A Framework for Lazy Parallel Evaluation of Equational Programs*

R.C. Sekar†, Shaunak Pawagi and I.V. Ramakrishnan
Department of Computer Science
SUNY at Stony Brook, NY 11794.
E-mail: {rcs,ram}@sbcs.sunysb.edu

Abstract

Strongly sequential systems, developed by Huet and Levy [HL79], has formed the basis of equational programming languages. Experience with such languages so far suggests that even complex equational programs are based only on strongly sequential systems with constructors. However, these programs are not readily amenable for efficient parallel execution. This paper introduces a class of strongly sequential systems called *path sequential systems*. We show that path sequential programs are natural for parallel evaluation. In particular, path sequential systems permit matching and search for redexes to proceed independently along different paths in a term. Such independent search, which avoids excessive synchronization and communication overheads, is not possible in strongly sequential systems. We describe a sound and complete parallel normalization algorithm for path sequential program and analyze its complexity. We then discuss the issue of expressive power of path sequential systems. By presenting a simple, syntactic method to transform any strongly sequential system with constructors into an equivalent path sequential system, we establish that the two classes have equal expressive power. Finally, we discuss other notions of sequentiality based on traversal orders (such as O'Donnell's strong left-sequentiality) and show that all such notions of sequentiality are equivalent under our transformation.

1 Introduction

Equational logic programming is a style of declarative programming in which systems of oriented equations (called *rewrite rules*) of the form $l \rightarrow r$ (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). This normal form is obtained 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.

Huet and Levy [HL79] pioneered lazy evaluation for equational programs, based on the concept of *needed redexes*. A needed redex for a term t is one that is reduced in every reduction sequence that normalizes t. They devised an algorithm for computing needed redexes for programs that are *strongly sequential* and also showed that in general, such redexes cannot be computed for other

*Partially supported by NSF grants CCR-8805734 and CCR-9102159. Preliminary versions of some of the materials in sections 6 and 7 of this paper were presented at third International Conference on Rewriting Techniques and Applications, 1989. A preliminary version of materials in sections 3,4 and 5 was presented in 11th international conference on Automated Deduction, 1992.

†Current Address: Bellcore, Rm 2A-274, 445 South Street, Morristown, NJ 07962-1910, E-mail: sekar@thumper.bellcore.com

programs. Their strategy for identifying needed redexes is driven by *lazy pattern-matching*, which means that we identify instances of lhs *without inspecting* any portion of the input term that corresponds to a variable in the matching lhs. They showed that this strategy reduces only needed redexes and that it is *complete*, i.e., normalizes a term whenever it has a normal form. Hoffman and O'Donnell [HO84] developed an equational interpreter for *strongly left-sequential systems*, which are a subclass of strongly sequential systems. Strandh [Str89] defined the class of forward branching programs (which properly includes strongly left-sequential systems and is properly included in strongly sequential systems) and developed an implementation for such programs. Most programs in practice are strongly sequential [OD85] and those that are not, can usually be so transformed [Lav88, PS90, SRR92].

The normalization algorithms mentioned above are all sequential (since they search for and reduce only one needed redex at a time) and the design of an efficient algorithm for lazy parallel evaluation of equational programs has remained an interesting open problem. In this paper we address the design of such a strategy. For the purpose of developing this algorithm, we assume that our parallel machine consists of processors interconnected by a network. We summarize our results in the following section.

1.1 Overview of Results

A lazy parallel normalization algorithm must search for and reduce several needed redexes in parallel. A natural way to do this is to search (in parallel) along different paths in a term and reduce needed redexes[1]. To avoid excessive communication and synchronization overheads, the search on different paths should proceed independently. In section 3, we show that natural extensions of Huet-Levy strategy for parallel evaluation do not support this independent search and incur (potentially) large communication overheads. Another serious problem with such extensions is that they cause an exponential blow-up in the size of compiled code.

• We propose a new framework for lazy parallel evaluation of equational programs, based on the concept of *path sequentiality*. We show that in such systems, the search for needed redexes can proceed *independently* along different paths with low communication and synchronization overheads. More importantly, the size of the compiled code is linear in the size of input program.

• In section 4.1 we devise a parallel lazy normalization algorithm for path sequential systems and establish its soundness and completeness. In section 5 we show that the total time spent by $Normalize$ is bounded by the time required to perform the needed reductions and that the search for needed redexes does not add to this complexity. We also show that matching (i.e., checking for the presence of a redex) is done with minimal communication. Finally, we discuss how the parallelism exploited by our algorithm can be further enhanced.

• Although most programs in practice appear to be path sequential [OD85], these systems are restrictive when compared to strongly sequential systems. In section 6 we devise an algorithm that transforms any strongly sequential program into an *equivalent* path sequential system. Although this transformation may generate several rules from a single rule, we show (in section 7) that the number of rules in the transformed program is at most thrice the number of rules in the original program. Furthermore, the computational effort required to match and reduce a rule in the original system is equal to that required for matching and reducing all the rules derived from it in the transformed system (see section 8.2).

[1] A question here is why we restrict ourselves to reducing only needed redexes in the parallel case. If we commit processor and other resources for computations that are not needed, then we must have a priority mechanism to ensure that needed computations make progress. We also need to be able to preempt resources from unneeded computations and give them to needed computations when we run out of resources for the latter. Both these mechanisms which are necessary for ensuring completeness of implementation, can create significant run-time overheads.

• We have so far described three notions of sequentiality: path sequentiality, which is based on ability to perform (lazy) matching based on independent traversal of paths; strong left-sequentiality which is based on preorder traversal; and strong sequentiality which is based on ability to identify some traversal that can be used for lazy pattern matching. A natural question that arises is whether we can have other notions of sequentiality based on different orders of traversal. We answer this affirmatively in section 8.1.1, where we first formalize a notion of a top-down traversal and associate a notion of sequentiality with each such traversal order.

• Our transformation has established the equivalence of path sequential in strongly sequential systems (in terms of expressive power). A natural questions is whether this implies any thing about the relationships among the different notions of sequentiality mentioned above. In section 8.1.3, we establish that *all such notions of sequentiality are equivalent* under our syntactic transformation procedure.

• A significant outcome of our transformation is that the path sequentiality restriction is transparent to the programmer, i.e., the programmer can still write strongly sequential programs which are not path sequential. More generally, pattern-matching using one traversal order may be preferable over others for several reasons. For instance, preorder traversal is simpler and potentially more efficient than the arbitrary order of traversal required for matching in strongly sequential systems; independent traversal of paths is more suited for parallel evaluation. Our transformation algorithm suggests that we can base an implementation on any notion of sequentiality that is sufficiently inclusive, (i.e., most programs observe this notion of sequentiality) yet enables a simpler and more efficient lazy evaluation algorithm than the Huet-Levy algorithm for strongly sequential systems. However, we do not have to restrict programs to observe this notion of sequentiality: all strongly sequential programs can be handled through our transformation. The same can be done with existing implementations such as O'Donnell's (which is based on strong left-sequentiality), thereby making the strong left-sequentiality restriction transparent to the user.

• It may appear intriguing how the above transformation procedure transforms a system that does not support independent search for needed redexes along different paths into one wherein such searches are possible. This question is discussed at length in section 8.2.

Like most languages based on equations (such as ML, HOPE, etc.), we will also be dealing only with programs that follow the *constructor discipline*[2] throughout the rest of this paper. However, we wish to point out that the notion of path sequentiality can be easily extended to orthogonal systems that do not follow the constructor discipline; all our arguments regarding the suitability of path sequential systems for parallel evaluation would still hold. However, if Klop's conjecture [KM89] that detecting strong sequentiality is NP-complete holds, our transformation procedure cannot be extended to the whole class of orthogonal systems – at least not unless the extended method takes an exponential time. We do not concentrate on these questions regarding orthogonal systems that do not observe the constructor discipline since this discipline has been adopted in almost all programming languages.

2 Preliminaries

In this section we introduce the terminology commonly used in term rewriting systems. We also develop the notations that will be used throughout the rest of this paper.

All the symbols used in a term rewriting system are drawn from a non-empty *ranked alphabet* Σ and a countable set of variables. We use the obvious tree representation for terms, and use *terms* and *trees* interchangeably.

The notion of a *path* (also called *occurrence*) is used to refer to subterms in a term as follows.

[2] In such programs the outermost symbols in all lhs are distinct from those that appear inside any lhs.

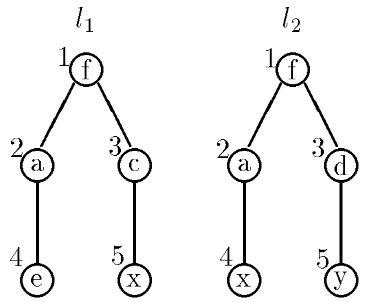

Figure 1

A path is either the empty string Λ that reaches the root or $p.i$ (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. We say $p \leq q$ whenever $\exists r \; p.r = q$. For a path $p = i_1.i_2....i_k$, let $s_0, s_1, ..., s_k$ be the symbols on this path in a term t (listed in the order from root to the node at p). The path string for a term t and a path p, denoted by $pathstr(t, p)$, refers to the string $s_0.i_1.s_1.i_2....i_k.s_k$. In fig.1, the paths 2.1 and 1 reach nodes 5 and 2 respectively in l_1, and $pathstr(l_1, 1.1) = f.1.a.1.e$.

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. A TRS is said to be a *constructor system* if $\mathcal{F}$ and $\mathcal{C}$ are disjoint, the lhs are linear (i.e., no variable in an lhs occurs more than once in it) and no two lhs unify. Throughout the rest of this paper we will be using the language of constructor systems and denote them by $\mathcal{R}$, $\mathcal{R}^N$, $\mathcal{R}'$ and $\mathcal{R}_i$. $\mathcal{L}$, $\mathcal{L}^N$, $\mathcal{L}'$ and $\mathcal{L}_i$ will denote the corresponding sets of lhs.

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. A *needed redex* is a redex that is rewritten in every reduction of t to normal form.

A reduction $t \rightarrow_l^p u$ means that u is obtained by replacing the redex $l\beta$ in t at path p by $r\beta$, where $[l \rightarrow r]$ is a rewrite rule. In this notation p and/or l may be dropped. The reflexive and transitive closure of $\rightarrow$ is denoted $\rightarrow^*$. A term t is in *head normal form* (HNF) if no term t' such that $t \rightarrow^* t'$ contains a redex at the root.

3 Issues in Lazy Parallel Evaluation

In this section we motivate the use of path sequential systems as the basis for lazy parallel evaluation of equational programs. We first describe the problems that arise when extending the known lazy sequential normalizing algorithm (due to Huet-Levy [HL79]) for parallel evaluation. We then show how these problems can be avoided by the use of path sequential systems.

The sequential normalizing algorithm operates by repeatedly searching for and reducing *one* needed redex. An intuitive way to parallelize this is to search and reduce several needed redexes in parallel. Such a search must be carried out in parallel along different paths in a term. We will show later on that extensions to the Huet-Levy algorithm for parallel evaluation do not allow such searches to be carried out independently on different paths. Consequently they incur a lot of communication and synchronization overhead and also cause an exponential blow-up in size of the compiled code. To show this we first outline the essential aspects of the lazy sequential normalizing algorithm for strongly sequential systems.

Suppose the TRS $\{f(a(e), c(x)) \rightarrow x, f(a(x), d(y)) \rightarrow x\}$ is used to normalize the term $t = f(a(t_1), t_2)$, where $t_1 = f(a(e), c(e))$ and $t_2 = f(a(c(e)), d(e))$. In order for t to become a redex, certain reductions must be performed within its subterms. In particular, t_2 must be reduced into a term with a constructor root (c or d). To identify such subterms, we define the following notions of *constructor prefix* and *index*[3].

Definition 1 *A constructor prefix u of a term t is a prefix such that all nonvariable symbols in u, except the one at its root, are constructors. A path p in u such that u/p is a variable is said to be an index of u with respect to a set of lhs $\mathcal{L}$ iff the following condition holds: $\forall\ l \in \mathcal{L}$ that unifies with u, l/p is a nonvariable.*

If all the lhs that unify with u have a nonvariable symbol at p then it is clear that the subterm t/p must be reduced to a term with constructor root (i.e., t/p must be in HNF) before t can become a redex. In the above example $u = f(a(x), y)$ and its index $p = 2$. Based on the notion of index, we can define strongly sequential systems as follows:

Definition 2 *In a strongly sequential system, every constructor prefix (or equivalently, any term with a function symbol at the root and no function symbols inside) that unifies with more than one lhs possesses an index.*

We remark that traditionally, the definition of various notions of sequentiality are based upon Kahn and Plotkin's [KP78] notion of a sequential predicate. In particular, a *match*-predicate is defined, which takes a prefix u as an argument and specifies whether there is a match for u. The system is said to be *match*-sequential if this *match* predicate is sequential at every prefix u. To determine if a system is *match*-sequential, we simply determine if every prefix u has a index with respect to the *match*-predicate. However, for matching in constructor systems, determining whether p is an index for u is equivalent to determining whether every lhs has a nonvariable at p [HL79]. Therefore we make use of the latter notion (which is more convenient for the development of materials in this section) for defining sequentiality.

The lazy normalization algorithm constructs such a prefix u, identifies its index p and reduces t/p to a term in HNF. If $root(t/p)$ is now a constructor (otherwise t can never become a redex) then u can be "extended" into a prefix u' by adding this constructor to it. This process is repeated until either t becomes a redex or we find that t can never become a redex (i.e., t is in HNF). In order to head normalize t/p the same process is used to identify and reduce redexes at the root of t/p until it is in HNF. In the above example t_2 is a redex and can be reduced to $c(e)$. The enlarged prefix $f(a(x), c(e))$ has 1.1 as an index. Now $t/1.1 = t_1$ is reduced to e, thereby yielding a redex at the root. On reducing this root redex we obtain the normal form e.

Thus matching in the above normalization algorithm involves identifying a prefix u, computing its index p, extending u into u' (after reducing the subterm at t/p into HNF) and repeating this on u' and so on till t becomes a redex. Since identification of a prefix and computation of its index are very time consuming, these operations will be driven by a matching automaton. (In [HL79], matching automata were called matching dags.) A state v of the automaton remembers the prefix u_v inspected in reaching this state from the start state. We (arbitrarily) choose one particular index of u_v and have transitions from v on pairs (p, c), where $c \in \{root(l/p) \mid l \text{ unifies with } u\}$. Fig.2 is such an automaton for the above example.

Note that each call to head normalize a subterm may reduce several redexes before returning. Thus significant amount of parallelism can arise if we head normalize in parallel all subterms t/p where p is an index for u. In a direct extension of the above lazy normalization algorithm for

[3] Our notion of a prefix coincides with the notion of an Ω-term used in [HL79, KM89], with variables in a prefix denoting unexamined portions of the input term. We deliberately blur the distinction such un-inspected portions (denoted by Ω in [HL79, KM89]) and a variable. This is because in linear terms, a variable stands for an unknown or arbitrary term, which correctly reflects our intuition that we have no knowledge of the structure of input term below the variables in a prefix.

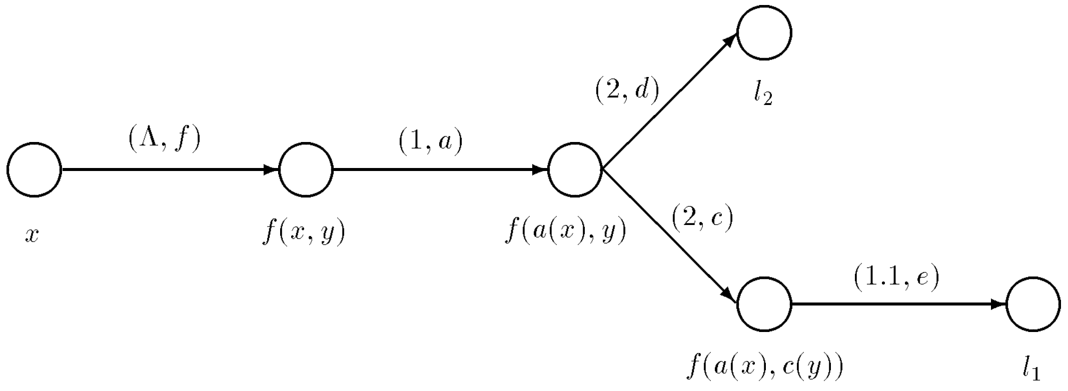

Figure 2: Automaton for patterns in fig.1

exploiting this parallelism, the processor P initiating a match at the root of t would first compute all such indexes $p_1, ..., p_k$ for a prefix u. Then it would initiate computations to head normalize all subterms t/p_i in parallel. Note that although head normalization of different subterms could proceed in parallel on different processors, the computation of the prefix and its indexes are all done *centrally* in P.

Such a centralized strategy suffers from two serious drawbacks. Firstly, P may have to successively copy (potentially) large number of nodes in order to extend the initial prefix $f(x_1, ..., x_n)$ (where f is the root symbol of t) into a redex. This may result in a significant amount of communication ($O(|l|)$ messages, where $|l|$ is the size of the lhs for which a redex is found). The second problem relates to the size of the automaton that drives such a matching process. Note that as soon as any one t/p_i is head normalized, the prefix u can be extended at p_i into an u' which may have additional indexes $q_1, ..., q_m$. In order to maximize parallelism, processor P must initiate computations to head normalize the subterms $t/q_1, ..., t/q_m$. However, the computations to head normalize all t/p_i's may complete in any order. Therefore P must be capable of extending the prefix at any p_i. This means that the state of the matching automaton that corresponds to having inspected the prefix u must have transitions on (p_i, c) for *all* $1 \leq i \leq k$ and c such that for some lhs l that unifies with u, $root(l/p_i) = c$. This contrasts sharply with the sequential case wherein there are transitions (p, c) for *only one* index p. The requirement to have transitions on all p_i's may result in the automaton having one state corresponding to each possible prefix of an lhs. The number of such prefixes can be exponential and thus the automaton can have exponential number of states. To see this, consider a TRS $\{l \rightarrow r\}$ where l is a complete binary tree of height h with only variables as leaves. Note that for a prefix u of l, any path p whose length is less than h (and for which u/p is a variable) is an index for u. It can be readily shown (by induction on the number of nonvariable symbols in a prefix) that the automaton will have states corresponding to every prefix of l.

We now show that the number of such prefixes is exponentially large. Let $T(h)$ denote the number of possible prefixes of a tree of height h. Note that a prefix of l is either a variable (i.e., the prefix is "empty") or $f(t_1, t_2)$ where f is the root of l and t_1 and t_2 are any prefixes of the left and right subtrees of l respectively. There are $T(h-1)$ prefixes for t_1 and t_2 and so

$$T(h) = 1 + (T(h-1))^2$$

with $T(1) = 2$. It can be readily established by induction on h that $T(h) \geq O(2^{2^{h-1}}) \geq 2^n$, where n is the number of nonvariables in the tree. Note that even if there are several rules in the TRS, the automaton can still have one state corresponding to each possible prefix of every lhs and thus exponential blow-ups can occur in practice.

Such a blow-up occurs because the index computation is based on the entire prefix. It is possible to avoid this blow-up if indexes can be computed based only on a part of the prefix. We therefore propose a new framework for lazy parallel evaluation based on *path sequential systems* (defined later) wherein we can decide whether a path p is an index based only on the symbols on this path in the prefix. This enables matching (and consequently the search for needed redexes) to be performed independently along different paths in a distributed fashion. Note that such an independent search is not possible with the centralized strategy for strongly sequential systems. For instance, in the example used to illustrate the Huet-Levy algorithm, note that although both t_1 and t_2 are needed redexes, t_1 can be chosen only after reducing t_2. Such a dependence can significantly decrease the degree of parallelism exploited by the normalization algorithm.

Matching in path sequential systems proceeds as follows. Let $t = f(t_1, ..., t_k)$ be split among processors $P, P_1, ..., P_k$ such that P holds the root of t and the subterm t_i is in processor P_i. In our framework, P will initiate a match at the root and ask $P_1, ..., P_k$ to continue the match. P_i attempts to match t_i with the ith subterm of all lhs whose outermost symbol is f and reports back to P the subset of these lhs that matched. Now P decides whether there is a redex at t based on the results of the submatches thus returned. Note that copying of nodes to processor P from P_i's is completely avoided.

To avoid excessive communication, the matching process within a P_i must take place without any information about the outcomes of the submatches proceeding in other processors. If this happens then observe that communication takes place only between those processors across which the portion of the term to be matched is split. For matching within a P_i to proceed independently of other submatches we require that P_i be able to determine, based entirely on the information sent to it by its parent P, whether to regard t_i as a variable substitution or to head normalize it to match a nonvariable symbol of an lhs. Another outcome of this independence is a matching automaton whose size is linear in the size of the program. To illustrate all this, we first define *path sequential systems* as follows.

Definition 3 *A constructor system $\mathcal{R}$ is* path sequential *iff there are no two lhs l_1 and l_2, a path p and an integer i such that $pathstr(l_1, p) = pathstr(l_2, p)$, $l_1/p.i$ is a variable and $l_2/p.i$ is a nonvariable.*

Consider any prefix u that is not a redex and unifies with an lhs l. Then there must be a path p in it such that u/p is a variable and l/p is a nonvariable. By path sequentiality, all lhs that unify with u must also have a nonvariable at p and hence it is an index for u. Thus every prefix u has an index and so path sequential systems are also strongly sequential. The lhs in fig.1 are strongly sequential, but not path sequential since $pathstr(l_1, 1) = pathstr(l_2, 1) = f.1.a$, but $l_1/1.1$ is a variable whereas $l_2/1.1$ is a nonvariable. In contrast, the set $\{f(b(e), c(x)), f(a(x), c(y))\}$ of lhs is path sequential.

Since we can determine whether p is an index based only on the path string from root to the parent of the node at p, matching can be based on path strings. Therefore the matching automaton will contain one state corresponding to each prefix of a path string. We show that the number of such prefixes of an lhs l is *linear* in the number of nonvariables symbols in it. Let $t = f(t_1, ..., t_k)$ be a term with n nonvariable symbols and n_i is the number of nonvariable symbols in t_i. Also let $T(n)$ denote the number of such prefixes for an lhs with n nonvariable symbols, excluding the only prefix x of l that contains no nonvariables. A prefix of a path string in t is either f or is obtained by concatenating a path string of t_i with $f.i$ for any $1 \leq i \leq k$. Therefore

$$T(n) = 1 + T(n_1) + \cdots + T(n_k) \text{ where } \sum_{i=1}^{k} n_i = n - 1$$

with $T(1) = 1$. The solution to this recurrence is $T(n) = n$ and hence the automaton based on path strings is linear. In contrast, the automaton to drive the centralized strategy will have exponential number of states *even for path sequential systems.*

In summary path sequential systems provide an elegant framework for efficient lazy parallel evaluation of equational programs. They permit us to search and reduce needed redexes independently along different paths. More importantly, the size of the automaton (which is the size of compiled code) is linear in size of the rules. A sound and complete parallel normalization strategy based on path sequential systems and an algorithm to transform any strongly sequential system to an equivalent path sequential system are discussed in the following sections.

4 Parallel Normalization

We now present an algorithm to normalize a term (in parallel) with respect to a path sequential system and establish its soundness and completeness.

4.1 Normalizing Algorithm

For a set of lhs $\mathcal{L}$ and path p, let $nvsons(\mathcal{L}, p)$ denote the set of integers $\{i \mid \exists l \in \mathcal{L}\ root(l/p.i)$ is a nonvariable $\}$. Note that if $pathstr(l_1, p) = pathstr(l_2, p)$ then by definition of path sequentiality, $l_1/p.i$ is a nonvariable iff $l_2/p.i$ is a nonvariable.

The normalizing algorithm makes use of three procedures: $Normalize$, $Hnorm$ and $Match$. All these procedures take a term v, passed by reference, as an argument. $Normalize$ normalizes its argument. It makes use of $Hnorm$ to find the HNF v_h of v. Since the root symbol of v_h appears in its normal form, the normal form of v can be computed by normalizing the subterms rooted at the children of v_h.

$Hnorm$ uses the procedure $Match$ to check if there is any redex at the root of its argument v. If a redex is found then it is rewritten and this process is repeated until v is reduced to a term v' in HNF. We refer to v' as the "term returned" by $Hnorm$. The details of $Normalize$ are as follows:

procedure $Normalize(v)$ /* v is the term to be normalized */
1. $Hnorm(v)$;
2. $Normalize(u \mid u$ **is an immediate subterm of** $v)$

procedure $Hnorm(v)$ /* v is the term whose HNF is to be found */
3. /* If $root(v)$ is a constructor then we can return immediately. */
4. **while true do**
5.   **if** $Match(\Lambda, v, \mathcal{L}) \neq \phi$
6.   **then** rewrite the redex v
7.   **else return**; /* Now v must be in HNF */
8. **end**

function $Match(p, v, \mathcal{S})$ /* p is a path, v a term and $\mathcal{S}$ is a set of lhs. */
9. $\mathcal{S}' := \{l \in \mathcal{S} \mid root(l/p) = root(v)\}$
10. **if** $nvsons(\mathcal{S}', p) = \phi$ **then return** $\mathcal{S}'$
11. **else return** $\bigcap_{j \in nvsons(\mathcal{S}')} Match(p.j, Hnorm(v/j), \mathcal{S}')$.

Note that procedure $Match$ performs matching based on path strings. Let u be the innermost subterm above v where $Match(\Lambda, u, \mathcal{L})$ was invoked. Then the argument $\mathcal{S}$ in a call $Match(p, v, \mathcal{S})$ consists of all lhs whose path strings matched the path string from the root of u to the parent of v. $\mathcal{S}'$ computed at line 9 consists of lhs in $\mathcal{S}$ whose symbols at p match that at u/p and thus $\mathcal{S}' = \{l \in \mathcal{L} \mid pathstr(l, p) = pathstr(u, p)\}$. Since the symbols on the inside of all lhs are constructors, $Hnorm$ (which would reduce any term with a function symbol root into one with a constructor root whenever possible) is called at line 11 before $Match$ is called recursively. Note that these calls to $Hnorm$ implicitly change v to v' by reducing the subterms below v and we refer to v' as "the term returned" by $Match$ at line 11. The set returned by $Match(p, v, \mathcal{S})$ consists of all lhs $l \in \mathcal{S}$ such that l/p matches v'.

Note that in order to perform the rewrite at step 6, we need the substitutions for the variables in the lhs that matched. This can be passed back to the processor P holding the root of the match in the following way. Each processor holding a substitution communicates directly with P to pass a pointer to the root of the substitution. To do this, the identity of P is propagated downwards as a fourth argument to the recursive calls of $Match$.

There are two sources of parallelism in this algorithm. First, the normalization of the different u's at line 2 can proceed in parallel. Second, at line 11, $Hnorm(v/j)$ and the subsequent matching on the paths $p.j$, for different j's can be done in parallel. The parallelism in the latter case is more significant in practice, since each invocation of $Hnorm$ may in turn result in several parallel invocations of $Hnorm$ on subterms below and so on, thereby yielding (potentially) a large degree of parallelism. We illustrate this through an example.

Consider a binary tree whose leaves are records of the form $leaf(id, value)$ and whose internal nodes are of the form $tree(l, r)$ where l and r are the left and right subtrees respectively. We want to search for a record with a particular value field. Note that this must be done by an exhaustive search of the leaf records since the tree is not organized as a search tree. The following equational program performs such a search in parallel.

1. $search(tree(l, r), v) \rightarrow select(search(l, v), search(r, v))$
2. $search(leaf(id, val), v) \rightarrow if(equals(v, val), found(id), nil)$
3. $select(found(x), nil) \rightarrow found(x)$
4. $select(nil, found(y)) \rightarrow found(y)$
5. $select(nil, nil) \rightarrow nil$
6. $select(found(x), found(y)) \rightarrow found(x)$
7. $if(true, x, y) \rightarrow x$
8. $if(false, x, y) \rightarrow y$

Here, $equals$ is an equality function that returns $true$ or $false$.

Consider the normalization of the term $search(t_1, 2)$, where t_1 and its subterms $t_2, ..., t_7$ are shown in fig.3. In the figure, t and l stand for $tree$ and $leaf$ respectively. The first call to $Match$ in the first invocation of $Hnorm$ (i.e., line 5 of algorithm $Normalize$) returns with a match at the root, and hence the term is rewritten into $select(search(t_2, 2), search(t_5, 2))$. Once again $Match$ is called at line 5, which matches the root symbol $select$ with that of lhs 3,4,5 and 6. Now the parallel invocations of $Hnorm$ at 11 start off two parallel computations of $Hnorm(search(t_2, 2))$ and $Hnorm(search(t_5, 2))$. The calls of $Match$ made from each of these $Hnorm$'s succeed immediately, rewriting the term into $select(select(search(t_3, 2), search(t_4, 2)), select(search(t_6, 2), search(t_7, 2)))$. Now there will be four parallel computations of $Hnorm$ on the subterms $search(t_3, 2), ..., search(t_7, 2)$. Finally, the algorithm terminates with $found(6)$ as the normal form. If the input tree is of size n and is balanced then the normal form is obtained in $O(\log n)$ time.

4.2 Soundness and Completeness

We now establish the soundness and completeness of $Normalize$. First we need the following facts.

Fact 1 When $Match(p, v, \mathcal{S})$ is invoked (line 5 or 11), let u be the innermost subterm above v where $Match$ has been invoked with Λ as the first argument. Then $\mathcal{S}$ is the set of all lhs l such that $pathstr(u, p)$ and $pathstr(l, p)$ are identical except for $root(v)$.

This is easily proved by induction on length of p (and is omitted). Now note that $Match(p, v, \mathcal{S})$ is invoked at p only if some lhs in $l \in \mathcal{S}$ has a nonvariable at p. By path sequentiality and fact 1,

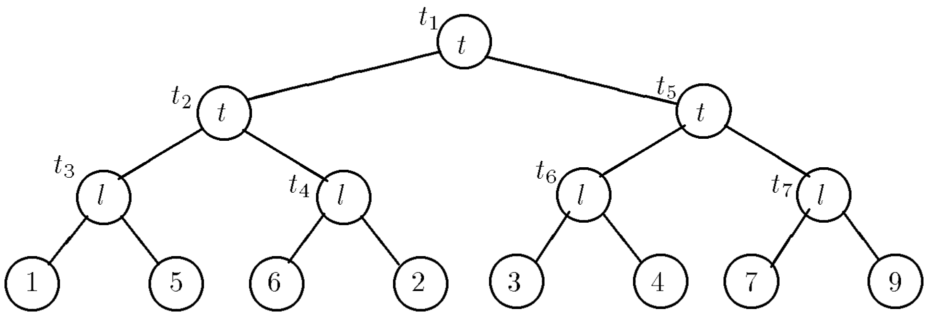

Figure 3: Input term for the *search* example

we have

Fact 2 All $l \in \mathcal{S}$ have a nonvariable term at p whenever $Match(p, v, \mathcal{S})$ is invoked.

This ensures that the comparison made at line 9 involves only nonvariable symbols. Now we are ready to prove correctness of $Match$.

Lemma 4 (Correctness of $Match$**)** *If* $Match(\Lambda, v, \mathcal{L})$ *returns a set* $\mathcal{S}$ *and a term* v' *then* $\mathcal{S} = \{l \in \mathcal{L} \mid u \geq l\}$.

Proof: Is immediate from the proof of the following stronger assertion:

If $Match(p, v, \mathcal{S})$ *returns with the set* $\mathcal{S}''$ *and a term* v' *then* $\mathcal{S}'' = \{l \in \mathcal{S} \mid v' \geq l/p\}$.

We prove this assertion by induction on the maximum height of l/p for any $l \in \mathcal{S}$. The base case is when this height is zero, in which case l/p must be a unary constructor for every $l \in \mathcal{S}$. In this case $\mathcal{S}'$ computed at line 9 is either empty or consists of all $l \in \mathcal{S}$ such that $v' = l/p$. In either case $\mathcal{S}'$ is the desired set and it is returned by $Match$ at line 10. For the induction step, note that $Match$ may either return from line 10 or line 11. In the former case, the proof is similar to the basis step. In the latter case, note that the maximum height of any $l/p.j$ for $l \in \mathcal{S}' \subseteq \mathcal{S}$ is less than the maximum height of any l/p for $l \in \mathcal{S}$. Therefore we can inductively assume that the invocations of $Match(p.j, Hnorm(v/j), \mathcal{S}')$ return with $\mathcal{S}_j = \{l \in \mathcal{S}' \mid v_j \geq l/p.j\}$ and the terms v_j. Now it is easy to see that their intersection yields $\mathcal{S}''$. ∎

Lemma 5 (Correctness of $Hnorm$**)** *If* $Hnorm(v)$ *returns with a term* v' *then* v' *is in HNF.*

Proof: Note that $Hnorm$ returns only when $Match$ returns ϕ at line 7. Therefore it suffices to show that whenever $Match(\Lambda, v, \mathcal{L})$ returns ϕ and a term v' then v' is in HNF. This is immediate from the proof of the following stronger assertion:

If $Match(p, v, \mathcal{S})$ *returns with a set* $\mathcal{S}''$ *and a term* v' *that is not a redex then* v' *cannot be reduced to a* $v'' \geq l/p$ *for any* $l \in \mathcal{S} - \mathcal{S}''$.

This is established by induction on height of v'. For the base case, height of v' is 0. Since v' is not a redex it must be a constructor and cannot be further reduced. Thus $\mathcal{S}'$ is the set of *all* lhs $l \in \mathcal{S}$ with $v' \geq l/p$ and $Match$ correctly returns this set at line 10. For the induction step there are three cases to consider:

<u>*Case 1.*</u> $Match$ returns from line 11: Let S_j be the set and v_j the term returned by a recursive call $Match(p.j, Hnorm(v/j), \mathcal{S}')$ at line 11. Then $\mathcal{S}'' = \bigcap \mathcal{S}_j$ and $v'/j = v_j$. Since the heights of v_j are less than that of v' we can inductively assume that v_j cannot be reduced to an instance of $l/p.j$ for any $l \in \mathcal{S}' - \mathcal{S}_j$. Now $root(v')$ can be either a constructor or a function symbol. In the former case it is clear that the lemma holds. In the latter case, $p = \Lambda$ and $\mathcal{S} = \mathcal{L}$ (or otherwise $\mathcal{S}'$

will be empty and $Match$ will return from line 10). Also note that $\mathcal{S}'' = \phi$ since v' is not a redex. This means that for every lhs l with the outermost symbol $root(v')$ there exists a j such that v_j cannot be reduced into an instance of l/j. Therefore v' cannot be reduced into an instance of any l and hence the lemma holds.

Case 2. $root(v')$ is constructor and $Match$ returns from line 10: Any v'' to which v' can be reduced has $root(v'') = root(v')$ and so $v'' \geq l/p$ holds for an $l \in \mathcal{S}$ only if $root(l/p) = root(v')$. All such lhs are in the set $\mathcal{S}'' = \mathcal{S}'$ and hence the lemma.

Case 3. $root(v')$ is a function symbol and $Match$ returns from line 10: First note that $v = v'$. In this case if $\mathcal{S}' \neq \phi$ then $v = v'$ will match a nonvariable subterm of some lhs l (by correctness of $Match$). This subterm must be l itself since $root(v)$ is a function symbol and so $v = v'$ will be a redex, which is a contradiction. Therefore $\mathcal{S}' = \phi$. Furthermore, this invocation of $Match$ must have been made from line 11 and *not* line 5, since in the latter case $p = \Lambda$ and $\mathcal{S} = \mathcal{L}$, which implies that $\mathcal{S}'$ cannot be empty when $root(v) \in \mathcal{F}$. Therefore v must have been returned by an invocation of $Hnorm(u)$ at line 11. This call of $Hnorm$ would have performed a series of invocations of $Match$ followed by rewrites (in the loop at lines 4–8) till a call $Match(\Lambda, u', \mathcal{L})$ returned ϕ and v. For this invocation of $Match$, $root(u') = root(v)$ is a function symbol and $\mathcal{S}'$ computed at line 9 will be nonempty. Therefore this invocation can return ϕ only from line 11. In such a case, by case 1 of this proof, $v = v'$ can never be reduced into a redex and so any v'' to which v' can be reduced will have $root(v'') = root(v') \in \mathcal{F}$ and so will not match l/p (Note $p \neq \Lambda$) for any $l \in \mathcal{S}$. ∎

Theorem 6 (Soundness) *$Normalize(t)$ terminates with the normal form of t.*

Proof: We need to show that if $Normalize(t)$ terminates with t_n then t_n is in NF. This is done by induction on height of t_n. The base case is when height of t_n is zero. Then $Normalize(t)$ calls $Hnorm$ which returns with t_n. By correctness of $Hnorm$, t_n is in HNF and hence in NF. For the induction step, note that $Normalize$ calls $Hnorm$ and then proceeds to normalize the immediate subterms of the term returned by it. The heights of the terms returned by these recursive invocations are less than that of t_n and so we can assume, by inductive hypothesis, that they are in NF. Then t_n will be in NF since it is in HNF and all its subterms are in NF. ∎

Theorem 7 (Completeness) *$Normalize(t)$ terminates whenever t has a normal form.*

Proof: Note that any procedure that repeatedly rewrites needed redexes is complete for constructor systems [HL79]. Therefore we need only show that $Normalize$ rewrites only needed redexes. To do this, we first show that $Hnorm(v)$ reduces only those redexes that will remain outermost in v irrespective of other reductions inside v. Observe that whenever $Normalize$ invokes $Hnorm(v)$, every subterm (of the term t being normalized) that is above v is in HNF. Therefore the redexes reduced by $Hnorm(v)$ will remain outermost in t irrespective of other reductions and hence are needed.

Now we need to show that any redex selected at line 6 by $Hnorm$ will remain outermost in v. This is implied by the following stronger assertion:

For every p such that $Hnorm(v/p)$ is invoked, no subterm v/q for $q < p$ can be reduced into a redex unless v/p is reduced into a term with a constructor root.

We prove this by induction on length of p. For the base case $p = \Lambda$ in which case there is no $q < p$ and hence the above assertion holds vacuously. For the induction step, $Hnorm(v/p)$ is called from line 11. It can be easily seen that there exists a longest path $q < p$ such that

1. $Hnorm(v/q)$ was invoked.
2. This resulted in a call of $Match$ at line 5.
3. This call of $Match$ resulted in further calls to $Match$ at line 11 until $Match(p', Hnorm(v/p), \mathcal{S}')$ was called.

By fact 1 and path sequentiality, every lhs that could potentially match at q has a nonvariable (constructor) at p and thus no redex can be found at q without reducing v/p to a term with constructor root. By inductive hypothesis, no redex can be found at any $r < q$ without reducing v/q into a term with constructor root. These two facts imply that no subterm at any $r < p$ can be reduced into a redex without reducing v/p to a term with constructor root. ∎

5 Parallelism and Complexity Issues

In this section we discuss the time complexity of *Normalize*, communication complexity of the matching process used by it and the parallelism it extracts.

5.1 Time Complexity

Our derivation of the time complexity of *Normalize* is based on *legal terms*. The normal forms of such terms will not contain function symbols. 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. Therefore we deal with legal terms only in our discussion of time complexity.

Observe that *Normalize* normalizes a term by repeatedly searching for redexes and replacing them. The search may often involve repeated traversals of the term and so it may appear that a lot of time is spent in this traversal/search rather than in reductions. However, we now show that the time spent by *Normalize* is bounded by the time required to perform the needed reductions. For showing this, we assume that the number of lhs with the same function symbol (typically 2–3) is smaller than the number of bits in a word (typically 32).

Observe that *Normalize* does not remember that a subterm is in HNF and so may repeatedly traverse that subterm to check if it is in HNF. This can be avoided by modifying *Hnorm* as follows:

```
procedure Hnorm(v)  /* v is the term whose HNF is to be found */
    if v is marked to be in HNF then return
    while true do
        if Match(Λ, v, L) ≠ φ
            then rewrite the redex v
        else mark v to be in HNF and return;
    end
```

Although *Normalize* does not do any wasteful reductions, the number of reductions performed by it may not be minimal when it uses tree representation of terms. For example, consider the rewriting system $\mathcal{R} = f(x) \rightarrow g(x,x)$ and the term $t = f(f(a))$. Although both $f(f(a)) \rightarrow f(g(a,a)) \rightarrow g(g(a,a),g(a,a))$ and $f(f(a)) \rightarrow g(f(a),f(a)) \rightarrow g(g(a,a),f(a)) \rightarrow g(g(a,a),g(a,a))$ reduce only needed redexes, the former performs only two reductions as against the three reductions performed by the latter. The reason for this is that the outer reduction $f(x) \rightarrow g(x,x)$ duplicates the inner redex in the latter case. We can avoid this by keeping the rhs as fully compact dags and thus sharing the multiple occurrences of these redexes. We make the following interesting observations about the length of reduction sequences using dags and trees.

Observation 1: If the rhs of the equations are such that their compact dag representation shares no function nodes, then there exist needed reduction sequences on tree representation that do only as much work as on the dag representation, but these sequences are not computable in general.

The idea is to perform the reductions inside out. This way the outer reductions cannot create multiple copies of inner redexes. Such a sequence may not be computable since it requires us to determine whether an inner redex is needed even before knowing what an outer redex will reduce to.

Observation 2: If the rhs are such that the compact dag versions share function nodes then there are some sequence of reductions for the dag version that do strictly less work than any tree version.

Consider the rewrite system $\mathcal{R} = \{f(x) \to g(h(x), h(x)), h(x) \to x\}$ and the term $t = f(a)$. If the rhs are represented as dags then t can be normalized in two steps. However, if they are represented as trees then every reduction sequence must reduce at least three redexes.

To handle dag representation of terms we modify $Normalize$ as follows. To ensure that shared subterms are normalized just once, every time $Normalize$ is invoked it checks if its argument v is marked as "visited", and if so returns immediately. Otherwise, it proceeds as before and finally, before returning from line 2, it marks its argument as having been "visited". Furthermore, we incorporate to have a reference count mechanism in order to properly reclaim shared structures that become garbage (i.e., can no longer be referenced) after a rewrite. Note that this adds an overhead that is proportional only to the size of the lhs l and rhs r involved. Finally, when there are more than one concurrent invocations of $Hnorm(v)$ on a shared subterm v, we first queue all these invocations. An invocation is allowed to proceed only after all invocations ahead of it in the queue are completed.

Theorem 8 *Let $t \to_{l_1} t_1 \to \cdots \to_{l_n} t_n$ be the sequence of needed reductions performed by $Normalize$, $E = \sum_{i=1}^{n}(|l_i| + |r_i|)$ and N be the size of the normal form. Then the normalization procedure executes in $O(N + E)$ time.*

Proof: We split the total time spent in three parts:

1. Time spent in $Normalize$: Note that it takes $O(1)$ time to invoke $Hnorm$. (The time spent inside the body of $Hnorm$ as a result of this call is accounted for in 2.) Therefore each invocation of normalize spends constant amount of time. Furthermore, it can be seen that there will be one invocation of $Normalize$ for each edge in the normal form and thus the time spent is bounded by $O(N)$.

2. Time spent in $Hnorm$, i.e., lines 3 – 8. If $Hnorm$ returns without performing any rewrites, then $O(1)$ time is spent inside $Hnorm$ and we "charge" this to the call of $Hnorm$ (made at line 2 or 11). Otherwise, it is clear that the cost is bounded by the time taken for rewrites performed at line 6 (excluding time spent inside calls of $Match$ as it is being accounted for in 3) and hence is bounded by $O(E)$.

3. Time spent inside $Match$. We claim that this is bounded by $O(E)$. We first show that each invocation of $Match$ spends constant amount of time. Observe that line 9 can be executed in $O(1)$ time using a matching automaton for path strings, wherein the state reached by a path string s stores the set of all lhs that have s in them. If $|nvsons(\mathcal{S}')| = k$ at 11 then it can be computed in time proportional to k and the set intersection operation can also be done in $O(k)$ time with the bit-vector representation. We distribute this cost as 1 unit each for the k recursive invocations of $Match$ on the k subterms. Thus each execution of body of $Match$ takes $O(1)$ time.

For an invocation of $Match(p, u', \mathcal{S})$, let u be the innermost subterm above u' with $root(u) \in \mathcal{F}$ such that the $Match$ was invoked on u and this resulted in recursive invocations of $Match$ on subterms of u till $Match(p, u', \mathcal{S})$ was called. If the match at u succeeded with a redex for l then a reduction will be performed, whose cost $|l| + |r|$ is accounted for in E. Note that $O(1)$ cost of $Match(p, u', \mathcal{S})$ is included in $|l|$. If the match at u fails then there must exist a path q such that $u = t/q$ and all subterms t/r for $r \leq q$ are in HNF. Otherwise, by proof of completeness, $Hnorm(u)$ and consequently $Match(\Lambda, u, \mathcal{L})$ will never have been invoked. Thus the function symbol at the root of u will appear in the normal form and hence the input term is not legal. ∎

We remark that any algorithm (based on an evaluation model wherein we must spend $O(|l|+|r|)$ time to perform a reduction using a rule $l \rightarrow r$) that normalizes legal terms must spend at least $O(N+E)$ time. This is because the redexes reduced by $Normalize$ must all be reduced by any other algorithm and furthermore, it takes at least $O(N)$ time to check if the output term is in normal form. Thus $Normalize$ is optimal for normalizing legal terms.

5.2 Communication Complexity of $Match$

As mentioned earlier (see section 3), processors need to communicate during the match phase if parts of a subterm being matched are split across different processors. If these split matches are handled naively (e.g., by first copying all the nodes to be matched to the processor holding the node at which the match is initiated and then performing the match locally) then it can lead to a large volume of communication. Ideally, we would like handle each split using $O(1)$ communication steps and this in fact is achieved by $Normalize$. The invocation of $Match$ at line 11 results in one communication step if $root(v/j)$ is in a processor different from the one in which $root(v)$ is. The return value of the recursive call is transmitted back to the processor holding $root(v)$ in another communication step.

Finally, note that the communication required to pass the substitutions to the processor P holding the root of the match is also minimal: one communication step for each substitution that resides on a processor different from P.

5.3 Enhancing Parallelism

We now argue that the degree of parallelism achieved by our algorithm cannot be improved by any other lazy normalization algorithm that is independent of the rhs of the rules. Observe that an invocation $Match(\Lambda, v, \mathcal{L})$ results in a downward traversal of all paths from the root of v. This traversal halts only on encountering a node that corresponds to a variable substitution for all potentially matching lhs. Therefore the parallelism can be enhanced only if the algorithm searches for and replaces redexes inside variable substitutions. But this cannot be done by any lazy algorithm that does not inspect the rhs since it is possible that the corresponding variable does not appear on the rhs. In such a case, all the redexes below this variable substitution are discarded and are therefore not needed.

However, by analyzing the rhs we can determine in many cases whether a redex below a variable substitution is needed. This information can then be used by $Normalize$ to enhance parallelism. The details of such an analysis appear in [SSR90].

6 Transformation

In this section we present algorithms for transforming a strongly sequential constructor system into a path sequential system. Our algorithm reports failure when the input system is not strongly sequential. We illustrate the algorithm with an example and prove its correctness.

Before we proceed with a description of our algorithm, we need the following definitions. We say there is a *block* in l at $p.i$, if there exists an $l' \in \mathcal{L}$ such that $pathstr(l', p) = pathstr(l, p)$ and $l'/p.i$ is a variable (here p is a path and i is an integer and l and l' are not necessarily different). The node at p is known as a blocking node. A path p is a block free path in l if there is no block in l at any $q \leq p$. If, in addition, there is a block in l at $p.i$ for $1 \leq i \leq rank(root(l/p))$ then we refer to p as a *maximal block-free* path. In fig.1, nodes 4 and 5 are blocking nodes in l_1 and l_2 and so paths 1 and 2 are maximal block-free paths.

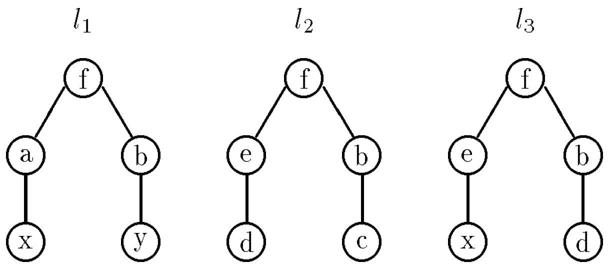

Figure 4

$Top(\mathcal{L}, l)$ is the structure obtained from l by deleting all subtrees rooted at blocking nodes in l. $Roof(\mathcal{L}, l)$ is the structure obtained by replacing each subtree rooted at a blocking node by a new variable. We use $top(l)$ and $roof(l)$ where $\mathcal{L}$ is understood. In fig.1, $top(l_1) = f(a, c)$, $top(l_2) = f(a, d)$ and $roof(l_1) = f(a(x), c(y))$.

Note that in a constructor system, *strong sequentiality* or *path sequentiality* is determined independently for each set of rules with the same outermost symbol. Hence we assume without any loss of generality that all the lhs in $\mathcal{L}$ have the same root symbol f. The bulk of the transformation work is done by two procedures – *BuildTree* and *Transform*. *BuildTree* constructs a tree which is essentially an automaton to recognize the maximal block-free path strings in $\mathcal{L}$. (Note that if $\mathcal{L}$ is already path sequential, then *BuildTree* constructs a matching automaton for $\mathcal{L}$.) *Transform* makes use of this tree to replace $\mathcal{R}$ by $\mathcal{R}^N = \mathcal{R}' \cup (\bigcup_i \mathcal{R}_i)$, where $\mathcal{R}'$ is path sequential, and the $\mathcal{R}_i$'s need to be processed recursively by *Transform*. Before proceeding with the details of the algorithm we first illustrate the transformation process on the following example. We will show the effect of *BuildTree* and *Transform* after each step of the transformation.

Let $\mathcal{R} = \{[f(a(x), b(y)) \to r_1(x, y)], [f(e(d), b(c)) \to r_2], [f(e(x), b(d)) \to r_3(x)]\}$. The lhs of these rules are illustrated in fig.4. Invocation of *BuildTree* on these lhs results in the tree shown in fig.5. Observe that there is a block at 1.1 in l_2 and l_3. (To see this, put $l = l_2$ and $l' = l_3$ in the definition of block.) The same block exists in l_1 (when $l = l' = l_1$). Similarly, there is a block at 2.1 for all three lhs. Thus, the maximal block-free path strings are $f.1.a$, $f.1.e$ and $f.2.b$.

We now show how *Transform* generates a new set of rules. Based on these blocks, the *top*'s of lhs l_1, l_2 and l_3 are $f(a, b)$, $f(e, b)$ and $f(e, b)$ respectively. We use a new function symbol g_1 to abbreviate $top(l_1)$ and g_2 to abbreviate $top(l_2)$ and $top(l_3)$. Next we introduce the following new equations in place of those in $\mathcal{R}$:

$$f(a(x), b(y)) \to g_1(x, y) \qquad \cdots (1)$$
$$g_1(x, y) \to r_1(x, y) \qquad \cdots (2)$$
$$f(e(x), b(y)) \to g_2(x, y) \qquad \cdots (3)$$
$$g_2(d, c) \to r_2 \qquad \cdots (4)$$
$$g_2(x, d) \to r_3(x) \qquad \cdots (5)$$

Note that in rule (3) the lhs is the same as $roof(l_2)$ (or $roof(l_3)$), and the rhs is obtained by making each variable in $roof(l_2)$ an argument to g_2. The lhs of (4) and (5) are obtained by making each blocking node in lhs l_2(or l_3) an argument to g_2.

Since there is only one rule with g_1 on lhs it is trivially path sequential. However, the rules (4) and (5) defining g_2 need to be processed further. *BuildTree* is called again on the lhs of (4) and (5), resulting in the tree shown in fig.6. Based on this, *Transform* generates the following rules to replace (4) and (5):

$$g_2(x, c) \to g_3(x) \qquad \cdots (6)$$
$$g_3(d) \to r_2 \qquad \cdots (7)$$
$$g_2(x, d) \to g_4(x) \qquad \cdots (8)$$
$$g_4(x) \to r_3(x) \qquad \cdots (9)$$

Note that these rules are again path sequential. Thus, the transformed system consists of rules (1),

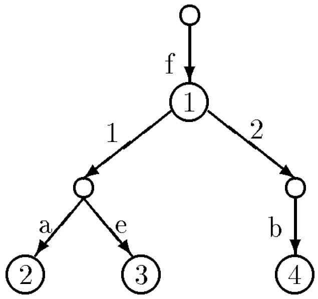

Figure 5

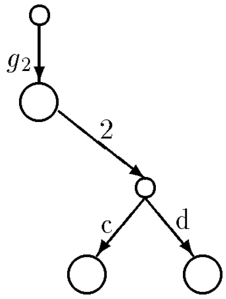

Figure 6

(2), (3), (6), (7), (8) and (9).

We now provide the algorithmic details of the process described above. We begin with $BuildTree$.

procedure $BuildTree(\mathcal{S}, p, v)$
  $\mathcal{S}$ is a list of lhs
  p is a path
  v is a dummy node in the tree. This invocation of $BuildTree$
    constructs the subtree rooted at v
1.   Partition $\mathcal{S}$ into
    $\mathcal{S}_x := \{l \in \mathcal{S} \mid l/p$ is a variable$\}$ and
    $\mathcal{S}_c := \{l \in \mathcal{S} \mid l/p = c\}$ for each $c \in \Sigma$
2.   If $\mathcal{S}_x \neq \phi$ then /* there is a block at p for every lhs in $\mathcal{S}$ */
3.   return(null)
  else
4.   for each $\mathcal{S}_c$ do
5.   create a new node v_c /* The v_c's are *state nodes* or *states* */
      create an edge from v to v_c, labelled c
      $poss(v_c) := \mathcal{S}_c$ /* $poss(v_c)$ is the set of lhs such that */
        /* $pathstr(l, p)$ = path in tree from root to v_c */
6.   for $i := 1$ to $rank(c)$ do
7.   create a new node v_{ci} /* The v_{ci}'s are the *dummy nodes* */
        create an edge from v_c to v_{ci}, labelled i
8. $BuildTree(\mathcal{S}_c, p.i, v_{ci})$
      end
    end
  end
  return(v)
end

On invoking $BuildTree$ on the rules in fig.4, the tree shown in fig.5 is obtained. The possibility sets associated with state nodes are: $poss(1) = \{l_1, l_2, l_3\}$, $poss(2) = \{l_1\}$, $poss(3) = \{l_2, l_3\}$ and $poss(4) = \{l_1, l_2, l_3\}$.

T refers to the tree constructed by this algorithm. For a state node v in T, define $trace(v)$ to be the sequence of labels on the edges from the root to v. The correctness of our algorithm depends on the following properties of T.

Proposition 9 *For each state v in T, $poss(v)$ is precisely the set*
$\{l \in \mathcal{L} \mid \exists p\; pathstr(l, p) = trace(v)\}$

Proof: by induction on length of $trace(v)$. ∎

Proposition 10 *For any state v in T, let v' be any child of v. For any two distinct children v_1 and v_2 of v', $poss(v_1) \cap poss(v_2) = \phi$. If v' has children $v_1, \ldots, v_k$, $poss(v) = poss(v_1) \cup \cdots \cup poss(v_k)$.*

Proof: Obvious, since the $\mathcal{S}_c$'s are distinct and $\bigcup_c \mathcal{S}_c = \mathcal{S}$. ∎

Lemma 11 *If p is a block-free path in l then there is a state node v in T such that $trace(v) = pathstr(l, p)$.*

Proof: by induction on length of p. For the basis $p = \Lambda$, in which case it is easy to see that the proposition holds. For the induction step, let $p = q.i$. Since q is also block-free, by inductive hypothesis, a state v_1 such that $trace(v_1) = pathstr(l, q)$ will be created at line 5. Consider the invocation $BuildTree(poss(v_1), p, v_{1i})$ (v_{1i} is the ith child node of v_1) from line 8. If there is no block in l at p then none of the lhs l' with $pathstr(l, q) = pathstr(l', q)$ will have a variable at p. By proposition 9, $poss(v_1)$ is precisely this set and so $\mathcal{S}_x$ computed at line 1 will be empty. Therefore the state v with $pathstr(l, p) = trace(v)$ is created at line 5. ∎

Note that in our transformation process we have to identify the *top*'s of lhs from T. This is done using the algorithm $Prune$ below.

```
procedure Prune(v, l)
        v is a dummy node in the tree T
        l is an lhs in ℒ
        for each child v_c of v do
            if l ∉ poss(v_c)
                then remove the edge from v to v_c
            else for each child v_ci of v_c do
                Prune(v_ci, l)
        end
        return(v)
end
```

Invoking $Prune$ on the tree shown in fig.5 with respect to lhs l_1 and l_2, results in the trees shown in figs.7 and 8. The corresponding *top*'s are shown in figs.9 and 10.

Now we state and prove some important properties of $Prune$ and show how $top(l)$ can be constructed from $Prune(T, l)$. In the following T_l refers to $Prune(T, l)$.

Proposition 12 *T_l contains exactly the states v in T such that $poss(v)$ includes l.*

Proof: Immediate from the definition of $Prune$ and proposition 10. ∎

It is clear from this and lemma 11 that T_l contains all and only maximal block-free paths in l. By definition of $top(\mathcal{L}, l)$, it follows that it also contains all and only maximal block-free paths of l. This establishes an obvious correspondence between the state nodes in T_l and $top(\mathcal{L}, l)$. An algorithm to compute *top* from $Prune$ would first merge each dummy node with its parent. This removes all integer labelled edges in T_l. Now the label on each edge is moved to the state node it points to. Dropping the root from this structure we get $top(\mathcal{L}, l)$. This procedure is illustrated in figs. 9 and 10. *Roof* can be obtained from *top* by adding variables at appropriate places.

In the following let $\mathcal{V}_l = \{v \mid v \text{ is a leaf of } Prune(T, l)\}$.

Lemma 13 $l_1 \in \bigcap_{v \in \mathcal{V}_l} poss(v)$ *iff* $Prune(T, l) = Prune(T, l_1)$

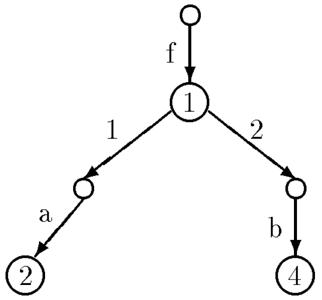

Figure 7

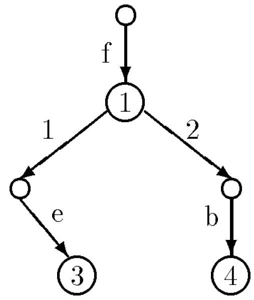

Figure 8

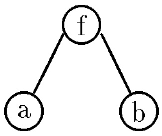

Figure 9

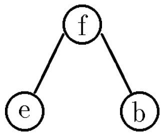

Figure 10

Proof: The if part is straight forward and is omitted. For the only if part, observe that (by proposition 10) $l_1 \in poss(v)$ for all ancestors of the leaves of $Prune(T, l)$. Thus, all the states v in $Prune(T, l)$ contain l_1 in $poss(v)$. From proposition 10, for every state v in T such that $l \in poss(v)$, exactly one of the children v' of each child v_i of v has $l \in poss(v')$. Since v' appears in $Prune(T, l)$, $l_1 \in poss(v')$. By proposition 10, no other child v'' of v_i can have $l_1 \in poss(v'')$. This implies that all v in T such that $l_1 \in poss(v)$ are in $Prune(T, l)$. Hence $Prune(T, l) = Prune(T, l_1)$ by proposition 12. ∎

We now present details of procedure $Transform$ that generates a new set of rules after a single step of transformation.

procedure $Transform(\mathcal{L})$
  $\mathcal{L}$ is the set of lhs of the input equations
  $\mathcal{L}' := \phi$; $i := 1$
1. $T := BuildTree(\mathcal{L}, \Lambda, root)$
2. while $\mathcal{L} \neq \phi$ do
    $\mathcal{L}_i := \phi$
3. choose an $l \in \mathcal{L}$
4. Let e be the equation $[roof(\mathcal{L}, l) \rightarrow g(\overline{x})]$ /* e is the equation *generating* g */
      where $\overline{x}$ is the list of variables in $roof(\mathcal{L}, l)$
        and g is a new functor symbol.
5. $\mathcal{L}' := \mathcal{L}' \cup e$
6. $\mathcal{L}_g = \bigcap_{v \in \mathcal{V}_l} poss(v)$
7. for each $l_1 \in \mathcal{L}_g$ do
8. Let t be the term obtained by rewriting[4] l_1 using e
9. $\mathcal{L}_i := \mathcal{L}_i \cup [t \rightarrow r_1]$, where $[l_1 \rightarrow r_1]$ is the original rule.
    end /* $[t \rightarrow r_1]$ is an equation *defining* g */
10. $\mathcal{L} := \mathcal{L} - \mathcal{L}_g$; i := i+1
  end /* while */
11. if $|\mathcal{L}'| = 1$ return($failed$)
end

Now we establish that $\mathcal{R}$ and $\mathcal{R}^N$ are equivalent. In the succeeding lemmas we show that every reduction sequence in $\mathcal{R}$ can be simulated by a corresponding sequence in $\mathcal{R}^N$ and vice versa. Note that the alphabet of $\mathcal{R}^N$ contains some g's that are not present in the alphabet of $\mathcal{R}$ and hence $\mathcal{R}$ cannot directly simulate $\mathcal{R}^N$. Therefore we define a function h that maps a term with g's into one that does not contain g's. Under this mapping we show that the two systems can simulate each other.

Let Σ^N and Σ be the alphabets of $\mathcal{R}^N$ and $\mathcal{R}$ and $l_g \to g(x_1, ..., x_{rank(g)})$ be the rule generating g for every $g \in (\Sigma^N - \Sigma)$. We denote by $t[t_1 \leftarrow t_2]$ the term obtained from t by replacing one of its subterms t_1 by t_2. We define a function h that maps a Σ^N-term to a Σ-term as follows.

$h(g(t_1, ..., t_{rank(g)})) = g(h(t_1), ..., h(t_{rank(g)})), \quad g \in \Sigma$

$h(g(t_1, ..., t_{rank(g)})) = l_g[x_1 \leftarrow h(t_1), ..., x_{rank(g)} \leftarrow h(t_{rank(g)})], \quad g \in (\Sigma^N - \Sigma)$

Note that the equation generating g does not contain g in its lhs and hence $h(t)$ is indeed a Σ-term. Intuitively, h "replaces" each g by the *top* it corresponds to. We make the following observation about h.

Fact 1: $h(t[u \leftarrow v]) = h(t)[h(u) \leftarrow h(v)]$

Lemma 14 *If* $t_1 \to^* t_2$ *in* $\mathcal{R}^N$*, then* $h(t_1) \to^* h(t_2)$ *in* $\mathcal{R}$.

Proof: By induction on the length of the reduction sequence $t_1 \to \cdots \to t_2$. The base case is for a sequence of length zero, whence $t_1 = t_2$ and $h(t_1) = h(t_2)$ and thus holds. Suppose $t_1 \to^* t'$ in n steps and $t' \to_{l''}^p t_2$. By induction hypothesis $h(t_1) \to^* h(t')$. There are two cases to consider:

1. $l'' \in \mathcal{L}'$: the rule used in reduction is of the form $l_g \to g(x_1, ..., x_{rank(g)})$. From the definition of h, it is clear that $h(t') = h(t_2)$ and thus the lemma holds.

2. $l'' \in \mathcal{L}_i$: In this case the rule used in reduction is of the form $g(u_1, ..., u_{rank(g)}) \to r$. Let $l \to r$ be the rule in $\mathcal{L}$ such that $l \to_{l_g} g(u_1, ..., u_{rank(g)})$. Let v' be the subterm of t' at p and v_2 be the corresponding subterm of t_2. By fact 1 it suffices to show that $h(v') \to h(v_2)$ in $\mathcal{R}$. It is clear by definition of h that $h(v') = l\sigma$. Thus $h(v') \to r\sigma = h(v_2)$ and this completes the proof. ∎

Lemma 15 *If* $t_1 \to^* t_2$ *in* $\mathcal{R}$*, then* $t_1 \to^* t_2$ *in* $\mathcal{R}^N$.

Proof: The proof is again by induction on the length of the reduction sequence in $\mathcal{R}$. The base case is once again for sequence length of zero whence it holds vacuously. For the induction step, assume that the lemma holds whenever $t_1 \to^* t'$ in $\mathcal{R}$ in n steps or less. Let $t' \to_l^p t_2$ in $\mathcal{R}$ and $[l' \to g(x_1, \ldots, x_{rank(g)})]$ and $[l_g \to r]$ be the rules (in $\mathcal{R}^N$) generated from $l \to r$. Since $l = l'\beta'$ we have $l\beta = l'\beta'\beta$. Hence t' can be reduced to t_2 in $\mathcal{R}^N$ by replacing $l'\beta'\beta$ by $g(x_1, \ldots, x_k)\beta'\beta = l_g\beta$. This can be further reduced by replacing $l_g\beta$ by $r\beta$, obtaining t_2. ∎

Theorem 16 $\mathcal{R}^N$ *is equivalent to* $\mathcal{R}$.

Proof: Straight forward from lemmas 6 and 7. ∎

Lemma 17 $\mathcal{R}'$ *is path sequential.*

Proof: Assume to the contrary that $\mathcal{R}'$ is not path sequential. Then, by definition there exists an l_1' and l_2' in $\mathcal{R}'$ and a path p such that $pathstr(l_1', p) = pathstr(l_2', p)$, and there is an i such that $l_1'/p.i$ is a variable and $l_2'/p.i$ is a nonvariable. From the fact that $pathstr(l_1', p) = pathstr(l_2', p)$ and proposition 9, it follows that there is a single state node v in T corresponding to l_1'/p and l_2'/p. Since $l_1'/p.i$ is a variable v cannot have an ith child. If $l_2'/p.i$ were to be a nonvariable then v must have to have an ith child – a contradiction. Therefore $\mathcal{R}'$ is path sequential. ∎

[4] In rewriting, the variables in l_1 are treated like constants

Lemma 18 *If $\mathcal{R}$ is strongly sequential then each of the $\mathcal{R}_i$'s is strongly sequential.*

Proof: Consider any constructor prefix t' that unifies with all and only lhs in a nonempty subset $\mathcal{L}_{it}$ of $\mathcal{L}_i$. Let $t = h(t')$ and $\mathcal{L}_t = \{h(l) \mid l \in \mathcal{L}_{it}\}$. It is clear that t unifies with all lhs in $\mathcal{L}_t$. Since $\mathcal{R}$ is strongly sequential, t had an index p. Let $t/p = x$. Then it is easy to see that the path q such that $t'/q = x$ is an index for t'. Therefore every such prefix t' has an index and hence the $\mathcal{R}_i$'s are strongly sequential. ∎

Lemma 19 *If the algorithm $Transform$ fails then the input system $\mathcal{R}$ is not strongly sequential.*

Proof: When $Transform$ reports failure at step 11, $|\mathcal{L}'| = 1$. Let t be the only lhs in $\mathcal{L}'$. Note that all lhs in $\mathcal{L}$ unify with t since it is the roof of all rules in $\mathcal{L}$. Now suppose that t has an index $p.i$. Then $pathstr(l, p)$ for all $l \in \mathcal{L}$ are identical and all of them have a nonvariable at $p.i$. Thus $p.i$ is a block-free path. By lemma 11 there must be a node in the tree constructed by $BuildTree$ that corresponds to $pathstr(l, p.i)$ and thus $root(l/p.i)$ will be a part of $t = top(l)$. This is a contradiction so t does not have an index, which means that $\mathcal{R}$ is not strongly sequential. ∎

The following procedure operates recursively on the $\mathcal{R}_i$'s to construct the equivalent path sequential system.

```
procedure PathSeq(R)
      R is the set of input equations
      if [Transform(R) ≠ failed] then
            for each system R_i generated by Transform do
                  if |R_i| > 1 then PathSeq(R_i)
      end
end
```

Theorem 20 *$PathSeq(\mathcal{R})$ returns a new path sequential system if $\mathcal{R}$ is strongly sequential.*

Proof: The algorithm $Transform$ returns sets of equations $\mathcal{R}'$ that is path sequential and $\mathcal{R}_i$'s that are strongly sequential. If any $\mathcal{R}_i$ contains just one equation then it is path sequential. Otherwise $Transform$ is applied on $\mathcal{R}_i$. Hence when $PathSeq$ terminates it has converted the original system into a path sequential system. To see that $PathSeq$ terminates, note that the number of $\mathcal{R}_i$'s is more than 1 (or else $Transform$ reports failure) so $|\mathcal{R}_i| < |\mathcal{R}|$. Thus after each call of $Transform$ we are left with smaller sets of equations that need to be processed. Since the processing stops when $\mathcal{R}_i = 1$, this process terminates. ∎

7 Complexity Issues of Transformation

In this section we present tight bounds on the size (i.e., the number of rules) of the transformed system in terms of the size of the original system, and the time complexity of the transformation algorithm.

7.1 Size Bounds for Transformed System

Let $|\mathcal{R}'| = k$ after one step of transformation. There are exactly k new function symbols introduced and let $|\mathcal{R}_1| = n_1, \ldots, |\mathcal{R}_k| = n_k$. It is clear that $n_1 + \cdots + n_k = n$. Representing the number of rules in the final system by $N(n)$, we have:

$$N(n) = k + \sum_{i=1}^{k} N(n_i) \qquad \text{such that} \quad \sum_{i=1}^{k} n_i = n$$

We will prove by induction that $N(n) \leq 3n-2$. It is clear that this holds for the base case $n=1$, since no further transformation is attempted by $PathSeq$ at step 3. For the induction step, since all n_i's are less than n, we can assume $N(n_i) \leq 3n_i - 2$.

$$N(n) \leq k + \sum_{i=1}^{k}(3n_i - 2) \qquad \textit{such that } \sum_{i=1}^{k} n_i = n$$

$$N(n) \leq k + 3n - 2k = 3n - k$$

By step 11 of $Transform$, $k > 1$, thus establishing the upper bound.

Now we show that this is a tight bound, in the sense that there exist systems of equations that expand to size $3n-2$ when transformed. Assume inductively that $N(n) \geq 3n-2$. This obviously holds for $n=1$. The worst case of $3n-2$ occurs when $k=2$, $n_1=1$ and $n_2=n-1$. In this case,

$$N(n) \geq 2 + N(1) + N(n-1) = 2 + 1 + 3n - 5 = 3n - 2$$

It remains to show that such a case actually occurs. An example of this worst case behavior is given below:

$$\mathcal{L} = \{f(a,b,x), f(b,x,t_1), \ldots, f(b,x,t_{n-1}))\}$$

where

$$t_1 = e(a,b,x),\ t_2 = e(b,x,t'_2), \ldots, t_{n-1} = e(b,x,t'_{n-1})$$

and so on. After one step of transformation,

$$\mathcal{L}' = \{f(a,x,y), f(b,x,y)\}$$

$$\mathcal{L}_1 = \{g_1(b,x)\}$$

$$\mathcal{L}_2 = \{g_2(x, e(a,b,y)), g_2(x, e(b,y,t'_2)), \ldots, g_2(x, e(b,y,t'_{n-1}))\}$$

It is easy to see that $\mathcal{L}_2$ is like $\mathcal{L}$, and hence in the recursive invocations of $Transform$, the same splitting of groups takes place. This establishes that in the worst case the transformed system can have as many as $3n-2$ rules.

7.2 Time Complexity

Let s be the sum of the sizes (i.e., the number of symbols) of all the lhs in $\mathcal{L}$ and $|\mathcal{L}| = n$. It is clear that $BuildTree$ looks at each node in every lhs in $\mathcal{L}$ at most once. All the operations involving a node take constant time. Hence $BuildTree$ takes $O(s)$ time. With a slight modification $Prune$ can be made to generate the *tops* in $O(s)$ time. Since $|roof(l)| < |l|$ and *top*(l) is already computed, computing the $roof$s take at most $O(s)$ time. The rewriting at step 8 in $Transform$ takes time proportional to the size of the rule being rewritten, and hence takes at most $O(s)$ time. Thus the total time taken at step 1 of $PathSeq$ is $O(s)$. The recurrence relation for the time taken by $PathSeq$ is therefore:

$$T(n,s) = c.s + \sum_{i=1}^{k} T(n_i, s_i)$$

where c is a constant. Noting that $s_i \leq s$ for all i, we get

$$T(n,s) \leq c.s + \sum_{i=1}^{k} T(n_i, s)$$

whose solution is $O(ns)$.

It can be shown that this bound of $O(ns)$ cannot be improved, as long as the variables are explicitly represented. The worst case example can be obtained by modifying the previous example slightly. Consider

$$\mathcal{L} = \{f(x_1, a, b, x), f(x_1, b, x, t_1), \ldots, f(x_1, b, x, t_{n-1}))\}$$

where

$$t_1 = e(x_2, a, b, x),\ t_2 = e(x_2, b, x, t_2'), \ldots, t_{n-1} = e(x_2, b, x, t_{n-1}')$$

and so on. It can be verified that the transformation process takes $O(ns)$ time.

8 Implications of Transformation

We now discuss some consequences of the transformation algorithm.

8.1 Equivalence of Notions of Sequentiality

Our transformation establishes the equivalence of the three notions of sequentiality discussed so far, viz., strong sequentiality, strong left-sequentiality and path sequentiality. Strong sequentiality is based on the more abstract notion of a sequential predicate and requires that we be able to identify an index for every prefix that is not a redex. The latter two notions further strengthen this requirement so that we are able to identify indexes while following a particular traversal order. This raises the natural question of whether there are other notions of sequentiality that can be based on different orders of traversal; and if so, whether they are all equivalent under our transformation. To answer this question, we have to first formalize what we mean by a notion of sequentiality based on a traversal order. Observe that strong left-sequentiality is based on the ability to perform lazy pattern matching (i.e., inspect exactly the nonvariable positions of the lhs for which a match is identified) in a preorder traversal. Strong sequentiality is based on traversing the term in a way that is best suited for lazy pattern matching and is determined by the set of lhs. Path sequentiality is based on the ability to accomplish the same by independent traversal of paths. Therefore it appears that we can base different notions of sequentiality on the basis of different traversals.

Pattern matching using one order of traversal may be superior to another order for several reasons. For instance, preorder traversal is simpler and potentially more efficient than the arbitrary order of traversal required for matching in strongly sequential systems; independent traversal of paths is more suited for parallel evaluation. Therefore, in general, a particular traversal order may be preferable over others for various reasons and so one may like to have a lazy normalization algorithm based on any such order of traversal. In section 8.1.1, we show that a notion of sequentiality can be based on any order of top-down[5] traversal. Informally, for a traversal order $\mathcal{A}$, we say that an equational program is $\mathcal{A}$-sequential if for each lhs in the program, $\mathcal{A}$-traversal visits all nonvariable nodes before all the variable nodes.

Having shown that there can be different notions of sequentiality based on different orders of traversal, a natural question is on the relative expressive power of classes of programs that observe each notion of sequentiality. In section 8.1.3 we show that *all such notions of sequentiality are equivalent* under the syntactic transformation presented before and thus all such classes are equally expressive. We first show that any equational program that observes any such notion of sequentiality is also strongly sequential. Next we show that path sequential systems are a subclass of systems that observe any one notion of sequentiality (again, say $\mathcal{A}$-sequentiality). This establishes the inclusion relationships among various notions of sequentiality as shown in figure 11. Since our transformation

[5] Bottom-up traversals are clearly inappropriate here since we need to identify an instance of an lhs by examining only the nonvariable symbols in the lhs and not any part of the term below variable substitutions.

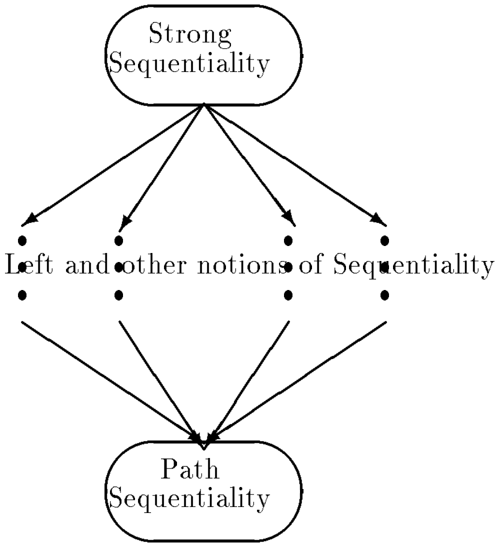

Figure 11: Inclusion relationships among notions of sequentiality.

procedure can transform any strongly sequential system into an equivalent path sequential system, these inclusion relationships imply that any two notions of sequentiality are equivalent.

Our transformation algorithm suggests that we can base an implementation on any notion of sequentiality that is sufficiently inclusive, (i.e., most programs observe this notion of sequentiality) yet enables a simpler and more efficient lazy evaluation algorithm than the Huet-Levy algorithm for strongly sequential systems. However, we do not have to restrict programs to observe this notion of sequentiality: all strongly sequential programs can be handled through our transformation. The same can be done with existing implementations such as O'Donnell's (which is based on strong left-sequentiality), thereby making the strong left-sequentiality restriction transparent to the user.

8.1.1 Formalizing Traversal Orders

The different notions of sequentiality exhibit inclusion relationships as shown in figure 11. To establish this, we first need to formally define what a top-down traversal is. Observe that a traversal visits nodes in a term successively and thus converts a term into a string representation. In order to maintain a unique correspondence between the term and the string obtained, we will specify not only the symbols seen as part of the string, but also the positions at which each symbol is inspected. If there is a variable at any position visited then it is denoted by the special symbol $*$ rather than the name of the variable. (This is because in linear pattern matching, variables act only as place holders and their names do not have any significance.) For instance, a preorder traversal of the term $f(a(c,d),b(x))$ converts it into a string $(\Lambda,f)(1,a)(1.1,c)(1.2,d)(2,b)(2.1,*)$. In order to allow independent (and simultaneous) traversal of different parts of a term, we relax the requirement that a traversal convert a term to a single string and allow a set of strings. Thus independent traversal of paths will produce the three strings $(\Lambda,f)(1,a)(1.1,c)$, $(\Lambda,f)(1,a)(1.2,d)$ and $(\Lambda,f)(2,b)(2.1,*)$ from the above term.

Although every traversal converts a term into a string as shown above, not every conversion of terms to strings corresponds to a traversal. Hence we need to impose some restrictions on conversion of terms into (set of) strings in order to identify which conversions correspond to traversals. Our restrictions are motivated by the need to construct a deterministic matching automaton to perform pattern-matching using a specified traversal order.

For a top-down traversal, we first require that all ancestors of a node be visited before the node itself. In particular, a conversion is said to be top-down iff in every string in the conversion, all ancestor nodes occur before descendent nodes. Secondly we require that there be some "consistency" or determinism in the conversion of terms to strings. For instance, if the term $f(a, b)$ converts to $(\Lambda, f)(1, a)(2, b)$ and the term $f(c, d)$ converts to $(\Lambda, f)(2, d)(1, c)$ then this does not correspond to any traversal order. This is because the root symbol in both terms are the same, yet the positions visited after f are different in the two terms. This means that after visiting f, the traversal "nondeterministically selects" one of the two argument positions to inspect next.

Definition 21 (Consistency) *Let t and t' be any two terms and $\mathcal{S}$ and $\mathcal{S}'$ be the respective conversions to set of strings. Let $S = (p_1, c_1) \cdots (p_n, c_n)$ and $S' = (p'_1, c'_1) \cdots (p'_m, c'_m)$ be any two strings such that $S \in \mathcal{S}, S' \in \mathcal{S}'$ and $S \neq S'$. Also let j be the maximum number such that $(p_i, c_i) = (p'_i, c'_i)$ for $1 \leq i \leq j$. Then the conversion is consistent only if $j < min(n, m)$ and $p_{j+1} = p'_{j+1}$.*

Only top-down, consistent conversions correspond to traversals. Consistency ensures that the next position visited depends only upon portions of the term inspected and not on the un-inspected portions. Having formalized the notion of a traversal, we now show how a notion of sequentiality can be based upon any such order of traversal.

Definition 22 *Given a traversal order $\mathcal{A}$, we say that an equational program is $\mathcal{A}$-sequential iff the following condition holds for any two lhs l_1 and l_2: let $\mathcal{S}_1$ and $\mathcal{S}_2$ be the set of strings obtained from l_1 and l_2 respectively using $\mathcal{A}$-traversal. Also let $S_1 = (p_1, c_1) \cdots (p_n, c_n)$ and $S_2 = (p'_1, c'_1) \cdots (p'_m, c'_m)$ be any two strings in $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $(p_i, c_i) = (p'_i, c'_i)$ for $1 \leq i < j$. Then it must be the case that either $c_j = c'_j$ or both c_j and c'_j are nonvariable symbols (note $p_j = p'_j$ by consistency).*

To see that strongly sequential systems fit into this definition, observe that the matching automaton for such systems actually specifies a traversal order that satisfies the top-down and consistency properties. Note that each position visited is an index and hence no lhs has a variable at any position visited while traversing as specified in the automaton. Thus every path in the automaton from the start state to a final state specifies the way to traverse all the nonvariable nodes in an lhs. (The order in which the variable nodes are visited is clearly irrelevant and hence is not specified in the automaton.) It is quite easy to see that the definitions of strong left-sequentiality and path sequentiality can be obtained from the above definition by letting $\mathcal{A}$ denote preorder and path traversals respectively. (We skip detailed proofs of these facts here since they are not important in establishing the results contained in the rest of the section.) The above definition can also be used to yield other notions of sequentiality such as based on breadth-first traversal or unordered depth-first traversal. (In an unordered depth-first traversal, the nodes are visited depth-first but the relative order in which the children of a node are visited can be arbitrary. Programs observing the notion of sequentiality based on such a traversal are called *strongly ordered programs* by Strandh [Str89].)

8.1.2 Inclusion Relationships among Notions of Sequentiality

We now establish the inclusion relationships among various notions of sequentiality as shown in figure 11.

Lemma 23 *For any traversal $\mathcal{A}$, any program that is $\mathcal{A}$-sequential is also strongly sequential.*

Proof: We establish this lemma by contradiction: assume that there exists a set of lhs $\mathcal{L}$ that is not strongly sequential, but is $\mathcal{A}$-sequential for some traversal order $\mathcal{A}$. Then, by definition of strong sequentiality, there must exist a term u that unifies with at least two lhs in $\mathcal{L}$, but has no index.

Now, consider the set of strings $\mathcal{S}$ obtained by traversing u using $\mathcal{A}$. There must exist one string $S \in \mathcal{S}$ that corresponds to visiting some sequence of positions within u and then visiting a position p such that u/p is a variable, but there is some lhs $l \in \mathcal{L}$ that unifies with u that has a nonvariable at p. (Such a p must exist or otherwise u will be a redex.) Let $S = (p_1, c_1)(p_2, c_2)\cdots(p_n, c_n)(p, c)$ be the string obtained, where c is a variable (i.e., $c = *$). Let $\mathcal{L}_S$ be the set of those lhs that, when traversed using $\mathcal{A}$-traversal, yield a string with the prefix S. Consistency and $\mathcal{A}$-sequentiality require that $\mathcal{S}$ include all the lhs that unify with u. Furthermore, since u is not a redex, $\mathcal{A}$-sequentiality implies that p be either a variable in all lhs in $\mathcal{L}_S$ or be a nonvariable in every lhs in $\mathcal{L}_S$. The choice of p ensures that at least one lhs in $\mathcal{L}_S$ has a nonvariable at p and hence by the above assertion, all lhs in $\mathcal{L}_S$ have a nonvariable at p. Therefore p is an index of u – a contradiction. ∎

Lemma 24 *For any traversal $\mathcal{A}$, if an equational program is path sequential then it is also $\mathcal{A}$-sequential.*

Proof: We establish by showing the contrapositive, i.e., whenever a program violates $\mathcal{A}$-sequentiality, then it also violates path sequentiality. Since the program violates $\mathcal{A}$-sequentiality, there must exist two strings S_1 and S_2 (belonging to the sets $\mathcal{S}_1$ and $\mathcal{S}_2$ of strings obtained from two lhs l_1 and l_2) that have a common prefix $(p_1, c_1)\cdots(p_j, c_j)$, but l_1 has a variable at p_{j+1} whereas l_2 has a nonvariable c_{j+1}. Observe that by the top-down condition that all the ancestors of p_{j+1} occur among $p_1, ..., p_j$. Let these by $p_{i_1}, ..., p_{i_k}$. Now, note that[6]

$$pathstr(l_1, p_{j+1}) = (p_{i_1}, c_{i_1})(p_{i_2}, c_{i_2})\cdots(p_{i_k}, c_{i_k})(p_{j+1}, *)$$

$$pathstr(l_2, p_{j+1}) = (p_{i_1}, c_{i_1})(p_{i_2}, c_{i_2})\cdots(p_{i_k}, c_{i_k})(p_{j+1}, c_{j+1})$$

Thus, by definition, this program violates path sequentiality. ∎

8.1.3 Transformation and Notions of Sequentiality

In the previous section we established the inclusion relationships among various notions of sequentiality as shown in figure 11. Since any $\mathcal{A}$-sequential system is strongly sequential, our transformation procedure can transform any such system into an equivalent path sequential system. Since path sequential systems observe every notion of sequentiality, our transformation procedure provides a way to transform a system that observes any one notion of sequentiality into a system that observes any other notion of sequentiality. This implies that all notions of sequentiality based on top-down traversals of terms are equivalent under our syntactic transformation. It also implies that programs observing any one notion of sequentiality are as expressive as programs observing any other notion of sequentiality.

8.2 Relationship between Original and Transformed Systems

Note that one rule in the original system $\mathcal{R}$ may generate several rules in the transformed system $\mathcal{R}^N$. However, we now show that the time spent for matching and reducing a single rule in $\mathcal{R}$ is the same as that for matching/reducing all the rules it generated in $\mathcal{R}^N$. For instance, let a rule $l \to r$ in $\mathcal{R}$ generate two rules $roof(l, \mathcal{L}) \to g(x_1, ..., x_n)$ and $l_g \to r$. In order to match these rules, note that after examining all the nonvariable symbols in $roof(l, \mathcal{L})$, there is no need to explicitly reduce it to $g(\cdots)$. Instead, we continue to examine the nonvariable symbols in l_g (except its root g) and finally perform a rewrite into r. Note that the sum of the number of nonvariable symbols in $roof$

[6] Here, we use a notation for path strings that is consistent with that used for denoting traversals. The correspondence between the original notation and this should be obvious: where we used an integer in the string to give relative position of a node with respect to its parent, here we use the full path.

and l_g (except the root of l_g) equals the number of nonvariable symbols in l and hence we have not spent any more time in matching in the transformed system. Also observe that in the above procedure for matching/rewriting in the transformed system, a single rewrite was performed after a series of matches. Thus, instead of doing an explicit transformation, we could build it directly into the matching automaton which will guide the normalizing procedure to perform a series of matches before a rewrite.

It may be intriguing how our transformation procedure transforms a system in which search for needed redexes cannot proceed independently along different paths into one in which this is possible. Note that if there is a block at p for an lhs l in the original system, then the search for needed redexes cannot proceed below p (in an input term s whose symbols on these paths match that of l) without the information about the symbols seen on other paths. Thus the search on this path must be suspended and can proceed only after a communication step that informs the processor holding the node at p about the symbols seen on other paths. This wait is present in the transformed system as well since $roof(l)/p$ is a variable. The information about symbols in the other paths is obtained by performing one or more reductions using rules $roof(l) \rightarrow g(\cdots), \ldots$ generated from l.

Although the transformation does not reduce the communication steps or waits, it does result in a matching automaton whose size is linear in the size of the original system. (Note that the size of the automaton is proportional only to the number of nonvariables symbols in the rule. Although a single rule is transformed into several rules, observe that the sum of the number of nonvariable symbols in all these rules is of the same order as the number of nonvariables in the original rule.) However we illustrate that this reduction in number of states can be potentially traded for *some* parallelism. Suppose that there is a block in a rule l at p and that l/p is a nonvariable term. Furthermore, assume that q is another block free path in l and that $c = root(l/q)$ is different from the symbol at q in any other lhs. In such a case, note that the search for needed redexes can proceed below p as soon as the symbol at q is determined to be c. But in the transformed system, the search will not proceed below p until every symbol in $roof(l)$ are inspected and a rewrite is performed.

However, it is not clear whether in practice this results in a significant loss of parallelism. This is because programs that are not path sequential appear not well suited for parallel evaluation to start with. For instance, the program for searching a tree presented in section 3.1 can be modified so that it includes the following two rules instead of the rules 3–6 defining *select*:

$select(found(x), y) \rightarrow found(x)$

$select(nil, y) \rightarrow y$

Lazy evaluation of this modified program requires that any redex in the second argument of *select* be reduced only if the first argument reduces to *nil*. This rules out the possibility of parallel searching and makes the algorithm sequential. On the other hand, path sequential systems express parallel computations and lead to a simple and efficient normalization algorithm with minimal communication. Thus it appears attractive to use path sequential systems as the basis for a parallel implementation. In such a case, our transformation offers an alternative to restricting the programmer to write only path sequential programs.

9 Concluding Remarks

In this paper we introduced a new class of strongly sequential systems called path sequential systems. We showed that such systems provide an elegant framework for lazy parallel evaluation of equational programs. In these systems the search for needed redexes can proceed independently along different paths of the input term. Communication between processors is reduced and is also uniform and localized between a processors holding parent and child nodes of the input term. Furthermore, matching in such systems can be driven by an automaton whose size is linear in input program size. From the results of this paper, it appears attractive to have path sequential systems as the basis for a parallel implementation of equational programs.

We also gave a transformation algorithm that transforms any strongly sequential system into an equivalent path sequential system. The consequence of this is that the path sequentiality restriction can be made transparent to the programmer. Finally, we discussed the equivalence of various notions of sequentiality based on top-down traversals.

We remark that the parallel normalization strategy developed in this paper was based only on analysis of lhs of the rewrite rules. Significant improvements can be obtained in terms of parallelism and efficiency by an analysis of rhs as discussed in [SSR90, Sek91]. Moreover, a real implementation must take into account many implementation details such as how to distribute the input term, load balancing etc. Such issues are addressed in EQUALS [KPRRS92], which is our parallel implementation of equational language on the Sequent shared memory multiprocessor.

References

[BMS80] R.BURSTALL, D.MACQUEEN AND D.SANELLA, HOPE: An Experimental Applicative Language, *1st Int. LISP Conf., 1980.*

[DJ91] N. DERSHOWITZ AND J.P. JOUANNAUD, Rewrite Systems, *Handbook of Theoretical Computer Science, Ch. 6, Vol II, North-Holland, 1991.*

[HL79] G. HUET AND J.J. LEVY, Computations in Nonambiguous Linear Term Rewriting Systems, *Tech. Rep. No. 359(1979), IRIA, Le Chesney, France.*

[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. O'DONNELL , Implementation of an Interpreter for Abstract Equations, *10th Symposium on Principles of Programming Languages, 1984.*

[KM89] J.W. KLOP AND A. MIDDELDORP, Sequentiality in Orthogonal 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.*

[KPRRS92] K. OWEN, S. PAWAGI, C. RAMAKRISHNAN, I.V. RAMAKRISHNAN, R.C. SEKAR, Fast Parallel Implementation of Functional Languages - The EQUALS Experience, *Lisp and Functional Programming*, 1992.

[Lav88] A. LAVILLE, Implementation of Lazy Pattern Matching Algorithms, *European Symposium on Programming, 1988.*

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

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

[PS90] L. Puel and A. Suarez, Compiling Pattern Matching by Term Decomposition, *Lisp and Functional Programming, 1990.*

[Sek91] R.C. Sekar, Efficient Evaluation of Equational Programs, *PhD thesis, Department of Computer Science, State University of New York at Stony Brook, NY, USA, 1991.*

[SR90] R.C. Sekar and I.V. Ramakrishnan, Programming in Equational Logic: Beyond Strong Sequentiality, *Logic in Computer Science, 1990. Also to appear in Information and Computation, 1993.*

[SR92] R.C. Sekar and I.V. Ramakrishnan, Lazy Parallel Evaluation for Programming with Rewrite Systems, *11th International Conference on Automated Deduction, 1992.*

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

[SSR89] R.C. Sekar, S. Pawagi and I.V. Ramakrishnan, Transforming Strongly Sequential Rewrite Systems with Constructors for Efficient Parallel Execution, *International Conference on Rewriting Techniques and Applications, 1989.*

[SSR90] R.C. Sekar, S. Pawagi and I.V. Ramakrishnan, Small Domains Spell Fast Strictness Analysis, *Symposium on Principles of Programming Languages, 1990.*

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

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