Programming with Equations: A Framework for Lazy Parallel Evaluation*

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

Abstract

Huet and Levy pioneered lazy sequential evaluation of equational programs based on the concepts of *strong-sequentiality* and *needed redexes*. Natural extensions of their strategy are not well-suited for parallel evaluation since they do not support independent searches for needed redexes along different paths in the input term. Furthermore, the size of compiled code can be exponential in program size. We therefore propose a different notion of sequentiality called *path-sequentiality* that overcomes these drawbacks and thus provides a natural framework for lazy parallel evaluation. We present a sound and complete algorithm for lazy parallel normalization of path-sequential systems. We show that our algorithm is optimal in the sense that its time complexity is bounded only by the time required to perform the needed reductions. The results presented in this paper are applicable to functional languages as well through the transformation of Laville.

1 Introduction

Programming with equations 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). Such an implementation is said to be *complete* if it finds the normal form s whenever $s = t$ follows logically from the input rules.

An equational program is evaluated by *term rewriting*, i.e., by repeatedly replacing *redexes* (instances of lhs) in the input term by the corresponding right-hand sides (rhs) until the input term is normalized. An implementation based on term rewriting can be complete if the input programs are guaranteed to be *confluent* or Church-Rosser. For programming purposes it is natural to do this by requiring input

*Partially supported by NSF grants CCR-8805734, 9102159 and NYS S&T grant RDG 90173.
†Current address: Bellcore, 445 South Street, Morristown, NJ 07962.

programs to be *orthogonal*[1] [DJ90].

For orthogonal systems (without overlaps at the root) Huet and Levy [HL79] developed a theory of optimal reduction strategies 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 showed that in such systems, every term that is not in normal form will have a needed redex and that a normalization strategy that reduces a needed redex at every step is complete. Any such strategy that reduces only needed redexes is said to be *lazy*. Unfortunately, needed redexes are not computable for all orthogonal programs[2]. Therefore they developed the class of *strongly-sequential systems* for which needed redexes can be computed based on the lhs alone. Subsequently, Hoffman and O'Donnell [HO84] developed an equational interpreter for *strongly-left-sequential systems*, which are a subclass of strongly-sequential systems.

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. Our high-level model of a parallel computation consists of (evaluator) processes interconnected by links. (The links may be implemented either through shared-memory or a network.) In [SSR89] we had proposed a notion of sequentiality called *path-sequentiality* that facilitates parallel evaluation in a natural way on such a model. But the focus in [SSR89] was only to formally demonstrate that the expressive power of path-sequential systems is the same as that of strongly-sequential systems. We show that in [SSR89] by syntactically transforming any strongly-sequential system into an equivalent path-sequential system. But *all* questions regarding parallelization of path-sequential and strongly-sequential systems were left unanswered, e.g., what are the inherent difficulties in parallelizing strongly-sequential systems on such a model? Do path-sequential systems overcome such problems? Furthermore, the important problem of developing a sound and complete normalization strategy for lazy parallel evaluation of path-sequential systems was not addressed. We address all these questions in this paper and present a simple and efficient normalizing algorithm and study its computational and communication complexities. The results in this paper complement the results on expressive power of path-sequential systems in [SSR89]. Together they complete a thorough study of problems, mechanisms and solutions to lazy parallel evaluation of equational programs. The following is a summary of results.

1.1 Main 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. To avoid excessive communication and synchronization overheads, the search on different paths should proceed inde-

[1] In such systems, the lhs of rules are linear and no two lhs unify except possibly at the root. If two lhs unify then their most general unifier must also unify the corresponding rhs. Until recently [DJ90] the term *regular* was used in the literature in place of *orthogonal*.

[2] Although Kennaway [Ken89] has presented a *sequential* algorithm for all orthogonal programs, this algorithm does not compute needed redexes.

pendently. In section 2, 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*. For such systems, we outline a strategy (described in detail in section 3) that searches for needed redexes *independently* along different paths with low communication and synchronization overheads. More importantly, the size of the compiled code (with this strategy) is linear in the size of input program. In contrast, natural extensions of the Huet-Levy strategy will result in exponential blow-up of code size *even for path-sequential systems*.

• In section 3 we devise a parallel lazy normalization algorithm (called *Normalize*) for path-sequential systems. Our algorithm is simple and efficient and demonstrates the appropriateness of path sequential systems as a basis for lazy parallel evaluation.

• We establish the soundness and completeness of *Normalize* in section 4. In spite of the simplicity of *Normalize*, these proofs are quite tricky and involved.

• Although it may appear that *Normalize* may spend a lot of time searching for needed redexes, we show in section 5 that the time complexity of *Normalize* is bounded only by the time required to perform the needed reductions. Thus the search for needed redexes does not add to the complexity. We also show that matching (i.e., checking for the presence of a redex) is done with minimal communication.

• In section 6 we show that the parallelism exploited by *Normalize* cannot be enhanced without looking at the rhs. We then remark how parallelism can be enhanced by analyzing the rhs using a technique called *ee*-analysis [SSR90]. Finally, we describe modifications to *Normalize* to enhance parallelism using this information.

For the rest of the paper, we will be dealing only with programs that follow *constructor discipline*, i.e., the outermost symbols of the lhs will be distinct from any other symbol appearing inside. It may appear that path-sequential constructor-based systems are restrictive. Therefore we now show how our results can be applied to functional or equational programming languages.

• Most functional languages such as ML [HMT88] and Hope [BMS80] permit overlapping rewrite rules, but make use of priorities to disambiguate among them. Laville [Lav87] has shown how these rules can be transformed into equivalent constructor-based orthogonal system. The transformed programs must be strongly-sequential (as they most often are) in order to have a complete lazy normalization procedure. We can now transform this system into a path-sequential system using our transformation presented in [SSR89]. Thus our results are applicable to functional programming languages as well.

• For languages based on orthogonal systems such as O'Donnell's equational language [OD85], programs that violate the constructor discipline can be transformed into constructor-based systems using Thatte's transformation procedure [That85][3].

We would like to remark that the above transformation procedures are often not necessary since most programs in practice (e.g., almost all the programs in [OD85]) appear to be already path-sequential and constructor-based.

[3]A caveat here is that Thatte's transformation does not always preserve strong-sequentiality.

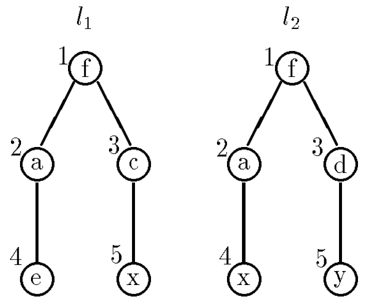

Figure 1

1.2 Notational Preliminaries

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. 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 $pathstr(l_1, 1.1) = f.1.a.1.e$. We use $root(t)$ to denote the root symbol 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 $\mathcal{R}$. $\mathcal{L}$ denotes the corresponding set of lhs.

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 u is a *redex* if $u \geq l$ for some lhs l. A reduction $t \rightarrow_l u$ means that u is obtained by replacing a redex $l\beta$ in t by $r\beta$, where $l \rightarrow r$ is a rewrite rule. (Sometimes we omit the subscript on $\rightarrow$.) The reflexive and transitive closure of $\rightarrow$ is denoted $\rightarrow^*$. A term t is in *normal form* (NF) if it contains no redexes. It is in *head normal form* (HNF) if there is no term t' such that $t \rightarrow^* t'$ contains a redex at the root. A *needed redex* is one that is rewritten in every reduction of t to NF.

2 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 Huet-Levy algorithm for parallel evaluation. We then show how these problems can be avoided by using a new normalizing strategy based on path-sequential systems. We start by outlining the Huet-Levy strategy.

2.1 Huet-Levy Normalization Strategy

Suppose the TRS $\{f(a(e), c(x)) \rightarrow x, f(a(x), d(y)) \rightarrow x\}$ (the lhs of this TRS are shown in fig.1) 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). More generally, let u be a *constructor prefix* of t, i.e., all nonvariable symbols in u, except the one at its root, are constructors. If all the lhs that unify with u have a nonvariable symbol at p (where u/p is a variable) 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. Such a p is called an *index* for u. In the above example $u = f(a(x), y)$ and $p = 2$. In a strongly-sequential system every constructor prefix that unifies with a nonempty set of lhs has an index.

The Huet-Levy 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. Reducing this redex yields 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, language implementations compile the lhs into an automata (which may either be represented by tables or case statements) that drives the matching process. 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 is any symbol that could appear at p in any lhs l that unifies with u. Fig.2 is such an automaton for the above example.

2.2 Extending Huet-Levy Strategy for Parallel Evaluation

Observe that in the Huet-Levy strategy, each call to head normalize a subterm may reduce several redexes before returning. Thus a 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 exploiting this parallelism, the processor P initiating a match at the root of t would compute all indexes $p_1, ..., p_k$ and then initiate computations to head normalize $t/p_1, ..., t/p_k$ 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 because of the nature of matching in strongly-sequential systems.

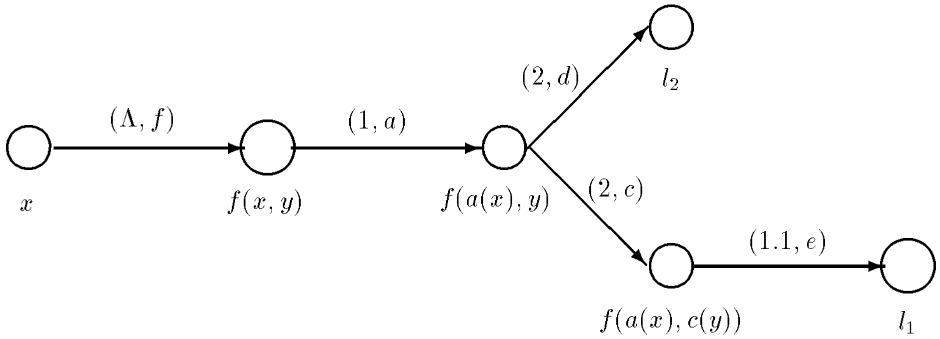

Figure 2: Automaton for patterns in fig.1

Such a centralized strategy suffers from two serious drawbacks. Firstly, P may have to successively copy (potentially) a 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 ultimately discovered). 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 has $root(l/p_i) = c$. This contrasts sharply with the sequential case wherein there are transitions (p, c) for *only one* index p.

We will now show that 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. 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.

2.3 Path-Sequential Systems

The exponential blow-up discussed in the previous section 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* 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. In particular, 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 only on the information sent to it by its parent P, whether t_i is a variable substitution or is to be head normalized to match a nonvariable symbol of an lhs. This is possible for the class of *path-sequential systems*:

Definition 1 *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.*

In other words, path-sequential systems are a subclass of strongly-sequential systems wherein, given any prefix u and a position p in u, we can determine whether p is an index for u based only upon the symbols on the path p. 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*. Observe that each such prefix starts at the root of an lhs and ends at a unique nonvariable node of this lhs. Consequently, the number of such prefixes is bounded by the total number of distinct nonvariable nodes in different lhs and is thus linear in the sum of the sizes of the lhs in the TRS.

In summary, a natural extension of the Huet-Levy strategy for parallel evaluation makes use of a centralized strategy that incurs significant amount of communication and synchronization overheads. More importantly, this strategy results in an exponential blow-up in the size of the matching automaton[4] *even for programs that are*

[4] Although this was established with a TRS consisting of a single rule, note that for TRS consisting of many rules, the automaton can have one state corresponding to each possible prefix of every lhs.

path-sequential. In contrast path-sequential systems provide a natural and 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 is linear in size of the rules.

3 Parallel Normalization Algorithm

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. In this algorithm, let $nvsons(\mathcal{L}, p)$ denote the set of integers

$$\{i \mid \exists l \in \mathcal{L}\ root(l/p.i) \text{ is a nonvariable } \}$$

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.

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 $\mathcal{S}' = \{l \in \mathcal{L} \mid pathstr(l, p) =$

Thus exponential blow-ups can occur in practice.

$pathstr(u,p)\}$. 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 are 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 will be 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.

4 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, 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, 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 2 (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 each (recursive) invocation $Match(p.j, Hnorm(v/j), \mathcal{S}')$ returns with $\mathcal{S}_j$ and term v_j such that $\mathcal{S}_j = \{l \in \mathcal{S}' \mid v_j \geq l/p.j\}$. Now it is easy to see that the intersection of $\mathcal{S}_j$'s yields $\mathcal{S}''$. ∎

Lemma 3 (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:

Case 1. $Match$ returns from line 11: Let $\mathcal{S}_j$ 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, $v = v'$ 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 4 (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 5 (Completeness) *$Normalize(t)$ terminates if 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 Complexity Issues

We modify $Normalize$ as follows before we proceed to establish its time complexity. First, observe that $Normalize$ uses a tree representation of terms. Because of this, reducing an outer redex can result in duplicating inner redexes. So the number of reductions performed by it may not be minimal although it performs no wasteful reductions. We can avoid duplication by keeping the rhs as fully compact dags and thus sharing multiple occurrences of these redexes. The details are straight-forward and so we omit them here.

Second, note that *Normalize* operates by repeatedly searching for redexes and replacing them. The search may often involve repeated traversals of the term and so a lot of time may be spent in such traversals rather than in performing reductions. Therefore we associate a flag with each node to indicate whether the term rooted at the node is in HNF. This flag is set on return from and checked before entry into (and the call skipped if set) *Hnorm*.

With the above modifications we now show that the time spent by *Normalize* is bounded by the time required to perform the needed reductions. To establish this result, 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). We also assume that the input term is *legal*, i.e., its normal form will not contain function symbols. In languages that follow the constructor discipline (such as ML and HOPE), it is an error attempting to normalize a term that is not legal and so we ignore such terms for the time complexity result.

Theorem 6 *Let $t \rightarrow_{l_1} t_1 \rightarrow \cdots \rightarrow_{l_n} t_n$ 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 into times spent inside the body of *Match*, *Hnorm* and *Normalize* respectively. 1. Time spent in *Normalize*: Note that it takes $O(1)$ time to invoke *Hnorm*. 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 dag representation of 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 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 bitvector representation. We distribute this cost as 1 unit each for the k recursive invocations of *Match* on the k subterms. Thus each execution 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 hence $Match(\Lambda, u, \mathcal{L})$ will never have been invoked. Thus the function symbol at the root of u will appear in the NF and hence the input term is not legal. ∎

We remark that any algorithm based on term rewriting 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.

5.1 Communication Complexity of $Match$

As mentioned earlier (see section 2), 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 can also be minimized: one communication step for each substitution that resides on a processor different from P.

6 Enhancing Parallelism

Normalization algorithms based on lhs alone will never reduce a needed redex u if it is possible to create an outer redex v by rewriting redexes other than u. This is because an algorithm based on lhs cannot rule out the possibility that the rewrite of the redex v may "throw away" the redex u and hence u may not be needed. For instance, consider the TRS $\{f(a(x)) \to b\}$ and the term $f(a(f(a(b))))$. Here, the inner redex $f(a(b))$ is discarded when the redex at the root is rewritten. Therefore such algorithms postpone reducing a redex until it is clear that the redex will remain outermost. Consequently, by not reducing such redexes early, we encounter significant loss of parallelism even in path-sequential systems. We later show how this problem can be alleviated by analyzing the rhs. First we show that based on lhs alone, it is not possible to enhance the parallelism detected by $Normalize$.

Observe that an invocation of $Match(\Lambda, v, \mathcal{L})$ results in a downward traversal of all paths from the root of v. All redexes encountered on these traversals are reduced. The traversal on a path 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.

6.1 Enhancing Parallelism by Analyzing rhs

By analyzing the rhs, we can often rule out the possibility of some redexes being "thrown away". For instance, consider the TRS $\{f(a(x), y) \to y\}$ and the term

$f(a(b), f(a(c), b))$. Analysis of the rhs indicates that the substitution for the second argument of f is not discarded and therefore any redex inside this argument can be reduced. Therefore we can reduce the root redex and the redex $f(a(c), b)$ *simultaneously*. Observe that this would not be possible if we had only considered the lhs since we would then be unable to determine if the redex $f(a(c), b)$ is needed.

We have developed a technique called ee-analysis that analyzed the rhs of rewrite rules and specified which arguments of a function f must possess normal forms in order for an application of f to possess a normal form. The function f is said to be ee-strict in such arguments. Observe that by this definition, all constructors are ee-strict in all their arguments. The details of this analysis have appeared in [SSR90]. Our technique is conceptually simple and very effective in practice.

We now modify $Normalize$ to identify and reduce needed redexes as early as possible using the information provided by ee-analysis. With this modification (given below), $Normalize$ detects all available parallelism in many example programs such as matrix multiplication and sorting.

procedure $Normalize(v)$ /* v is the term to be normalized */
0. for $i := 1$ to $arity(root(v))$ do
   if $root(v)$ is ee-strict in ith argument then $Normalize(v/i)$
1. $Hnorm(v)$;
2. $Normalize(u \mid u$ is an immediate subterm of $v)$

7 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 with minimal communication. Furthermore, matching in such systems can be driven by an automaton whose size is linear in input program size. We then presented an algorithm for lazy parallel normalization of such systems based on the lhs alone and established its soundness and completeness. We also showed that it is optimal in the sense that its time complexity is bounded only by the time required to perform the needed reductions; the search for redexes does not add to this complexity. We then showed how to enhance the parallelism exploited using information obtained by analyzing rhs.

We have been continuing our work on parallel evaluation of equational programs and recently completed implementing EQUALS [OPRRS92], a system to support equational programs on the Sequent shared-memory multiprocessor. Our initial results were obtained on a six-processor Sequent Symmetry. On many example programs such as matrix multiplication and sorting our implementation achieves substantial speedups over existing sequential implementations of functional languages such as Standard ML of New Jersey. Preliminary results on a larger Sequent indicate that we obtain almost linear speedups on many programs even upto 16 processors.

References

[BMS80] R.Burstall, D.MacQueen and D.Sanella, HOPE: An Experimental Applicative Language, *Proc. of 1st Int. LISP Conf., 1980.*

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

[FGJM85] K. Futatsugi, J. Goguen, J.P. Jouannaud and J. Meseguer, Principles of OBJ2, *POPL 1985, pp. 52-66.*

[HMT88] R.Harper, R.Milner and M.Tofte, The Definition of Standard ML, *Report ECS-LFCS-88-62, LFCS, U. of Edinburgh, 1988.*

[HO84] C. Hoffmann, and M.J. O'Donnell , Implementation of an Interpreter for Abstract Equations, *POPL 1984, pp. 111 – 120.*

[HL79] G. Huet and J.J. Levy, Computations in Nonambiguous Linear Term Rewriting Systems, *TR No. 359(1979), INRIA, France.*

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

[Lav87] A. Laville, Lazy Pattern Matching in the ML Language, *Proc. of FST&TCS, Springer-Verlag LNCS, 1987.*

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

[OPRRS92] K. Owen, S. Pawagi, C.R. Ramakrishnan, I.V. Ramakrishnan and R.C. Sekar, Fast Parallel Implementation of Lazy Languages – The EQUALS Experience, *Lisp and Functional Programming, 1992.*

[SSR89] R.C. Sekar, S. Pawagi and I.V. Ramakrishnan, Transforming Strongly Sequential Rewrite Systems with Constructors for Efficient Parallel Execution, *RTA 1989.*

[SSR90] Sekar, R.C., Shaunak Pawagi and Ramakrishnan, I.V., Small Domains Spell Fast Strictness Analysis, *POPL 1990.*

[Str88] Robert I. Strandh, Compiling Equational Programs into Efficient Machine Code, *PhD Thesis, Johns Hopkins University, 1988.*

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