in $B(R_1, R_2, R_3)$, but it is undecidable which two. However, in both these cases, we can identify a *necessary set* such that at least one of the redexes in this set must always be reduced. For instance, $\{or(T, T), or(T, F)\}$ and $\{R_1, R_2\}$ respectively constitute necessary sets for the above two cases. Based on this concept of necessary sets, we develop a normalization algorithm that minimizes wasteful reductions for the entire class of orthogonal systems.

- In section 4 we formally develop the concept of a necessary set and establish *soundness* and *completeness* of a strategy that repeatedly replaces all the redexes in such a set.
- The problem of determining whether a set of redexes is necessary is undecidable in general.

[1] Even programs written in functional languages such as ML, HOPE and Miranda are typically orthogonal and those that are not can usually be so transformed (details appear in [Lav88, PS90, SRR92]).

$$
\begin{array}{lcl@{\qquad}lcl@{\qquad}rcl}
or(T,x) & \rightarrow & T & B(T,F,x) & \rightarrow & T & merge(x:xs, y:ys) & \rightarrow & x:y:merge(xs,ys) \\
or(x,T) & \rightarrow & T & B(F,x,T) & \rightarrow & T & merge(x,nil) & \rightarrow & x \\
or(F,F) & \rightarrow & F & B(x,T,F) & \rightarrow & T & merge(nil,y) & \rightarrow & y
\end{array}
$$

Figure 1: Parallel-or, Berry's example and Merge.

Therefore in section 5, we develop the notion of a *strongly necessary set* whose necessity can be decided by examining the lhs alone. We devise an algorithm $FindNS$ that computes a strongly necessary set in any term not in normal form. Any set computed by $FindNS$ is called a *cover*. $FindNS$ forms the basis of our complete normalization algorithm for orthogonal systems.

• In section 6, we formulate a notion of completeness that is appropriate for normalization algorithms that do not examine rhs. We show that any such algorithm that does not always reduce a superset of a cover is not complete w.r.t. this notion whereas our algorithm is complete. This means that our algorithm is lazy on strongly sequential parts of a program and "relaxes laziness minimally" in order to handle the other parts.

• Some implications of our results are discussed in section 7. In particular, we show that by appropriately choosing the necessary set to be reduced at each step, parallel-outermost and Huet-Levy strategies can be obtained from our strategy.

• Computing necessary sets by direct implementation of $FindNS$ is prohibitively expensive. Therefore in section 8 we describe how to compile the program into a matching automaton that is used by $FindNS$ to quickly identify a necessary set at runtime. We then give a tight characterization of equivalence of states in this automaton and use this to directly construct an *optimal* automaton.

• We can show that using the matching automaton, $FindNS$ can identify a necessary set in time that is linear in the size of the input term. More importantly, $FindNS$ leads to an efficient incremental algorithm for computing necessary sets. In particular, we can show that after performing a reduction, recomputation of a necessary set can be performed very quickly. Incrementality is crucial for obtaining a practical normalization algorithm based on necessary sets. (See section 8.1.)

• Equational programming languages have so far been based on strongly sequential systems and its subclasses. Although violations of strong sequentiality are not frequent, it is nevertheless a cumbersome restriction. This is all the more so because it is hard for a programmer to check if his program observes this discipline, and even harder for a compiler to report where a violation occurred. Therefore it is quite useful to relax the strong sequentiality restriction. Our method, which can handle non-sequential systems without compromising the efficiency on strongly sequential components, has shown that such relaxation is indeed possible.

To make our technical development intuitive and simple to follow, throughout this paper we will be dealing with orthogonal programs that follow the *constructor discipline*[2]. Note that this entails no loss of generality since equational programs almost exclusively follow this discipline [OD85] and those that do not can be syntactically transformed so as to follow it (details appear in [That85]).

Finally, we would like to clarify how our work in this paper differs from our previous work on parallel evaluation of *path sequential systems* presented in [SPR89, SR92]. As pointed out by one of the reviewers of this paper, it appears on the surface that these two works are related since they both

[2] In languages that follow this discipline (such as HOPE and ML), the outermost symbols in all lhs are distinct from those that appear inside any lhs.

address the problem of parallel evaluation strategies for orthogonal systems. However, we would like to point out that indeed the two works address entirely different problems and the solutions presented are also equally different. Whereas the subject matter of this paper relates to relaxing strong sequentiality requirement, the work in [SPR89, SR92] concerns imposing further restrictions on strongly sequential systems so as to enable efficient parallel searches for needed redexes. Thus these two works address classes of systems that are at two different extremes in terms of expressive power. Secondly, parallel strategies are employed in [SPR89, SR92] *only as an optimization* to reduce execution time; it has no bearing on completeness, which can still be obtained with a sequential algorithm. In contrast, parallel strategies are necessary to assure completeness for the systems considered in this paper. Furthermore, the parallel evaluation discussed in [SPR89, SR92] is lazy and reduces only needed redexes, whereas the strategy presented in this paper will sometimes perform wasteful reductions.

2 Notations and Definitions

This section contains terminology commonly used in term rewriting systems. The notion of a *path* (also called *occurrence* or *position*) is used to refer to subterms in a term as follows. A path is either the empty string Λ that reaches the root or pi (p is a path and i an integer) which reaches the ith argument of the root of the subterm reached by p. We use t/p to refer to the subterm of t reached by p and $t[p \leftarrow s]$ to denote the term obtained by replacing the subterm t/p by s. We say $p \leq q$ whenever $\exists r\ pr = q$, i.e., p reaches an ancestor of q. We say that two occurrences p and q are disjoint if neither $p \leq q$ nor $q \leq p$.

A *substitution* maps variables to terms. An *instance* $t\beta$ of a term t is obtained by substituting $\beta(x)$ for every variable x in t. If t is an instance of u then we say $u \leq t$ and call u a *prefix* of t.

A term rewriting system (TRS) R is a set of *rewrite rules* of the form $[l_i \rightarrow r_i]$ such that the variables in r_i are a subset of those in l_i. Let $\mathcal{F}$ be the set of outermost symbols of all lhs (called function symbols) and $\mathcal{C}$ be the set of non-variable symbols (called constructors) that appear inside left-hand sides. An orthogonal TRS is said to be a *constructor system* iff $\mathcal{F} \cap \mathcal{C} = \phi$. Henceforth, we will use the terms *equational program* and *orthogonal systems* interchangeably.

We use $root(t)$ to refer to the root symbol of a term t. A term u is a *redex* if $u \geq l$ for some lhs l. A term t is in *normal form* (NF) if it contains no redexes. If the subterm t/p is a redex then we call p a *redex occurrence* in t.

Let $\mathcal{U} = \{p_1, p_2, \ldots, p_n\}$ be a set of mutually disjoint redex occurrences in term t. If $t/p_i = l\beta$ then let $t'_i = r\beta$, for $1 \leq i \leq n$. We say that t reduces to s if $s = t[p_1 \leftarrow t'_1, ..., p_n \leftarrow t'_n]$ and denote this $A : t \Rightarrow s$, where $A = \langle t, \mathcal{U} \rangle$ is called an *elementary derivation*. Although A is associated with t, by a slight abuse of notation we say $A : t' \Rightarrow s'$ if $\mathcal{U}$ is also a set of redex occurrences for t' and s' is obtained by reducing these redexes.

A derivation is a (possibly empty) sequence of elementary derivations $A_1; A_2; \cdots; A_n$. The notation $A : t \Rightarrow^* s$ is used when A is nonelementary. We use $A; B$ for the concatenation of derivations A and B, ϕ for the empty derivation, $A[i, j]$ for the subsequence $A_i; A_{i+1}; \ldots; A_j$ in A. For an elementary derivation C we use $|C|$ to denote the number of redexes rewritten by it. We use $A : t \rightarrow s$ if $|A| = 1$ and $B : t' \rightarrow^* s'$ if $|B_i| = 1$ for every elementary derivation in B.

$$
\begin{aligned}
f(a(e), c(x)) &\rightarrow x \\
f(a(x), d(y)) &\rightarrow y
\end{aligned}
\qquad\qquad
\begin{aligned}
t &= f(a(t_1), t_2) \\
t_1 &= f(a(e), c(e)) \\
t_2 &= f(a(e), d(d(e))).
\end{aligned}
$$

Figure 2: Needed redexes in term t.

3 Evaluation Strategies for Orthogonal Systems

In this section we present a brief overview of previously known results related to evaluation strategies for orthogonal programs. We remark that most of these results are not necessary for the development of our paper; they are presented so that the reader can get familiar with the concept of sequentiality and understand how our results relate to previous ones.

Huet and Levy, in their seminal paper on call-by-need computations in unambiguous orthogonal systems [HL79], developed the notion of *needed redexes.* (In unambiguous orthogonal systems, no two lhs can unify.) A needed redex in a term t is a redex that is reduced in every reduction sequence that takes t to its normal form. For instance, consider the example shown in fig. 2 and fig. 3. (The redex reduced in each reduction is indicated above the corresponding arrow in the fig. 3.) Observe that the redex t_2 is reduced in all normalizing sequences and is hence needed. On the other hand, t_1 is reduced only in two of the three reductions and so is not needed. The following significant result, due to Huet and Levy, lays the formal foundation for complete normalization strategies for unambiguous orthogonal systems. As pointed out earlier with the parallel-or example, this theorem <u>does not</u> apply to ambiguous orthogonal systems.

Theorem 1 (Huet-Levy) *In unambiguous orthogonal systems, if a term contains redexes, then it contains a needed redex. If it possesses a normal form then the normal form can be obtained by repeatedly replacing needed redexes.*

Observe that for orthogonal systems, at least one of the outermost redexes is a needed redex. To see this, note that all of these redexes will continue to remain in the term irrespective of other reductions that are performed inside and so normal form cannot be reached without reducing any of them. Unfortunately, we cannot in general decide which of these redexes is needed. This is because the definition of a needed redex requires that all normalizing reduction sequences (of which there could be infinitely many) be examined in order to pick a needed redex. In order to understand

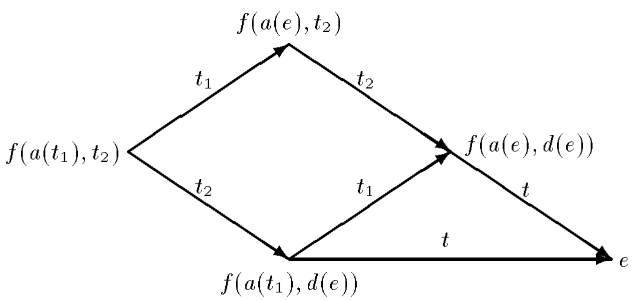

Figure 3: Reduction sequences that normalize t.

the need to look ahead at all normalizing reduction sequences, consider Berry's example shown in figure 1. Observe that in order to normalize the term $B(R_1, R_2, R_3)$, two of the three redexes R_1, R_2 and R_3 must be reduced. However, we cannot determine which two need to be reduced without knowing what each of R_1, R_2 and R_3 reduce to. For instance, R_1 and R_2 are needed redexes iff the two terms can be reduced to T and F respectively. Since it requires unbounded look-ahead (i.e., performing an unbounded number of reductions) to determine whether R_1 and R_2 will reduce to T and F respectively, we cannot decide which two of R_1, R_2 and R_3 are needed redexes.

3.1 Sequentiality

Examining all possible reduction sequences is clearly not a reasonable strategy for identifying needed redexes. Therefore we must eliminate look-aheads in selecting a needed redex. To do this, we can impose the following condition on equational programs: Let $C[t_1, ..., t_n]$ denote a term obtained from a prefix C by substituting the n variables in it by $t_1, ..., t_n$. Then, on inspecting any prefix C of a term $t = C[t_1, ..., t_n]$, we must be able to determine an i such that we need to "work inside" (in our case, search for needed redex inside) t_i. Note that this choice of i is independent of the terms $t_1, ..., t_n$ and so there is no need for any look-ahead to determine what these subterms reduce to.

The condition given above is known as the *sequentiality restriction* because it appears necessary for having a complete *sequential* normalizing strategy, i.e., a strategy that picks *only one* t_i to work on after inspecting the prefix C. In contrast, a parallel strategy may choose to work simultaneously on several of the terms $t_1, ..., t_n$. Intuitively, such a parallel strategy seems necessary for dealing with Berry's example: since we cannot determine which one of the subterms R_1, R_2 and R_3 to "work on", we can simultaneously work on all of them. Then we will be guaranteed to find the normal form of $B(R_1, R_2, R_3)$ even if one of R_1, R_2 and R_3 does not have a normal form. On the other hand, a sequential strategy that chooses to work inside only one of the subterms, say R_2, is not complete. This is because R_2 may not possess a normal form even when $B(R_1, R_2, R_3)$ possesses a normal form. For instance, if R_1 and R_3 reduce to F and T respectively, then $B(R_1, R_2, R_3)$ reduces to $B(F, R_2, T)$, which can then be reduced to the normal form T.[3]

The *sequentiality* condition given above falls short of what is required since it is undecidable whether an equational program satisfies this condition. For instance, Berry's example would be sequential if $B(R_1, R_2, R_3)$ can always be reduced to R_1. In that case, any work performed inside R_1 will always be useful and so a sequential strategy can always choose to work inside R_1. However, detecting whether $B(R_1, R_2, R_3)$ always reduces to R_1 is clearly undecidable. The problem here again is that we have to answer the question whether a term t always reduces to another term t'. To avoid dealing with this question of reducibility, Huet and Levy imposed a stronger condition called *strong sequentiality* that is independent of the rhs. This made the notion of strong sequentiality decidable and yielded an algorithm for computing needed redexes.

[3]The above definition of a sequential strategy is called sequential* by Klop and Middeldorp [KM89] and closely resembles Huet and Levy's definition of sequentiality based on sequential predicates of Kahn and Plotkin [KP78]. Kennaway [Ken89] has shown that even those programs that are not sequential in this sense, such as Berry's example and parallel-or, can still be normalized using a sequential algorithm that selects only one redex for reduction in any term. Although this result is quite surprising and theoretically very interesting, it does not lead to a useful normalizing algorithm in practice and so we do not discuss it further.

3.2 Strong Sequentiality

In this section we introduce the concept of strong sequentiality. To do this, we need to formalize the idea of "being independent of rhs" using the concept of *arbitrary reductions* that permit us to replace a redex by any arbitrary term[4]. For instance, the redex R_1 in the term $B(R_1, R_2, R_3)$ can be reduced to T or F or any other term using arbitrary reductions. Now we say that a redex r in a term t is *strongly needed* if every *arbitrary reduction sequence* that normalizes t reduces r. Note that every strongly needed redex is a needed redex, but the converse is not true.

Based on the notion of strongly needed redexes, Huet and Levy defined the class of *strongly sequential programs* for which we can efficiently compute a strongly needed redex in any term that is not in normal form. A formal definition of such systems is given below. (This definition is adapted from the definition of strongly sequential* systems in [KM89], wherein it is also established that the notions of strong sequentiality and strong sequentiality* are equivalent. The definition of strong sequentiality given in [HL79, KM89] uses Kahn and Plotkin's [KP78] definition of a sequential predicate and is more complex.) For a more detailed development of the concept of sequentiality, the reader is referred to [HL79, KM89, That84, That87].

Definition 2 (Strong Sequentiality I) *An equational program is strongly sequential iff the following condition is satisfied by every prefix C that contains no redexes: there is an i such that r_i is a strongly needed redex for $C[r_1, ..., r_n]$ irrespective of the redexes $r_1, ..., r_n$.*

For strongly sequential programs, in any term not in normal form, there is at least one strongly needed redex among the outermost redexes whose neededness depends only on portions of the term C above these redexes. Therefore an algorithm for computing needed redex can be developed if we tabulate the strongly needed redex for every possible prefix. However, this is impossible since there are infinitely many such prefixes and so the above definition does not lead to an algorithm for computing needed redexes. What is needed is a method that directs us successively to a needed redex on the basis of a "small portion" of the term and not on the basis of everything above the outermost redexes. For constructor systems, an alternative definition of strong sequentiality was given in [HL79] that is based only on the prefixes of lhs, of which there are only finitely many. In order to present this definition we need the following notion of an *index*. (The notion of index is more complex for systems that do not follow the constructor discipline.)

Definition 3 (Index) *Let u be a prefix of an lhs and $\mathcal{L}_u$ denote the set of all lhs that unify with u. Then an occurrence p in u is said to be an index if*

1. *u/p is a variable, and*

2. *$\forall l \in \mathcal{L}_u \; l/p$ is a nonvariable term.*

Using the notion of index, we can now define strong sequentiality as follows:

Definition 4 (Strong Sequentiality II) *A system of lhs $\mathcal{L}$ is said to be strongly sequential iff $\forall l \in \mathcal{L}\; [u < l \Rightarrow u$ possesses an index$]$.*

[4]Thatte [That87] argues that arbitrary reductions do not satisfactorily capture the idea of being independent of rhs since an arbitrary reduction sequence can replace different occurrences of the same redex with different terms. Instead, he introduces the concept of *possible reductions* and develops a notion of sequentiality known as *left sequentiality*. However, the concept of possible reductions is much more complex and significantly complicates proofs. Furthermore, for programs that follow the constructor discipline, the notions of left sequentiality and strong sequentiality coincide for every set of lhs [That87]. For these reasons, we prefer the concept of arbitrary reductions over possible reductions.

Based on this definition, Huet and Levy presented an algorithm to construct a matching automaton, also known as matching dag. This automaton guides the search for needed redexes. In particular, each state of the automaton remembers the prefix of the input term inspected in reaching that state. It also specifies the index for the prefix. Once an index is identified, the search for needed redex is restarted at this index position. This search is again guided by the same automaton.

Since the work of Huet and Levy, there have been many theoretical developments regarding the notion of sequentiality and practical developments in equational programming. A good survey of results related to sequentiality appears in [KM89]. Practical developments include the equational interpreter developed by Hoffman and O'Donnell [HO84] and later enhanced by Strandh [Str89], and the parallel equational programming system developed by Owen et al [OPRRS92]. Several works such as [Lav88, PS90, SRR92] have also extended Huet-Levy's work to deal with prioritized systems of equations. In the following section, we develop the technical results of this paper.

4 Complete Normalization Strategy

We now formalize the concept of necessary sets and describe a complete normalization strategy based on necessary sets. First we need the following notion of *residuals.*

Definition 5 *[HL79] Let u and v be redex occurrences in t and A be the elementary derivation that rewrites u using the rule $l \to r$. The set $v \setminus A$ of residuals of v by A are*

$$v \setminus A = \begin{cases} \phi, \textit{ if } u = v \\ \{v\}, \textit{ if } v \not\geq u \\ \{up'q \mid r/p' = x\}, \textit{ if } v = upq \textit{ and } l/p = x \\ \quad \textit{where } x \textit{ is a variable} \end{cases}$$

Intuitively, if t is rewritten to s then a subterm s' of s that is a copy of a subterm t' of t is called the residual of t'. For instance, using the rule $f(x, y) \to g(f(x, x))$ on the term $f(f(0, 0), 1)$ the residual of $f(0, 0)$ are the subterms $f(0, 0)$ in $g(f(f(0, 0), f(0, 0)))$. Note that residuals of disjoint redexes remain disjoint.

We now extend this notion of residuals to a set $\mathcal{U}$ of redexes as $\mathcal{U} \setminus A = \bigcup_{u \in \mathcal{U}} u \setminus A$. When A is a sequence $A_1; \cdots; A_n$ of elementary derivations then $\mathcal{U} \setminus A = \mathcal{U} \setminus A_1 \setminus A_2 \setminus \cdots \setminus A_n$. (Note $\setminus$ associates to the left.) Using the concept of residuals we now define a necessary set.

Definition 6 *A set of redexes in a term t is a necessary set if in every reduction sequence that takes t to its normal form, at least one redex in this set or its residual is rewritten.*

A simple normalization strategy based on necessary sets is:

Algorithm *Normalize*(t)
 While $\exists$ a nonempty necessary set $\mathcal{R}$ for t do
  Let t' be obtained from t by rewriting all redexes in $\mathcal{R}$;
  $t := t'$;
 end

This algorithm assumes that by repeatedly reducing necessary sets of redexes we can reach the normal form. The correctness of this algorithm is established by showing that it is *sound* (i.e., if it terminates with t_n then t_n is the NF of the input term) and *complete* (i.e., it terminates on all terms that possess a normal form). Soundness is immediate since the set of all redexes in a term constitutes a necessary set and so if $\mathcal{R}$ is empty then t is in NF. Intuitively, it is apparent that *Normalize* is complete since it always performs some "useful reductions" whenever it reduces all the redexes in a necessary set. Therefore after every step of *Normalize* the term should be in some sense "closer" to its NF than before. We formalize this as follows. Let $A : t \Rightarrow^* t_n$ (where t_n is the NF of t) and $B : t \Rightarrow t'$ by reducing a necessary set of t. Then there exists an $A' : t' \Rightarrow^* t_n$ such that A' is less than A in the well founded ordering defined as:

Definition 7 *For two derivations $A = A_1; \cdots; A_n$ and $B = B_1; \cdots; B_m$*

1. *If $n > m$ then $A \succ B$.*
2. *If $n = m$ then $A \succ B$ if $\exists i$ such that $\forall j > i\ |A_j| = |B_j|$ and $|A_i| > |B_i|$.*
3. *If $n = m$ and $|A_i| = |B_i|$ for $1 \leq i \leq n$ then $A \simeq B$. If $A \simeq B$ or $A \succ B$ we say $A \succeq B$.*

Note that this is just the lexicographic ordering on $\langle |A_n|, |A_{n-1}|, ..., |A_1| \rangle$. Now we are ready to establish the completeness of *Normalize*. Observe that a necessary set contains at least one redex that must be reduced in order to reach NF, but other redexes could be "useless". We first prove that there is no harm in reducing such redexes, i.e., if $B : t \Rightarrow t'$ by reducing only "useless" redexes then t' is "no farther" from its NF than t. Formalizing this,

Lemma 8 *Let $B = \langle t_0, \mathcal{B} \rangle : t_0 \Rightarrow t'$ and $A = A_1; \cdots; A_n : t_0 \Rightarrow^* t_n$. If no residual of any redex in $\mathcal{B}$ occurs in t_n or is reduced in any of $A_1, A_2, \ldots, A_n$ then $\exists C : t' \Rightarrow^* t_n$ such that $A \succeq C$.*

Proof: We first construct a sequence $A' : t_0 \Rightarrow^* t_n$ that reduces only those redexes that are above or disjoint from any residual of a redex in $\mathcal{B}$. In order to do this, we define a function $f_i(t)$ which replaces each subterm in t that is a residual of a redex r in $\mathcal{B}$ by a new constant c_r. Formally, $f_i(t) = t[(r \setminus A[1, i-1]) \leftarrow c_r]$, for each redex r in $\mathcal{B}$. Now split the reductions in A_i into A'_i and A''_i such that A'_i reduces only those that are disjoint to or above the residuals of the redexes in $\mathcal{B}$ and A''_i constitutes the rest of the reductions in A_i. Let $s_i = f_i(t_i)$ for $0 \leq i \leq n$. Now we show that $\forall i\ A'_i : s_{i-1} \Rightarrow s_i$. Let $A'_i : t_{i-1} \Rightarrow t'_i$ and $A''_i : t'_i \Rightarrow t_i$. Since A'_i reduces redexes that are above the residuals of $\mathcal{B}$ in t_{i-1}, it also reduces redexes above the new constants in s_{i-1}. Therefore $f_i(t'_i) = s_i$. Since A''_i reduces only redexes which are below a residual of $\mathcal{B}$ in t'_i, the position of every residual of any redex in $\mathcal{B}$ is unchanged in t'_i and t_i and hence $f_i(t'_i) = f_i(t_i) = s_i$. Therefore $A'_i : s_{i-1} \Rightarrow s_i$.

Now $f_n(t_n) = s_n = t_n$ since t_n contains no residual of $\mathcal{B}$ and therefore $A' = A'_1; \cdots; A'_n : s_0 \Rightarrow^* t_n$. Note that A' reduces s_0 to t_n even if arbitrary terms are substituted in the place of the new constants in s_0. Therefore t', obtained by substituting the new constants with replacements for redexes in $\mathcal{B}$ is also reduced to t_n by A'. Thus $C = A' : t' \Rightarrow^* t_n$ and $A \succeq C$ since $|A_i| \geq |A'_i|$ for $1 \leq i \leq n$. ∎

Using this we now show that if we perform "some useful" reductions on t then we will be closer to its NF. To formalize and prove this, we need the following notion of a residual of one derivation by another[HL79]. The residual of an elementary derivation $B = \langle t, \mathcal{U} \rangle$ by A, denoted $B \setminus A$, is the elementary derivation reducing the set $\mathcal{U} \setminus A$. Now $A \setminus B$ and $B \setminus A$ are related as follows:

Lemma 9 (Parallel Moves Lemma [HL79]) *(Fig. 4) Let $A : t \Rightarrow t_1$ and $B : t \Rightarrow t_2$ be two elementary derivations. Then $A; (B \setminus A) = B; (A \setminus B)$*

$$
\begin{tikzcd}
 & t_1 \arrow[dr, "B \setminus A"] & \\
t \arrow[ur, "A"] \arrow[dr, "B"'] & & t' \\
 & t_2 \arrow[ur, "A \setminus B"'] &
\end{tikzcd}
$$

Figure 4: Parallel Moves Lemma

The parallel moves lemma allows us to generalize the notion of residuals to non-elementary derivations. In such a case, $B \setminus A$ is defined inductively as:

- $\phi \setminus A = \phi$
- $(B_1; B') \setminus A = (B_1 \setminus A); (B' \setminus (A \setminus B_1))$, where B_1 is elementary.

Huet and Levy [HL79] also showed that parallel moves lemma can be generalized to non-elementary derivations. Based on generalized parallel moves lemma, we will now formalize and prove that by performing "some useful" reductions, we go closer to NF.

Theorem 10 *Let $A : t \Rightarrow^* t_n$ be any sequence that normalizes t and $B = \langle t, \mathcal{B} \rangle : t \Rightarrow t'$. If A reduces at least one residual of a redex in $\mathcal{B}$ then $\exists A' : t' \Rightarrow^* t_n$ such that $A \succ A'$.*

$$
\begin{tikzcd}
 & t_m \arrow[dr, "B \setminus A{[1,m]} = \phi"] & \\
t \arrow[ur, "A{[1,m]}"] \arrow[dr, "B"'] & & t'_m \\
 & t' \arrow[ur, "A{[1,m]} \setminus B"'] &
\end{tikzcd}
$$

Figure 5: Proof of theorem 10: Step 1

Proof: Let A_l be the last elementary reduction step in A in which a residual of a redex in $\mathcal{B}$ is reduced and t_m be the first term in which no residual of any redex in $\mathcal{B}$ appears. We construct A' in three steps.

<u>Step 1</u>: Let $A'[1, m] : t' \Rightarrow^* t'_m$ and $A'[m+1, n] : t'_m \Rightarrow^* t_n$, where t'_m is such that $A[1, m] \setminus B : t' \Rightarrow^* t'_m$. Now observe from fig. 5 that $t_m = t'_m$ and so $A'[m+1, n] = A[m+1, n]$.

<u>Step 2</u>: $A'[1, m]$ is constructed in two steps (see fig. 6). We divide A_l into A_l^1 and A_l^2 such that A_l^1 consists solely of residuals of redexes in $\mathcal{B}$ and $A_l^2 = A_l - A_l^1$. (Note $A_l^1 \neq \phi$.)

Since no residual of any redex in $\mathcal{B}$ (and hence of $\mathcal{B} \setminus (A[1, l-1]; A_l^1)$) is reduced in sequence $A_l^2; A[l+1, m]$, by preceding lemma, there exists a sequence $A_l'^2; A'[l+1, m] : t_l'^1 \Rightarrow^* t'_m$ such that $A_l^2; A[l+1, m] \succeq A_l'^2; A'[l+1, m]$. Also note $|A_l'^2| \leq |A_l^2|$.

<u>Step 3</u>: Consider the sequence $A'[1, l-1]; A_l'^1 = (A[1, l-1]; A_l^1) \setminus B$. Note $A_l'^1 = \phi$ (since A_l^1 contains only residuals of redexes in $\mathcal{B}$, none of which are in t') and hence it can be dropped and let $A'_l = A_l'^2$.

Now $A \succ A'$ since

$$\begin{tikzcd}
t \arrow[r, "{A[1,l-1];\, A_l^1}"] \arrow[d, "B"] & t_l^1 \arrow[r, "{A_l^2;\, A[l+1,m]}"] \arrow[d, "{B\backslash(A[1,l-1];\, A_l^1)}"] & t_m \arrow[d, "\phi"] \\
t' \arrow[r, "{A'[1,l];\, A_l'^1}"'] & t_l'^1 \arrow[r, "{A_l'^2;\, A'[l+1,m]}"'] & t'_m = t_m
\end{tikzcd}$$

Figure 6: Proof of theorem 10: Steps 2 and 3

$A[m+1,n] \simeq A'[m+1,n]$
$A[l+1,m] \succeq A'[l+1,m]$ and
$|A'_l| = |A_l'^2| \leq |A_l^2| < |A_l^1| + |A_l^2| = |A_l|$. ∎

Corollary 11 *$Normalize$ is complete.*

Proof: If a term t has a normal form s, then there exists a sequence A that normalizes it. Let $B_1; B_2; \cdots$ be the sequence of reductions performed by $Normalize$ such that $B_1 : t \Rightarrow t_1, B_2 : t_1 \Rightarrow t_2$ etc. Since each B_i reduces a necessary set, it satisfies the antecedent of the above theorem and hence there exist $A^1, A^2, \ldots$ such that A^i normalizes t_i and $A \succ A^1 \succ A^2 \succ \cdots$. Since the ordering $\succ$ is well-founded, there is no infinite descending chain $A \succ A^1 \succ A^2 \cdots$ and so $Normalize$ must terminate. ∎

5 Computing Necessary Sets

Note that for non-overlapping orthogonal systems, the singleton set consisting of a needed redex is a necessary set, by theorem 1. Since it is undecidable whether a redex is needed, it immediately follows that it is undecidable whether an arbitrary set of redexes is necessary. However, if we restrict our reasoning to be based on lhs alone then we can have a "partial" decision procedure that terminates with the answer "yes" or "don't know" on every input set. If we ensure that this procedure answers "yes" on at least one set of redexes in any term not in NF then it can be used for normalizing a term by repeatedly reducing such a set. We refer to those sets for which the answer is "yes" as *strongly necessary sets*, i.e., the necessity of such a set can be established using lhs alone. Now we develop an algorithm to compute a strongly necessary set. First we need the following definitions.

A *constructor prefix* of a term t is any prefix u all of whose nonvariable symbols, with the possible exception of its root, are constructors. A *maximal constructor prefix* of t, denoted $MCP(t)$, is a constructor prefix u such that there is no other constructor prefix $v > u$. A constructor prefix u is known as a *partial redex* if there exists a redex $v \geq u$ (i.e., if u unifies with an lhs). For instance, in

Example 12 $R = \{f(b(c), x) \rightarrow a, f(x, a) \rightarrow b(c)\}$ *and* $s = f(b(c), g(d))$.

the terms $f(x,y)$, $f(b(x),y)$ and $f(b(c),y)$ are all constructor prefixes (as well partial redexes) of s.

Now we are ready to present our algorithm $FindNS$ shown below. The algorithm takes a term t as a parameter and computes a necessary set of t. We will later show that it returns a nonempty necessary set for t whenever t is not in normal form. $FindNS$ can identify many different necessary sets for t. In order to capture this, we present $FindNS$ as a non-deterministic algorithm that returns one such set. A deterministic algorithm can be readily obtained from $FindNS$ by arbitrarily selecting any one of the non-deterministic choices. In the algorithm description below, we use the notation $p \cdot \mathcal{D}$ to denote the set obtained by prefixing the path p to every occurrence in $\mathcal{D}$. More precisely, if $\mathcal{D} = \{q_1, ..., q_m\}$ then $p \cdot \mathcal{D}$ denotes the set $\{pq_1, ..., pq_m\}$.

Algorithm $FindNS(t)$

Case 1. t is a redex: return($\{\Lambda\}$)

Case 2. t is not a redex: Let $u = MCP(t)$,

$\mathcal{D} = \{p \mid u/p$ is a variable $\}$ and

$\mathcal{D}_{ne} = \{p \in \mathcal{D} \mid FindNS(t/p)$ is nonempty $\}$.

Case 2a. $\mathcal{D}_{ne} = \phi$: return(ϕ). <u>**Otherwise**</u>

Case 2b. u is not a partial redex: return($p \cdot FindNS(t/p)$), for any $p \in \mathcal{D}_{ne}$.

Case 2c. u is a partial redex: Choose any $\mathcal{D}' \subseteq \mathcal{D}$ such that $\bigcup_{p \in \mathcal{D}'} \mathcal{S}_p = \mathcal{L}_u$.

Case 2c.1. $\mathcal{D}' \subseteq \mathcal{D} - \mathcal{D}_{ne}$: /* i.e., $FindNS(t/p)$ is empty $\forall p \in \mathcal{D}'$ */

return($p \cdot FindNS(t/p)$), for any $p \in \mathcal{D}_{ne}$.

Case 2c.2. Otherwise: return($\bigcup_{p \in \mathcal{D}'} p \cdot FindNS(t/p)$)

end

We now give an intuitive outline of our algorithm. Suppose the term t being normalized is itself a redex. Then the singleton set containing this redex constitutes a necessary set of the term since it must be reduced[5] in order to normalize t. Therefore this redex occurrence $\{\Lambda\}$ is returned in case 1 of the algorithm. Otherwise consider $u = MCP(t)$. (We will now explain cases 2b and 2c of the algorithm and consider case 2a later.) If u is not a partial redex then no redex can ever be uncovered at the root of t by reducing redexes inside it. In this case, NF of t can be found only by normalizing each subterm below u. Thus we can choose a necessary set of any of these subterms as a necessary set for t. This explains case 2b. If u is a partial redex then we must reduce the redexes inside some subterms of t as there is a likelihood of "extending" u into an instance of some lhs. The lhs for which this is possible constitute the *match set* $\mathcal{L}_u$ of u, i.e., $\mathcal{L}_u$ is the set of all lhs that unify with u. Let $p_1, ..., p_r$ be some occurrences such that u/p_i is a variable for $1 \le i \le r$. Furthermore, assume that every lhs in $\mathcal{L}_u$ has a nonvariable in one of these positions, i.e., if $\mathcal{S}_p$ denotes the set $\{l \in \mathcal{L}_u \mid l/p$ is a nonvariable term$\}$ then $\bigcup_{i=1}^{r} \mathcal{S}_{p_i} = \mathcal{L}_u$. Then it is easy to see that the redexes inside one of the subterms t/p_i must be reduced before u can be extended and therefore we choose the union of the necessary sets of these subterms as a necessary set for t. This explains case 2c.2. However, if $FindNS$ returns an empty set on each of these subterms then, as mentioned earlier, each of t/p_i must be in normal form. So u can never be extended into a redex and hence this case is similar to case 2b. Therefore, in case 2c.1, we once again select any p such that $FindNS(t/p)$ is nonempty. Finally, when $FindNS(t/p)$ is empty for every variable occurrence p in u, t must be in normal form and hence case 2a.

We refer to any set returned by $FindNS(t)$ as a *cover* for t. As an illustration of $FindNS$, consider Berry's example (fig. 1) and the term $B(R_1, R_2, R_3)$ (R_1, R_2, R_3 are redexes). For this term, $FindNS$ computes $u = B(x, y, z)$ and $\mathcal{D} = \mathcal{D}_{ne} = \{1, 2, 3\}$ (case 2). Also note that $\mathcal{L}_u = \{l_1, l_2, l_3\}$, $\mathcal{S}_1 = \{l_1, l_2\}, \mathcal{S}_2 = \{l_1, l_3\}$ and $\mathcal{S}_3 = \{l_2, l_3\}$. Since u is a partial redex, $FindNS$ chooses (case 2c)

[5] This is because in an orthogonal system, an outer redex can never "disappear" by performing reductions inside.

a $\mathcal{D}'$, which can be any one of $\{1,2\},\{2,3\},\{1,3\}$ and $\{1,2,3\}$. All the recursive invocations of $FindNS$ made in case 2c.2 return from step 1 since R_1, R_2 and R_3 are redexes. Thus $FindNS$ may return any of $\{R_1,R_2\}$, $\{R_2,R_3\},\{R_1,R_3\}$ and $\{R_1,R_2,R_3\}$. Similarly, it can be verified that for $or(R_1,R_2)$, $FindNS$ returns $\{R_1,R_2\}$.

It may appear odd that the applicability of different cases are determined by first invoking $FindNS$ on the subterms and checking if they are empty. This is done only to simplify the description of the algorithm. In a real implementation, we need not explicitly compute $\mathcal{D}_{ne}$ or check if it is empty. Instead, in case 2b we iterate over the p's until $FindNS(t/p)$ returns a nonempty set. In case 2c we pick any $\mathcal{D}'$ and if $\bigcup_{p\in\mathcal{D}'} FindNS(t/p)=\phi$ then we can once again iterate over p's as in case 2b to pick a $p\in\mathcal{D}_{ne}$. We would also like to point out that in a real implementation, $FindNS$ will be made deterministic by making a fixed choice in cases 2b, 2c.1 and 2c.2. This choice may be either arbitrary or be based on a heuristic such as choosing a $\mathcal{D}'$ of smallest size.

Henceforth we will use $FindNS$ for computing the necessary set $\mathcal{R}$ in algorithm $Normalize$. Therefore for proving soundness of this modified $Normalize$ we need to show

Theorem 13 *If $FindNS(t)=\phi$ then t is in NF.*

Proof: By induction on height of t. For the basis, height of t is zero whence it is a redex or is in NF. In the former case the antecedent of the theorem does not hold (since $FindNS(t)=\{\Lambda\}$) and so the theorem holds vacuously. In the latter case also it holds since t is already in NF. For the induction step, we need to consider the following two cases.

Case 1. t is a redex: Antecedent of theorem does not hold.

Case 2. t is not a redex: Observe that $FindNS$ returns ϕ only in case 2a. This happens when $FindNS(s)$ returns ϕ on every subterm s below MCP of t. Since the heights of these subterms are less than that of t, by inductive hypothesis, each of these subterms are in NF and so t is in NF. ∎

For proving completeness of $Normalize$ we need to show that $FindNS$ indeed computes necessary sets. We make use of the following facts to do this.

Fact 1 Let $\mathcal{R}$ be any set returned by $FindNS(t)$. If no redex in $\mathcal{R}$ is rewritten in $t\to t'$ then $root(t')=root(t)$.

Proof: If $root(t')\neq root(t)$ then the redex reduced must be at the root of t. But this is not possible since such a redex will be in $\mathcal{R}$. ∎

Fact 2 Let $u=MCP(t)$ be a partial redex (but not a redex) and $p_1,...,p_r$ be such that u/p_i $(1\leq i\leq r)$ is a variable and $\bigcup_{i=1}^{r}\mathcal{S}_{p_i}=\mathcal{L}_u$. If $t\to t'$ and $root(t'/p_i)\in\mathcal{F}$ for $1\leq i\leq r$ then $MCP(t')$ is not a redex.

Proof: Follows from the facts that $u'=MCP(t')\geq u$ and every lhs which could potentially match any term $t\geq u$ has a constructor symbol at one of $p_1,...,p_r$. ∎

Although all the redexes in a set returned by $FindNS$ will remain in the term until at least one of them is reduced, a proof of this fact is quite cumbersome. However, observe that it suffices to show that a *nonempty subset* (rather than *all*) of these redexes will remain in the term for proving the necessity of such a set. By proving this (weaker) assertion, we show that

Theorem 14 *$FindNS(t)$ is a necessary set for t.*

Proof: We first establish by induction on height of t the following.

Let $\mathcal{R}(\neq \phi)$ be any set returned by $FindNS(t)$ and $t \rightarrow t'$ without reducing any redex in $\mathcal{R}$. Then one of the sets that can possibly be returned by $FindNS(t')$ is a nonempty subset of $\mathcal{R}$.

For the basis, height of t is zero and so it is either a redex or is already in NF. In the former case, there can be no reduction $t \rightarrow t'$ without reducing the redex t which is in $\mathcal{R}$. In the latter case there is no such $\mathcal{R}$ and thus the proposition holds vacuously. The induction step is established by analyzing each case of $FindNS$.
Case 1. t is a redex: Clearly the redex at the root of t is unchanged by a reduction inside t. Thus $FindNS(t') = \{\Lambda\}$ is a nonempty subset of $\mathcal{R} = \{\Lambda\}$.

Case 2a. $\mathcal{R} = \phi$: the antecedent of the proposition does not hold.

In the following cases, there is no redex at the root and hence $u' \geq u$ (where $u' = MCP(t')$ and $u = MCP(t)$). Also let $\mathcal{R}_p$ be any set returned by $FindNS(t/p)$.

Case 2b. u is not a partial redex: In this case $\mathcal{R} = \mathcal{R}_p$ for some $p \in \mathcal{D}_{ne}$. Since no redex in $\mathcal{R}$ and hence in $\mathcal{R}_p$ is rewritten, by fact 1 $root(t'/p)$ is a function symbol and so u'/p is a variable. Therefore, noting that $u' \geq u$ is not a partial redex, any set returned by $FindNS(t'/p)$ can be chosen as a necessary set for t'. Since the height of t/p is less than that of t, we can assume inductively that one of the sets returned by $FindNS(t'/p)$ is a nonempty subset of $\mathcal{R}_p$ and so a set returned by $FindNS(t')$ is a nonempty subset of $\mathcal{R}$.

Case 2c.1 u is a partial redex and $\mathcal{D}' \subseteq \mathcal{D} - \mathcal{D}_{ne}$: The redex reduced cannot be within t/q for $q \in \mathcal{D}'$ since $FindNS(t/q) = \phi$ and so (by theorem 13) each such subterm is in NF. Therefore, by fact 2, u' is not a redex. There are two cases to consider.

(i) u' is a partial redex: In this case since $u' \geq u$ we have $\mathcal{L}_{u'} \subseteq \mathcal{L}_u$ and hence $\bigcup_{p \in \mathcal{D}'} \mathcal{S}_p = \mathcal{L}_{u'}$. Thus the same $\mathcal{D}'$ can be chosen for u' and so $\mathcal{D}' \subseteq \mathcal{D} - \mathcal{D}_{ne}$ for t' also. The rest of the proof carries through as in case 2b.

(ii) u' is not a partial redex: the proof carries through exactly like case 2b.

Case 2c.2 u is a partial redex and $\mathcal{D}' \not\subseteq \mathcal{D} - \mathcal{D}_{ne}$: Here $\mathcal{R} = \bigcup_{p \in \mathcal{D}'} \mathcal{R}_p$. Again there are two cases to consider.

(i) u' is not a partial redex: Note that there exists a $p \in \mathcal{D}'$ such that $p \in \mathcal{D}_{ne}$. Since no redex in $\mathcal{R}$ and hence in $\mathcal{R}_p$ is reduced, (by fact 1) t'/p is a function symbol and so u'/p is a variable. Therefore $FindNS(t')$ can return $FindNS(t'/p)$, which can be a nonempty subset of $\mathcal{R}_p$ by induction hypothesis. Thus a set returned by $FindNS(t')$ is a nonempty subset of $\mathcal{R}$.

(ii) u' is a partial redex: Note that no redex in any $\mathcal{R}_p$ (for $p \in \mathcal{D}'$) is reduced and so (by fact 1) $root(t'/p) \in \mathcal{F}$. Therefore, u' is not a redex (by fact 2) and u'/p (for $p \in \mathcal{D}'$) is a variable. Also, since $u' \geq u$ we have $\mathcal{L}_{u'} \subseteq \mathcal{L}_u$ and so $\bigcup_{p \in \mathcal{D}'} \mathcal{S}_p = \mathcal{L}_{u'}$. By inductive hypothesis $FindNS(t'/p)$ can return a nonempty subset of $\mathcal{R}_p$ for $p \in \mathcal{D}' \cap \mathcal{D}_{ne}$. Since there exists at least one such p we see that $FindNS(t')$ can return $\bigcup_{i=1}^{n} FindNS(t'/p)$, which can be a nonempty subset of $FindNS(t)$.

Now we show by induction on the length of a reduction sequence that if $t \rightarrow^* t'$ without reducing any of the redexes in a set $\mathcal{R} \neq \phi$ returned by $FindNS(t)$ then $FindNS(t')$ can return a nonempty subset of $FindNS(t)$. The basis is for the empty sequence whence the proposition holds vacuously. The induction step follows directly from the above result for a single step reduction.

Therefore, in every t' such that $t \rightarrow^* t'$, at least some of the redexes in $\mathcal{R}$ are present and so t' is not in NF. Thus the NF of t cannot be reached without reducing at least one redex in $\mathcal{R}$. ∎

6 Necessity of Reducing Covers

Observe that $Normalize$ may perform some "useless" reductions, e.g., whereas $or(or(T,F), or(F,F))$ can be normalized to T without reducing $or(F,F)$, $Normalize$ reduces this redex as well. However, we now show that the useless reductions performed by $Normalize$ cannot be avoided by any algorithm that does not examine rhs. Specifically, if any such algorithm does not always choose a set of redexes that includes a cover, then it is "incomplete". This means that $Normalize$ is in some sense optimal among such algorithms, since they always have to perform at every step, a superset of reductions performed by $Normalize$.

As usual we use the notion of *arbitrary reductions*, to capture the idea of "not looking at rhs". Formally, a term s can be reduced to another term t through an arbitrary reduction iff there is a redex occurrence p in s such that $t = s[p \leftarrow t']$, for some arbitrary term t'. The arbitrary reduction relation is denoted by the symbol $\rightsquigarrow$ and its reflexive and transitive closure by $\rightsquigarrow^*$.

The arbitrary reduction $s \rightsquigarrow t$ captures the intuitive idea that there is a possibility that s can actually be reduced to t through a series of normal reductions (i.e., $s \rightarrow^* t$), and that this possibility cannot be ruled out by an algorithm that does not examine the rhs of equations. This means that an arbitrary reduction sequence $A : t \rightsquigarrow^* t_n$ (with t_n in NF) that replaces redexes $s_1, ..., s_n$ by $r_1, ..., r_n$ respectively captures the possibility that $s_1, ..., s_n$ can be reduced through normal reductions to $r_1, ..., r_n$ respectively, and if so t has a normal form t_n. Each arbitrary reduction sequence that normalizes t captures one such possible case. Since a normalization algorithm *cannot rule out any* such case without looking at the rhs, we say that it is *complete w.r.t. arbitrary reductions* if in every case that corresponds to each arbitrary reduction sequence, it normalizes t. Specifically, for the sequence A mentioned above, if the normalizing algorithm chooses a redex $s_i (1 \leq i \leq n)$ then the replacement will be r_i since in the case corresponding to A, s_i can be reduced to r_i (through normal reductions). Otherwise, if the redex chosen is not an s_i then note that the NF of t can be reached irrespective of what the replacement for this redex is. Therefore the sequence generated by the normalizing algorithm must normalize t irrespective of the replacement for such redexes.

We now establish the necessity of reducing covers by showing that an algorithm is complete w.r.t. arbitrary reductions iff it always selects (for replacement) a set of redexes that includes a cover. This shows that no algorithm based on lhs alone can do less wasteful reductions than our algorithm. It is in this sense that we use the term "relaxes laziness minimally" when we talk about how our algorithm deals with systems that are not strongly sequential.

Our focus here is on *legal* (or error-free) terms, in which no subterm can have a head normal form[6] t with a function symbol at root of t. There are two important reasons why we focus on legal terms. Firstly, in languages that follow the constructor discipline (such as ML, HOPE and Miranda without its lawful types), it is an error attempting to normalize a term that is not legal. In particular, attempting to normalize an illegal term will result in a run-time error. Second reason is that for terms that are not legal, it is possible in some cases to reduce the size of the necessary set selected by $FindNS$. In particular, we can use a bottom-up procedure to first mark all terms that are in head normal forms. Based on this information, we may be able to use case 2b of the algorithm whereas a top-down method such as $FindNS$ would have chosen case 2c.2. However, adding such a bottom-up component to $FindNS$ will incur significant overheads. Given that our interest is in the context of programming and that the improvement can be there only in for terms

[6] A term t is in head normal form if no t' such that $t \rightarrow^* t'$ contains a redex at the root.

that would cause run-time errors, we believe that the overhead is not worth the gain of being able to handle terms that are not legal. Therefore we deal with legal terms only in our proof of necessity of reducing covers.

If a normalizing algorithm chooses for replacement a set of redexes that is not a superset of a cover then all these redexes could potentially be "useless" and therefore may not make any further progress towards NF. This means,

Lemma 15 *If a normalizing algorithm $\mathcal{N}$ selects (for replacement) a set of redexes $\mathcal{R}$ in a legal term t such that $\mathcal{R}$ does not include any cover, then $\mathcal{N}$ is* **not** *complete.*

Proof: We first show that there exists an arbitrary reduction sequence A that normalizes t without reducing any redex in $\mathcal{R}$. There are two cases to consider.

Case I: t has a function symbol root. We show by induction on height of t the stronger result that there is an A that reduces t to an *arbitrary term* in normal form (without reducing any redex in $\mathcal{R}$). The basis is for a term of height zero and so t is a redex (whence $FindNS(t) = \{\Lambda\}$) or is in NF (whence $FindNS(t) = \phi$). In the former case, any normalizing algorithm must reduce the only redex in a term and so $FindNS(t) \subseteq \mathcal{R}$ and in the latter case $\mathcal{R}$ vacuously includes $FindNS(t)$. Thus in both cases the antecedent does not hold. For the induction step, assume that the lemma holds for terms whose height is less than k. Consider the following cases for a term t of height k:

Case 1 t is a redex: In this case, A will be the sequence that replaces the root redex by an arbitrary term in normal form.

Case 2 t is not a redex:

Case 2a $\mathcal{D}_{ne} = \phi$: t is in NF with a function symbol root and hence is not legal.

Case 2b u is not a partial redex: This implies t is in head normal form with a function symbol root, and hence is not legal.

Case 2c u is a partial redex: Consider the following two subcases:

Case 2c.1 $\exists \mathcal{D}' \subseteq \mathcal{D} - \mathcal{D}_{ne}$ such that $\bigcup_{p \in \mathcal{D}'} \mathcal{S}_p = \mathcal{L}_u$: Once again, this is not possible for legal terms since such a term is in head normal form with a function symbol at its root.

Case 2c.2 No such $\mathcal{D}'$ exists: Let $p_1, ..., p_r$ be all paths such that u/p_i is a variable. Let $\mathcal{R}_i$ be the set of redexes in $\mathcal{R}$ that are inside t/p_i. Now we claim that $\exists l \in \mathcal{L}_u$ such that for every $\mathcal{S}_{p_j}$ that includes l, $\mathcal{R}_j$ does not include any cover. (Otherwise, it must be the case that $\forall l \in \mathcal{L}_u$ there is an i such that $l \in \mathcal{S}_{p_i}$ and $\mathcal{R}_i$ includes a cover for t/p_i. But then $\mathcal{R}$ will include a cover for t, which contradicts our assumption.) By inductive hypothesis, each t/p_j such that l/p_j is a nonvariable can be normalized to an arbitrary term and therefore into an instance of l/p_j without reducing any redex in $\mathcal{R}$. Thus t can be reduced to a redex for l which can then be rewritten into an arbitrary term. By concatenating the sequences to normalize t/p_j's (for j such that l/p_j is a nonvariable) and following these reductions with the reduction at the root we get an arbitrary sequence to reduce t to an arbitrary term in NF without reducing any redex in $\mathcal{R}$.

Case II: t has a constructor root. The proof is once again by induction on height of t. As before, let $p_1, ..., p_r$ be all paths such that u/p_i is a variable and $\mathcal{R}_i$ be the set of redexes in $\mathcal{R}$ that are inside t/p_i. In this case, since the algorithm does not choose a necessary set computed by $FindNS$, by case 2b of $FindNS$, we know that if $\mathcal{R}_i$ is nonempty then it does not include a cover for t/p_i. Therefore, for such i, there exist (by induction hypothesis) arbitrary sequences that normalize t/p_i without reducing the redexes in $\mathcal{R}_i$. By concatenating these sequences (along with sequences normalizing subterms t/p_i for i such that $\mathcal{R}_i$ is empty), we get a sequence to normalize t that does not reduce any redex in $\mathcal{R}$.

Having established the existence of A, we now show that $\mathcal{N}$ does not generate a normalizing sequence corresponding to A. Since A reduces no redex in $\mathcal{R}$, $\mathcal{N}$ must find NF of t irrespective of the replacement for these redexes. By specifying the replacement for each r_i in $\mathcal{R}$ to be r_i itself, $\mathcal{N}$ will loop for ever, repeatedly reducing t to itself. ∎

In contrast, using the completeness proof (refer corollary 11 in section 4) we show that $Normalize$ is complete w.r.t. arbitrary reductions. First, in order to check whether a redex reduced by $Normalize$ is the same as the one reduced in an arbitrary reduction sequence, we attach unique labels to nodes with function symbols in the input term. (Since all redexes have a function symbol at their roots, there is no need to label constructor nodes.) We denote a function symbol f with a label j as f^j. When a reduction is performed, the nodes with function symbols in the replacement term are given new labels. Now if a redex reduced in the arbitrary reduction sequence has the same label as another redex reduced by the normalizing algorithm then the same labelled term will be the replacement for both redexes.

Lemma 16 *$Normalize$ is complete with respect to arbitrary reductions.*

Proof: We construct a rewrite system E corresponding to each arbitrary reduction sequence A : $t_0 \rightsquigarrow t_1 \rightsquigarrow t_2 \rightsquigarrow \cdots \rightsquigarrow t_n$ such that $t_i \rightarrow t_{i+1} (0 \leq i < n)$ by rules in E. This is done by making E a system with labelled function symbols on lhs, i.e., E includes lhs $f^j(s_1, ..., s_{rank(f)})$ for $j = 1, 2, ...$ whenever the original system includes $f(s_1, ..., s_{rank(f)})$. The rhs are specified as follows. If a redex $f^j(s_1, ..., s_{rank(f)})\beta$ was replaced by s' in A, then the rhs for $f^j(s_1, ..., s_{rank(f)})$ is specified as s'. Otherwise, the rhs can be specified to be any arbitrary term. It is clear that $t_i \rightarrow t_{i+1}$ in E. Therefore, by corollary 11, $Normalize$ generates a sequence to normalize t in E. Now we show that this is the normalizing sequence generated by $Normalize$ that corresponds to A (when given the original set of lhs and the unlabelled term $unlabel(t)$).

Note that by construction of E and since $Normalize$ depends only on lhs, the actions taken by $Normalize$ on a prefix $f^j(s_1, ..., s_{rank(f)})$ (in system E) will be the same as on $f(s_1, ..., s_{rank(f)})$ (in the original system). This means that if $Normalize$ chooses a set $\mathcal{U}$ of redexes when given the system E and a labelled term t, then it will choose the same set when given the original system of rules and the unlabelled term $unlabel(t)$. Furthermore, the replacements in E are such that the redexes replaced in A are replaced by the same terms, and for other redexes the replacement is arbitrary (by construction of E). Therefore the replacements are consistent with that specified by our criteria for completeness and hence the sequence generated for E is the normalizing sequence corresponding to A. ∎

In conclusion,

Theorem 17 *An algorithm that does not examine rhs is complete iff it always chooses for replacement a set of redexes that includes a cover.*

From the proofs of lemmas 15 and 16 it immediately follows that $FindNS$ tightly characterizes strongly necessary sets when the notion of arbitrary reductions is used to formalize the idea of "reasoning based on lhs alone", i.e.,

Corollary 18 *A set of redexes is strongly necessary iff it includes a cover.*

7 Implications

In this section we discuss some interesting implications of our results. First,

Corollary 19 *Parallel outermost strategy is complete.*

Proof: Immediate, since the set of outermost redexes in t is a super set of $FindNS(t)$. ∎

Second, Huet-Levy strategy emerges as a special case of our strategy for strongly sequential systems.

Corollary 20 *For strongly sequential systems $FindNS(t)$ returns a singleton set containing a needed redex.*

Proof: We first show that for a strongly sequential system, every partial redex u (that is not already a redex) has an index. Let u be a partial redex and u' be its maximal (in the partial order $\leq$) prefix such that $\forall l \in \mathcal{L}_{u'} u' \leq l$. Then by definition of strong sequentiality II (definition 4), u' has an index p. Note that by maximality of u', u/p is a variable. (Otherwise, consider an u'' obtained from u' by replacing the variable at u'/p by the symbol at u/p. Note that since p is an index for u', $\forall l \in \mathcal{L}_{u'}$ l/p is a nonvariable. So $\forall l \in \mathcal{L}_{u''}$ $l \geq u''$ and this contradicts the maximality of u'). It is also clear that $\mathcal{L}_u \subseteq \mathcal{L}_{u'}$ and hence p is an index for u. Therefore every MCP that is a partial redex (but not a redex) has an index.

Now note that if p is an index of u then $\mathcal{S}_p = \mathcal{L}_u$. Therefore in strongly sequential systems, it is possible in case 2c.2 of $FindNS$ to choose $\mathcal{D}'$ to be the singleton set consisting of the index. This will ensure that we always obtain a singleton necessary set. Then it follows from soundness of $FindNS$ that the only member of such a set is a needed redex. ∎

Theorem 21 *A complete normalization algorithm based on lhs can be lazy iff the program is strongly sequential.*

Proof: The if part is immediate from the preceding lemma. For the only if part, note that if a system is not strongly sequential then there exists a partial redex u which has no index. Now consider a term t obtained by instantiating each variable in u with a redex. Let $\mathcal{R}$ be any necessary set returned by $FindNS(t)$. Since u has no index, it follows that there are two paths p and q such that $\{t/p, t/q\} \subseteq \mathcal{R}$ and $\mathcal{S}_p \neq \mathcal{S}_q$. So there must be an l in $\mathcal{S}_p$ that is not in $\mathcal{S}_q$. Let $p_1, ..., p_k$ be paths such that l/p_i is a nonvariable and u/p_i is a variable. Now specify the rhs of the rule for which t/p_i is a redex, as an instance of l/p_i. Also specify the rhs for l to be a constructor c. Now t can be normalized by reducing t/p_i (for $i = 1, ..., k$) and following it with root reduction. Note that t/q need not be reduced (since it is "thrown away" by the root reduction) to reach NF and thus there is an useless redex in any set $\mathcal{R}$ returned by $FindNS(t)$. However, it follows from the proof of theorem 17 that any complete normalization algorithm based on lhs must reduce all redexes in some set returned by $FindNS$ and so must reduce such "useless" redexes and hence cannot be lazy. ∎

8 Automata-Driven $Normalize$

Observe that if $Normalize$ is implemented directly then in every invocation of $FindNS$ at run-time, we will have to compute MCP, the corresponding $\mathcal{L}_u$ and then select $\mathcal{D}'$. This is very expensive.

Firstly, computing MCP may involve traversal of the entire term. Secondly, we need to examine all the lhs to construct $\mathcal{L}_u$. Finally, note that for efficiency, it is crucial to select a small $\mathcal{D}'$. However, we show that

Theorem 22 *Computing a $\mathcal{D}'$ with minimum number of elements is NP-hard.*

Proof: is by reducing set cover problem to an instance of computing $\mathcal{D}'$ with minimum number of elements. Let $\mathcal{C}_1, ..., \mathcal{C}_n$ with $\bigcup_{i=1}^{n} \mathcal{C}_i = \{1, ..., m\}$ be an instance of the set cover problem. Consider the orthogonal TRS $\{l_1 \rightarrow c, ..., l_m \rightarrow c\}$ with each l_i of the form $f(v_1, ..., v_n)$, where v_j (for $j = 1, ..., n$) is a constructor c if $\mathcal{C}_j$ includes i and a variable otherwise. For the prefix $u = f(x_1, ..., x_n)$, observe that $\mathcal{L}_u = \{1, ..., m\}$ and $\mathcal{S}_i = \mathcal{C}_i$. Therefore the minimum $\mathcal{D}'$ will specify the sets $\mathcal{C}_k$ that are in the minimum set cover for $\{1, ..., m\}$. ∎

We now show how to compile the program into a DFA so that by precomputing $\mathcal{L}_u$ and $\mathcal{D}'$ (for all possible prefixes) at compile time, their computation is completely avoided at run-time. Furthermore the automaton guides $FindNS$ to examine a small prefix of MCP of a term that embodies all the information contained in the MCP. Specifically, the prefix u examined is such that (in case 2c of $FindNS$) there is a function symbol in the input term at every $p \in \mathcal{D}'$. We now describe the matching automaton and proceed to give an algorithm to construct it.

The states v of the (matching) automaton "remember" the prefix u_v inspected in reaching that state from the start state. Since $Normalize$ reduces only redexes below $p \in \mathcal{D}'$, the function symbols at these positions alone may change to constructors. Therefore v will have transitions on all $p \in \mathcal{D}'$ to dummy nodes v_p, from where there will be transitions on all constructors c such that $c = root(l/p)$ for some $l \in \mathcal{L}_{u_v}$. There will also be a transition from v_p on '?', a new constructor not in $\mathcal{C}$, which is taken on seeing a constructor symbol different from those on the other edges. Observe that if for a prefix u, $u/p =$ '?' then $\mathcal{L}_u$ will consist only of lhs that have a variable at some $q \leq p$.

$FindNS(t)$ makes use of the automaton as follows. If there is a transition on a path p from the current state v and there is a constructor c in the input term at p then a transition is made on p and then on c. Otherwise, it must be the case that for all p on which there is an outward transition from v there is a function symbol at t/p. Then these paths constitute $\mathcal{D}'$. Putting this all together, an algorithm to construct the matching automaton is shown in fig.7. In the algorithm, we assume that $\mathcal{D}'$ is a minimal cover. (It may indeed be a *minimum* cover.)

Procedure *build* is invoked by the call $build(root_f, f(x_1, ..., x_{rank(f)}))$, where $root_f$ is the root of the matching automaton for all lhs with outermost symbol f. It constructs the matching automaton recursively in depth first order. The parameter v is the current state of the automaton and u is the prefix that corresponds to this state. In lines 1 and 2 the termination conditions are checked. In line 3 we select a $\mathcal{D}'$. The dummy states corresponding to each $p \in \mathcal{D}'$ are created in the loop in steps 4–6. $build1$ is invoked on each of these dummy nodes at line 6 to extend the partial redex u.

$build1$ has three parameters – p is the path at which the partial redex u is to be extended and v is the current dummy node. $build1$ finds out the possible symbols that could appear at p and creates a transition for each of them. A default transition on '?' is also created. Partial redex u is appropriately updated at lines 9 and 11 to reflect the transition. The construction of the automaton is continued by the call to *build* in lines 9 and 11.

The automaton constructed by *build* for the rules in example 12 is shown in fig. 8. The correctness of *build*, (i.e., it correctly guides $FindNS$ to identify a strongly necessary set) can be

```
procedure build(v, u)
1.  if 𝓛_u = φ then mark v a failure state
2.  else if ∃l ∈ 𝓛_u such that u ≥ l
        then mark v final and set match[v] := l
3.  else choose 𝓓' = {p_1, ..., p_r} such that ⋃_{p∈𝓓'} 𝓢_p = 𝓛_u
4.      for j := 1 to r do
5.          create a new node v_j and an edge from v to v_j labelled p_j
6.          build1(p_j, v_j, u)
        end
    end
end

procedure build1(p, v, u)
7.  for each c ∈ 𝓒 for which ∃l ∈ 𝓛_u l/p = c do
8.      create a new node v_c and an edge from v to v_c labelled c
9.      build(v_c, u[p ← c(y_1, ..., y_rank(c))])
    end
10. create a new node v_'?' and an edge from v to v_'?' labelled '?'
11. build(v_'?', u[p ← '?'])
end
```

Figure 7: Algorithm for building a Matching Automaton

established based on the following observations:

1. In an invocation of $build(v, u)$, u correctly denotes the prefix of the input term examined in reaching v from the start state.

2. The choice of $\mathcal{D}'$ at line 3 coincides with the choice made by $FindNS$ in case 2c. Also, if v is marked a failure node at line 1 then u is not a partial redex (case 2b).

3. Every possible extension of u at p into a larger prefix is considered in line 8 and line 10 of $build1$.

Based on these observations, a formal proof (that $build$ correctly identifies $\mathcal{D}'$ and when the prefix does not match any lhs) can be easily established by induction on the maximum length of any path from v to any final state.

We also need to show that $build$ terminates. To prove this, let s_l denote the number of nonvariable symbols in an lhs $l \in \mathcal{L}_u$ that are not part of u. Now consider the quantity $d_u = \Sigma_{l \in \mathcal{L}_u} s_l$. Note that d_u reduces with each invocation of $build$ at line 9. This is because, by choice of p, at least one $l \in \mathcal{L}_u$ has a nonvariable at p and hence the quantity s_l decreases, which in turn implies that d_u decreases.

Observe that the automaton has a tree structure with several redundant states. (The equivalent states have been given identical labels.) Two distinct states may have same constructor prefix u because of the different orders in which the symbols in u were examined (e.g., states labelled '1').

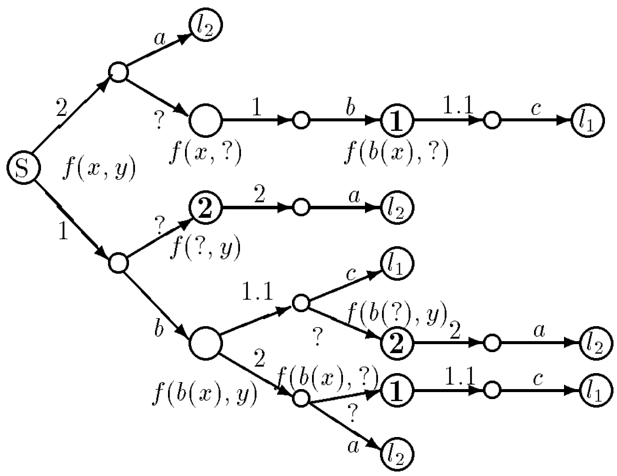

Figure 8: Matching Automaton for example 12. (Failure states not shown).

Even otherwise, states with distinct prefixes u_1 and u_2 may be equivalent because these prefixes differ in positions that "need not be examined" to identify a match. To make this idea more precise, we define the concept of a *representative set* $\overline{\mathcal{L}_u}$ of a prefix u which is a minimal set of all lhs which can match any term $v \geq u$. Observe that such a v is an instance of u as well as an $l \in \mathcal{L}_u$, and therefore an instance of $unify(l,u)$. If there is an $l' \in \mathcal{L}_u$ such that $unify(l,u) \geq unify(l',u)$ then any redex for l is also a redex for l'. If we always choose l' over l at step 2, there will be no match state for l that can be reached from a state with prefix u. Note that this choice has no effect other than changing the annotation (indicating the matching lhs) on the final state and thus the structure of the automaton below this state does not depend on l and so $\overline{\mathcal{L}_u}$ does not contain l. More precisely, $\overline{\mathcal{L}_u} = \{l \in \mathcal{L}_u \mid \exists \text{ no other } l' \in \mathcal{L}_u \text{ such that } unify(l',u) \leq unify(l,u)\}$ and can be computed by successively removing from $\mathcal{L}_u$ the lhs that need not be retained. For instance $\overline{\mathcal{L}_{f(b(x),a)}} = \{l_2\}$ and $\overline{\mathcal{L}_{f(b(c),x)}} = \{l_1\}$, whereas the match set for both prefixes is $\{l_1, l_2\}$. Observe that using $\overline{\mathcal{L}_u}$ in the place of $\mathcal{L}_u$ does not change the resulting automaton except (possibly) for the matches indicated on the final states. This is because at every path $p \in \mathcal{D}$ where l' has a nonvariable, l has a nonvariable too and so $\bigcup_{p \in \mathcal{D}'} = \overline{\mathcal{L}_u}$ holds exactly when $\bigcup_{p \in \mathcal{D}'} = \mathcal{L}_u$ also holds.

Now suppose u and u' have the same representative set $\overline{\mathcal{L}_u}$ and differ only in positions p such that $\forall l \in \overline{\mathcal{L}_u}$ l/q (where $q \leq p$) is a variable (e.g., states labelled '2'). Then it is clear that two states with prefixes u_1 and u_2 are equivalent. Based on this we define the *relevant prefix* of u as the term obtained by replacing with a new constant '?' all those subterms u/p such that l/q $(q \leq p)$ is a variable $\forall l \in \overline{\mathcal{L}_u}$. For instance, the relevant prefix of $f(b(x),a)$ is $f(\text{'?'},a)$ and that of $f(b(\text{'?'}),x) = f(\text{'?'},x)$. By showing that two states are equivalent iff the corresponding relevant prefixes are identical, we establish

Theorem 23 *The automaton obtained by merging states with identical relevant prefixes is optimal.*

Proof: First we need to show that the states merged as above are indeed equivalent. Note that only the relevant prefix takes part in deciding $\overline{\mathcal{L}_u}$ and in choosing $\mathcal{D}'$ (step 3 of *build*). This implies that the subtrees created by two distinct invocations $build(v_1,t_1)$ and $build(v_2,t_2)$ will be identical if both t_1 and t_2 have the same relevant prefixes and so v_1 and v_2 are equivalent.

Now we need to show that no two states v_1 and v_2 with distinct relevant prefixes u_1 and u_2 are equivalent. If $\overline{\mathcal{L}_{u_1}} \neq \overline{\mathcal{L}_{u_2}}$, let $l \in \overline{\mathcal{L}_{u_1}}$ and $l \notin \overline{\mathcal{L}_{u_2}}$. Then, by the properties of representative set (and correctness of the matching automaton), there is a path from v_1 to a matching state for l. On the other hand, there is no such path from v_2 and hence v_1 and v_2 are not equivalent.

If $\overline{\mathcal{L}_{u_1}} = \overline{\mathcal{L}_{u_2}}$ then $\exists p\ root(u_1/p) \neq root(u_2/p)$, since $u_1 \neq u_2$. Since the representative sets are the same, u_1 and u_2 cannot have different constructor symbols at p. Hence one of these relevant prefixes, say u_1, has a constructor at p and the other has a variable at p. If this constructor is '?', every lhs in the representative set must contain a variable at p. The definition of relevant prefix then implies that $root(u_2/p)$ must also be '?'. Since we assumed that u_1 and u_2 differ at p, this is also not possible. Therefore $root(u_1/p)$ must be a $c \in \mathcal{C}$. Let l be an lhs in the representative set such that $l/p = c$. Now, there must be a path from v_1 to a matching state for l, and the symbol at p is not examined on this path. In contrast, the symbol at p is examined on every path from v_2 to a matching state for l. Therefore v_1 and v_2 are not equivalent. ∎

Based on this, we directly build an optimal matching automaton as follows. A new state is created (step 8 or 10) only when there is no other state v with the same relevant prefix; otherwise a transition to v is created and the recursive call (step 9 or 11) to construct the automaton below v is skipped. This direct construction avoids repeated reconstructions of identical subtrees and minimizes the runtime storage requirement with no compile-time overhead for minimizing a non-optimal automaton. Sharing equivalent states in the automaton of fig. 8 leads to an optimal automaton with 7 states as shown in fig. 9 (as against 14 in fig. 8).

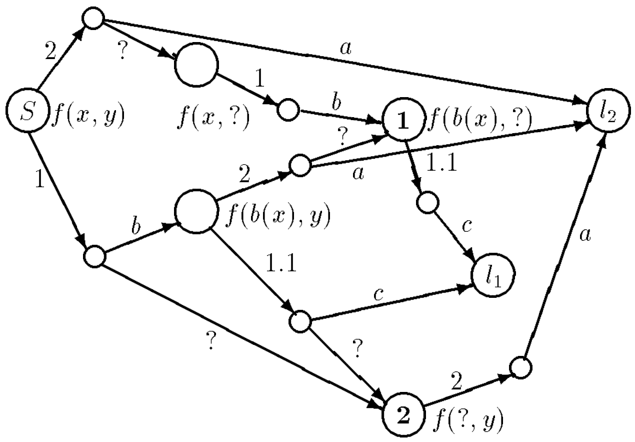

Figure 9: Optimal automaton for Example 12.

8.1 Discussion of Time Complexity of *FindNS*

In this section, we make some remarks on the time complexity of *FindNS*. The treatment in this section is intuitive and not very formal and is meant to point out that *FindNS* is indeed very efficient. More importantly, *FindNS* is incremental in the sense that a search for a necessary set need not be started at the root of the term after each rewrite; it is possible to start the search only "around" the portions of the term that were rewritten. We will explain below why such incrementality is critical. In fact, incrementality was an important design goal in the development of the algorithm *FindNS*.

Observe that the only expensive operations in $FindNS$ are computation of $\mathcal{L}_u$ and $\mathcal{D}'$, identification of MCP and checking whether the MCP is a redex or a partial redex. All these operations are now simplified by the automaton, which lets us perform all these operations in unit time. Based on this, we can easily show that $FindNS$ takes time linear in size of the input term. We do not prove this result formally since a linear algorithm is not necessarily efficient. For instance, consider a program to append two lists and the term $append(t_1, t_2)$, where t_1 is a list of size n. If $FindNS$ takes linear time to compute a necessary set, then it will take quadratic time to normalize $append(t_1, t_2)$. This is clearly unacceptable since this operation takes only linear time in almost every declarative language that is currently in use. What we need is that $FindNS$ be incremental. In particular, we would like successive invocations of $FindNS$ to take very little time – time proportional only to the changes made as part of reductions in each step. Such an algorithm would enable us to normalize the above term in linear time.

Consider an invocation $FindNS(t)$, where t is not in normal form. Suppose that this invocation results in recursive invocations of $FindNS(t/p_1), ..., FindNS(t/p_n)$ in case 2c.2. These invocations may in turn result in further recursive invocations until a necessary set is identified for t. Now suppose that these redexes are rewritten. Then we can make the following observation:

If none of the symbols at $t/p_1, ..., t/p_n$ have changed into constructors then $FindNS(t)$ will once again invoke $FindNS(t/p_1), ..., FindNS(t/p_n)$.

This is because of the following reason: if none of the symbols at $t/p_1, ..., t/p_n$ change, then $MCP(t)$ will be unchanged after the reduction and hence $FindNS$ will once again choose the same $\mathcal{D}'$. This same argument holds in cases 2b and 2c.1 of $FindNS$.

The above observation implies that we do not have to search the entire term for necessary redexes each time after a rewrite. We can "localize" the search to those portions of the term that were changed as a result of the reductions. In particular, it is sufficient to restart $FindNS$ exactly on those subterms s of t such that one of the following conditions hold:

1. s is a redex that was returned by $FindNS(t)$ and was hence rewritten. After the rewrite, the root of s is still a function symbol, OR,

2. $FindNS(t)$ resulted in invocation of $FindNS(s)$, the invocation of $FindNS(s)$ immediately resulted in a recursive invocation of $FindNS(s/p)$. Moreover, the reduction changed the symbol at s/p into a constructor.

Intuitively, the reason for limiting the search to such s's is that portions above them are unchanged as a result of rewrites. Therefore, the behavior of $FindNS$ will be identical on these portions before and after the rewrite .

9 Concluding Remarks

In this paper we developed a complete normalization algorithm for the entire class of orthogonal systems which was based on the concept of necessary sets. Unlike needed redexes which are neither computable in general nor always exist, we showed that *at least one* necessary set can always be computed in any term that is not in normal form. Our algorithm is "optimal" among normalization algorithms that are based on left-hand sides alone. This implies that our algorithm is lazy on strongly sequential parts of a program and "relaxes laziness minimally" to handle the other parts and thus does not sacrifice generality for efficiency.

Finally, we remark that it is possible to improve the efficiency of our algorithm by reasoning about right-hand sides using some form of strictness analysis. We are currently investigating how some of the techniques developed in [SPR90] can be applied to reduce the size of necessary sets.

References

[BMS80] R.Burstall, D.MacQueen and D.Sanella, HOPE: An Experimental Applicative Language, *Proceedings of the 1st International LISP Conference, 1980.*

[DJ90] N. Dershowitz and J.P. Jouannaud, Rewrite Systems, *Handbook of Theoretical Computer Science, Ch. 6, Vol II, North-Holland, 1990.*

[HMT88] R.Harper, R.Milner and M.Tofte, The Definition of Standard ML, *Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, University of Edinburgh, 1988.*

[HO84] C. Hoffmann and M.J. O'Donnell, Implementation of an Interpreter for Abstract Equations, *Proceedings of the 10th ACM Symposium on Principles of Programming Languages, 1984.*

[HL79] G. Huet and J.J. Levy, Computations in Nonambiguous Linear Term Rewriting Systems, *Tech. Rep. No. 359, IRIA, Le Chesney, France, 1979.*

[Ken89] J.R. Kennaway, Sequential Evaluation Strategies for Parallel-Or and Related Reduction Systems, *Annals of Pure and Applied Logic 43, pp. 31–56, 1989.*

[KM89] J.W. Klop and A. Middeldorp, Strongly Sequential Term Rewriting Systems, *Report CS-R8932, Centre for Mathematics and Computer Science, Amsterdam, 1989.*

[KP78] G. Kahn and G. Plotkin, Domaines Concretes, *Tech Report 336, IRIA Laboria, Le Chesney, France, 1978.*

[Lav88] A. Laville, Implementation of Lazy Pattern Matching Algorithms, *Proceedings of the European Symposium on Programming, Springer-Verlag LNCS 300, 1988.*

[OD77] M.J. O'Donnell, Computing in Systems described by Equations, *Springer LNCS 58, 1977.*

[OD85] M.J. O'Donnell, Equational Logic as a Programming Language, *MIT Press, 1985.*

[OPRRS92] K. Owen, S. Pawagi, C.R. Ramakrishnan, I.V. Ramakrishnan and R.C. Sekar, Fast Parallel Implementation of Lazy Languages: The Equals Experience, *Proceedings of the ACM Conference on Lisp and Functional Programming, 1992.*

[PS90] Laurence Puel and Ascander Suarez, Compiling Pattern Matching by Term Decomposition, *Proceedings of the ACM Conference on Lisp and Functional Programming, 1990.*

[SPR89] R.C. Sekar, S. Pawagi and I.V. Ramakrishnan, Transforming Strongly Sequential Rewrite Systems with Constructors for Efficient Parallel Execution, *Proceedings of the International Conference on Rewriting Techniques and Applications, 1989.*

[SPR90] R.C. Sekar, S. Pawagi and I.V. Ramakrishnan, Small Domains Spell Fast Strictness Analysis, *Proceedings of the ACM Conference on Principles of Programming Languages, 1990.*

[SR92] R.C. Sekar and I.V. Ramakrishnan, Programming with Equations: A Framework for Lazy Parallel Evaluation, *Proceedings of the 11th International Conference on Automated Deduction, 1992.*

[SRR92] R.C. Sekar, R. Ramesh and I.V. Ramakrishnan, Adaptive Pattern Matching, *Proceedings of the International Conference on Automata, Languages and Programming, 1992.*

[Str89] R.I. Strandh, Classes of Equational Programs that Compile into Efficient Code, *Proceedings of the International Conference on Rewriting Techniques and Applications, 1989.*

[That84] S. Thatte, Demand driven Evaluation with Equations, *Technical Report CRL-TR-34-84, University of Michigan Computing Research Laboratory, Ann Arbor, 1984.*

[That85] Satish Thatte, On the correspondence between two classes of Reduction systems, *Information Processing Letters 20 (2), pp. 83-85, 1985.*

[That87] S. Thatte, A Refinement of Strong Sequentiality for Term Rewriting With Constructors, *Information and Computation 72(1), 1987.*