THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

DEMAND DRIVEN EVALUATION WITH EQUATIONS

Satish Thatte

CRL-TR-34-84

August 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

Demand Driven Evaluation with Equations

Satish Thatte

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

1. Introduction

Term rewriting systems, such as The λ-calculus (Barendregt, 1981), combinatory logic (Curry and Feys, 1958), and the FP system (Backus, 1978), are fundamental to applicative programming. A recent trend is to permit a programmer to describe term rewriting systems using sets of first-order equations (Burstall, *et al*, 1980, Goguen and Tardo, 1979, Hoffmann and O'Donnell, 1982, Turner, 1976). These systems are the focus of this paper, and we refer to them as First-Order Equational Systems (FOES). FOES expand the expressive power of applicative programming in many ways. They can be used to directly define primitive functions on tree-like data-structures (Burstall, *et al*, 1980). Hoffmann and O'Donnell (1982) describe their use in succinct definitions of interpreters for languages such as LISP and LUCID (Ashcroft and Wadge, 1977). Their reliance on pattern matching in defining functions largely eliminates the need for the explicit use of selectors and predicates such as LISP's CAR, CDR and NULL, and greatly improves clarity and readability. FOES readily admit a many-sorted interpretation (Burstall, *et al*, 1980, Goguen and Tardo, 1979), and polymorphism (MacQueen, 1981, Milner, 1978), and hence a very sophisticated form of strict static type-checking. A lazy interpretation of FOES extends the notion of computation with infinite structures to all tree-like structures, and indirectly permits applicative definitions of circular data structures such as doubly-linked lists, which can be viewed as cyclically infinite binary trees.

An examination of the operational semantics of existing term rewriting systems reveals that two approaches are possible. The evaluation of an expression in any rewriting system amounts to repeated application of the rewrite rules until an irreducible (normal) form is reached. Most expressions admit many alternative routes for the rewriting process, not all of which may terminate. Those that do terminate may not all reach the same normal form. Some systems determine the correct route, or computation rule, by decree. For instance, HOPE (Burstall, *et al*, 1980) uses innermost first evaluation except for a special variant of the cons constructor. FP (Backus, 1978) and OBJ (Goguen and Tardo, 1979) uniformly use the innermost first rule. The advantage in choosing a rule based on operational considerations is that efficiency and simplicity of implementation can be built into the choice. Often, restrictions on the permissible combinations of equations are also avoided.

The other commonly used approach is to treat the rewrite rules as the basis for a congruence relation and declare congruence classes to be the meaning of the terms they contain. The correctness or safeness of a computation rule is judged by the requirement that it must reduce each term to its *unique* normal form equivalent if one exists. This more "mathematical" approach has been used for the λ-calculus, combinatory logic, and some FOES (*see*, *e.g.*, Hoffmann and O'Donnell, 1982). Apart from the obvious appeal of semantic elegance, the mathematical approach coincides with lazy evaluation, which has been shown to be an attractive way to deal with many programming problems (Friedman and Wise, 1976, Henderson and Morris, 1976, Kahn and MacQueen, 1977). One clear limitation of the approach is that it can only be applied to systems which satisfy the confluence property (Huet, 1977). More troublesome is the need to *find* a safe and efficient computation rule for each system.

The standard example of a safe and optimal computation rule is the "normal order" rule for the λ-calculus (Vuillemin, 1974). The trouble with FOES is that they are meant to be user-defined, therefore it is impossible to look for an appropriate computation rule for each individual system. The ideal solution would be a rule that is safe and optimal for all confluent FOES. The "full substitution" rule of Vuillemin (1974) is safe but not optimal, since it requires all outermost redices in an expression to be reduced, including, in general, many unnecessary ones. One way to approach optimality is to look for a safe *sequential* rule, i.e., a rule that selects exactly one redex at each step. Such a strategy turns out to be optimal in number of reduction steps in the so-called noncopying implementations, in which expressions are represented by directed acyclic graphs, and all residuals of a given subexpression share the same representation (O'Donnell, 1977, Vuillemin, 1974). Safe sequential evaluation is identical in spirit to the less specific notion of *demand driven* evaluation, and we use the two terms interchangeably throughout this paper. Unfortunately, not only is a safe sequential rule for all confluent FOES not known, it almost certainly does not exist, since taking account of the right-hand sides of equations amounts to unbounded look-ahead. The reported work on the subject therefore confines itself to considering sequentiality based only on the left-hand sides of equations (Hoffmann and O'Donnell, 1984, Huet and Levy, 1979).

Our goal is to extend the notion of demand driven evaluation to confluent FOES, i.e., to find a computation rule for FOES that is both safe and sequential, even when the right-hand sides of equations are disregarded. The most commonly studied class of confluent FOES is the class called *regular* by Hoffmann and O'Donnell (1984), and *nonambiguous linear* by Huet and Levy (1979). We shall adopt the former term. In this paper, we study a subclass of regular FOES that is practically important and substantially easier to deal with. The defining characteristic of members of this class is that constructors and defined functions are represented by disjoint sets of symbols. This restriction is used in several actual languages incorporating FOES

(Burstall, *et al*, 1980, Subrahmanyam and You, 1983, Turner, 1976), and is analogous to the separation of function and predicate symbols in logic programming (Kowalski, 1979). We call this subclass *constructor* FOES to emphasize that argument patterns in equations of this class contain only constructor symbols and variables. It turns out that only a subset of the class of constructor FOES admits the kind of safe sequential rule we need. Along with the rule, we therefore need a decision procedure to identify members of this subset.

In Section 2 we informally describe our approach to the problem, Section 3 contains the mathematical results, in Section 4 we discuss possible applications and extensions of the results, Section 5 describes related work, and Section 6 contains the conclusions.

2. Informal Description

Let us start with an example to show that the leftmost outermost first rule is not safe for all FOES.

Example 1.

```
pair_int(X,nil) = nil
pair_int(X,A.L) = <X,A>.pair_int(X,L)
append(nil,L) = L
append(A.L1,L2) = A.append(L1,L2)
f = f
evaluate: pair_int(f,append(nil,nil))
```

The syntactic convention in *Example 1* and subsequent examples is that identifiers beginning with upper case letters are variables and the rest are function symbols. The symbols cons is represented by the infix operator "." for brevity. It is clear that although both arguments of pair_int in the expression are redices, choosing the leftmost first leads to divergence, while choosing the rightmost one quickly leads to the normal form nil for the entire expression. The example suggests that, first, it is necessary to infer the strictness or otherwise of a defined function with respect to each of its arguments, and second, a function is nonstrict with respect to an argument when that argument is represented by a variable in some equation.

Actually, the example is too simple to bring out the full complexity of the situation. The distinction between divergent and nondivergent arguments is not sharp. For instance, the second argument of pair_int needs only as much constructor structure as is necessary to match one of the two patterns available for it, as *Example 2* illustrates.

Example 2.

```
head(nil) = error,          head(A.L) = A
evaluate: head(pair_int(1,append(1.nil,f)))
```

Although an attempt to *completely* evaluate the second argument of pair_int in the expression would diverge, any outermost first strategy yields the normal form <1,1> for the entire expression. *Example 3* shows that even this degree of strictness may be conditional.

Example 3.

```
pair_list(X,nil) = nil
pair_list(A.L1,B.L2) = <A,B>.pair_list(L1,L2)
pair_list(nil,B.L2) = <-1,B>.pair_list(nil,L2)
evaluate: (1) head(pair_list(f,nil))
          (2) head(pair_list(f,1.nil))
```

Expression (1) leads to normal form "error" and (2) is divergent, although both contain exactly the same divergent subexpression in the same argument position. In other words, pair_list tolerates a divergent first argument only if the second argument is nil. Once the second argument reveals sufficient constructor structure to preclude this possibility, the structure of the first argument is needed to make further progress.

The outlines of the overall evaluation strategy we need have now emerged. As in any other demand driven strategy, applications of defined functions are evaluated from the outside inwards, as needed. Only the outermost applications are needed a priori. The reason for selecting any other application is that it stands in such an argument position for a needed outer application that its evaluated (constructor) structure must be made manifest in some degree, and the only way to reveal constructor structure is to reduce the application. In order to turn the application into a redex, we must decide which equation is applicable to it. We therefore choose an argument whose structure is critical to that decision. When sufficient structure has been revealed to narrow the choice down as far as possible, we choose another argument to narrow it down still further, and so on, until a single equation remains. Note that we concern ourselves only with the left-hand sides of equations. The structure of these must be such that we are able to choose a "critical argument" at each step. If we cannot, the strategy fails. Fortunately, it turns out that in such a situation *no* strategy can succeed if it relies only on the structure of the left-hand sides.

This somewhat simplified description has glossed over two important details. First, in all our examples so far, the argument patterns have consisted of at most one constructor. The approach generalizes to deeper patterns with a little care, as illustrated by *Example 4*. The example uses two equations for dual cases of AVL tree rebalancing by single rotations (*see* Knuth, 1973, p.454).

Example 4.

```
rotate(node(Alpha,d_plus,node(Beta,plus,Gamma)))
    = node(node(Alpha,balance,Beta),balance,Gamma)
rotate(node(node(Alpha,minus,Beta),d_minus,Gamma))
    = node(Alpha,balance,node(Beta,balance,Gamma))
```

Let us ignore the real significance of the balance factors d_plus, etc., and consider an arbitrary application of rotate. After ensuring that the outermost constructor of the argument is "node", neither the left nor the right subtree of the node can be safely evaluated since one of the equations has a "don't care" attitude in each case. The next critical structure is the balance factor for the node. If this is d_plus, then the second equation does not apply while the first may, and vice versa for d_minus.

The reader will have recognized that the selection process taking place here is commonly done manually in languages like LISP, where the outermost structure of an argument is first queried using a predicate such as NULL. The argument may then be taken apart using selectors, and the pieces further queried selectively, etc. It is instructive to rewrite the rotate function using conditionals, selectors, and predicates, in order to appreciate the gain in clarity brought about by using equations. The safe sequential computation rule we are seeking can be interpreted as an automatic version of the selection process based on the implicit logical necessity described by equations.

The second detail to note is that all functions may not be *exhaustively* defined, i.e., none of the equations may be applicable for certain combinations of arguments. In such an expression, which we call "root-stable" following Hoffmann and O'Donnell (1984), the outer function symbol is playing the role of a constructor. All applications immediately subordinate to a root-stable expression are therefore needed a priori, as in *Example 5*.

The outermost application of rotate cannot be dealt with

Example 5.

evaluate: rotate(node(head(e1),balance,rotate(e1,e2,e3)))

using either of the equations in *Example 4*, hence the applications of head and rotate immediately subordinate to it become "outermost". Another way to describe the situation is that the outermost rotate will be a part of any eventual normal form for the expression.

We conclude this section with two further examples. *Example 6* shows that all confluent FOES do not safely admit a sequential evaluation rule. The example uses the "parallel" logical or function provided as a primitive in certain applicative languages.

Example 6.

or(X,true) = true,   or(true,X) = true

or(false,false) = false

evaluate: or(g,h)

Case 1: g=true, h=h.   *Case 2:* g=g, h=true.

It is impossible to choose either argument for evaluation. Choosing the first argument is unsafe if g and h are defined by the equations labeled *Case 2*, and the second one is unsafe if *Case 1* is used. Finally, we would like to emphasize that FOES for which our approach fails may actually turn out to be sequential, if the right-hand sides are taken into account, as in the case of the "funny append" in *Example 7*:

Example 7.

```
append(L,nil) = L,      append(nil,L) = L
append(A.L1,L2) = A.append(L1,L2)
```

Although the first equation appears to add generality, in fact it is redundant, since lack of structure for the first argument still means lack of structure for an application of append. Therefore, it is still safe to evaluate the first argument first.

3. Mathematical Results

3.1 Preliminaries and Definitions

3.1.1 Preliminaries

Most of the material in this section is a review of terminology and constructions commonly used in the literature on term rewriting systems.

A reduction system operates in a non-empty *ranked alphabet* Σ which contains all symbols in the system. T_Σ denotes the set of all *ground* Σ-terms for the alphabet. Σ-terms, in general, may contain *variables* drawn from a countable set X. A Σ-term is said to be *linear* iff no variable occurs more than once in it. We drop the prefix signifying the alphabet and speak of terms whenever possible without confusion.

A *path* p is a possibly empty string of integers. We say that p *reaches* subterm t/p in term t. The empty string Λ reaches the term itself, the string 'k' reaches the kth argument, 'km' reaches the mth argument of the kth argument, etc. Given paths p and q, p.q denotes their *concatenation*. The symbols $\le$ and $<$ denote the *prefix* and *proper prefix* relations on paths respectively. Paths p and q are said to be *independent* iff neither $p \le q$ nor $q \le p$. *Paths*(t) denotes the set of all paths that reach some subterm in t. *Paths*(t) is partitioned into *XPaths*(t) and *ΣPaths*(t), where *XPaths*(t) is the subset that reaches variables in t. We frequently refer to a path $p \in$ *Paths*(t) as an *occurrence* of the *subterm* t/p in t. The expression $t[p \leftarrow w]$ denotes the term obtained by replacing t/p at p by w.

A *substitution* is a map from variables to terms. The meaning of a substitution can be extended naturally to a map from terms to terms. The application of a substitution α to a term t is conventionally denoted by $t\alpha$, where $t\alpha$ is the *instance* of t produced by *simultaneously* substituting $\alpha(x)$ for every variable x in t. We use the notation $t \le u$ to denote that u is an instance of t; $t < u$ means $t \le u$ and $t \ne u$. If neither $t \le u$ nor $u \le t$ then t and u are said to be *independent*. The relation $\le$ is clearly a partial order.

The usual first-order unification algorithm of Robinson (1965) is denoted by *UNIFY*. If two terms t and u have no common instance, then *UNIFY*(t,u) fails, otherwise it succeeds and returns a substitution α such that $t\alpha = u\alpha$ is the *least* common instance of t and u, i.e., in order-theoretic terminology, their *join*.

Since we shall consider safety of computations independent of the right-hand sides of equations, it is natural for us to treat the collection of left-hand sides as an independent entity with its own properties. We refer to this collection as a base for a reduction system.

Definition 1: A *base* L for a reduction system in the alphabet Σ is a finite set $\{l_i,\ m \geq i \geq 1\}$ of linear Σ-terms such that:

(1) $i \neq j$ implies l_i and l_j are independent.

(2) $\exists t \varepsilon T_\Sigma$ such that $l_i \not\leq t$, $m \geq i \geq 1$.

(3) If $l_1, l_2 \ \varepsilon\ L$ and $p \varepsilon \Sigma Paths(l_1)$, then $UNIFY(l_1/p, l_2)$ fails *unless* $p = \Lambda$ and $l_1 = l_2$.

Each member of L is called a *(redex) pattern*. ■

Condition (1) states that there are no redundant left-hand sides, (2) states that there are normal forms, and (3) states that there are no "critical pairs" (Knuth and Bendix, 1970), which is used to ensure that any reduction system based on L will be confluent (Huet, 1977).

Definition 2: A reduction system S in alphabet Σ is a finite set $\{<l_i, r_i>,\ m \geq i \geq 1\}$ of pairs of Σ-terms such that:

(1) $L = \{l_i,\ m \geq i \geq 1\}$ is a valid base.

(2) Each variable in r_i appears also in l_i, $m \geq i \geq 1$.

■

Throughout the following, we shall deal with a fixed set Σ of function symbols, a fixed base L, and a fixed reduction system S based on L, unless mentioned otherwise. In many definitions, the entities being defined are qualified with the subscript L or S signifying the context. These subscripts are dropped whenever the base or system concerned is the fixed one.

Any $u \in T_\Sigma$ such that $u \geq l$, for some $l \in L$, is called a *redex*. If t/p is a redex for some $p \in \Sigma Paths(t)$, then p is called a *redex occurrence* in t. The set of all redex occurrences in t is denoted by $RO_L(t)$.

A *simple reduction* $t \xrightarrow{s} u$ occurs in S iff $t/s=l\alpha$ for some $l \in L$, $<l,r> \in S$, and $u=t[s \leftarrow r\alpha]$. We write $t \longrightarrow u$ iff $t \xrightarrow{s} u$ for some $s \in RO(t)$. Finally, $t \xrightarrow{*} u$ is called a *reduction sequence* where $\xrightarrow{*}$ is the reflexive transitive closure of $\longrightarrow$. We use the notation $A{:}t \longrightarrow u$ in order to attach a name (in this case A) to a simple reduction, and similarly also $B{:}t \xrightarrow{*} u$. Λ ambiguously denotes all empty reduction sequences. As with paths, we use the notation A.B to denote the concatenation of sequences A and B. Such concatenation is meaningful only if the last term in A is identical to the first term in B.

The set $NF_L \subseteq T_\Sigma$ is the set of *normal forms*, and $t \in NF_L$ iff $RO_L(t)=\phi$. If $t \xrightarrow{*} u$ in S and $u \in NF_L$, then t is said to *normalize to* u in S, written as $u=NORM_S(t)$.

$P \subseteq Paths(t)$ is said to be an *independent set of paths* in t iff all pairs of distinct paths in P are independent. If $P \subseteq RO_L(t)$ then P is called an *independent set of redex occurrences*, denoted by $P \in ISRO_L(t)$. The significance of independent sets of redices is that the order in which they are reduced is immaterial to the result obtained by reducing them all. This permits us to introduce the notion of a multireduction.

Definition 3: Let $Q \varepsilon ISRO(t)$, and let $\{q_1,..,q_m\}$ be an arbitrary enumeration of Q, where $t/q_i=l_i\alpha_i$, $<l_i,r_i>\varepsilon S$, $m\geq i\geq 1$. Let $t_1=t$, and $t_{i+1}=t_i[q_i\leftarrow r_i\alpha_i]$, $m\geq i\geq 1$. We say that the *multireduction* $t\xrightarrow{Q}t_{m+1}$ occurs in S. The term t_{m+1} is clearly independent of the enumeration of Q. ∎

We treat a multireduction as a sequence of simple reductions whenever convenient. Although the notion of reduction sequence generalizes naturally to that of a multisequence, we do not need the generalization. In fact, our only serious use of multireductions occurs in Proposition 35, where it is necessary to deal with the forward migration of residual redices in rearranging a reduction sequence.

The notion of residuals, which is introduced next, is crucial in analysing the behavior of sequences of reductions. Residuals describe the way in which different parts of an expression are affected by a simple reduction. The left-hand side pattern in the redex is destroyed. Parts of the expression that are reached by extensions of the redex occurrence going beyond the pattern are rearranged according to the right-hand side. A possible rearrangement is disappearance, which is how outermost reductions sometimes make divergent inner expressions disappear. The rest of the expression is essentially unaffected.

Definition 4: Suppose $A:t\xrightarrow{a}u$, where $t/a=l\alpha$, $<l,r>\varepsilon S$. Let $q\varepsilon Paths(t)$. The set $q\backslash A$, called the *residuals* of q after A, is defined as follows:

(1) if $a \not\leq q$ then $q \backslash A = \{q\}$

(2) if $a=q$ then $q \backslash A = \phi$

(3) otherwise, let $a=q.s$. if $s \in \Sigma \mathit{Paths}(l)$ then $q \backslash A = \phi$. Otherwise, $\exists s' \in X\mathit{Paths}(l)$ such that $s'<s$. Let $s=s'.w$, $x=l/s'$, and $R = \{v \mid r/v=x\}$. Then $q \backslash A = \{a.v.w \mid v \in R\}$

■

*Proposition* _*5*_: If $A{:}t \xrightarrow{s} t'$ and $q \in \mathit{Paths}(t)$, then $q \backslash A$ is an independent set of paths.

*Proof*: Obvious from the definition of $q \backslash A$. ■

The following Proposition is a succinct statement of the reason for excluding "critical pairs" in Definition 1.

*Proposition* _*6*_: $q \in \mathit{RO}(t)$, $A{:}t \longrightarrow u$, implies that $q \backslash A \subseteq \mathit{RO}(u)$.

*Proof*: Straightforward by condition (3) in Definition 1. ■

Suppose $A{:}t \longrightarrow u$ and $Q \subseteq \mathit{Paths}(t)$. Then the set $Q \backslash A$ is simply $\cup\{q \backslash A \mid q \in Q\}$. This can be extended to sequences of reductions by composition, i.e.,

$$Q \backslash \Lambda = Q$$

$$Q \backslash B.C = (Q \backslash B) \backslash C$$

If $q \in \mathit{Paths}(t)$, we use the notation $q \backslash B$ instead of $\{q\} \backslash B$.

*Proposition* _*7*_: If $P \in \mathit{ISRO}(t)$, and $A{:}t \longrightarrow t'$, then $P \backslash A \in \mathit{ISRO}(t')$.

*Proof*: Simple consequence of Propositions 5 and 6. ■

We conclude this section by stating a fundamental property of our class of confluent FOES. This property is called "closure" by O'Donnell (1977), and the "parallel moves lemma" by Huet and Levy (1979). Another appropriate name would be the "one step confluence" property.

Lemma 8: Let $A{:}t\xrightarrow{P}u$ and $B{:}t\xrightarrow{Q}v$ be multireductions. Let $R=P\backslash B$, $S=Q\backslash A$, and $u\xrightarrow{S}w$. Then, $v\xrightarrow{R}w$.

Proof: By Proposition 7, $S\varepsilon ISRO(u)$, and $R\varepsilon ISRO(v)$. For the rest, see the proof of Lemma 11 in Huet (1977) or the proofs of Lemma 12 and Theorem 17 in O'Donnell (1977). ■

3.1.2 Definitions

The first concept in need of a definition is the concept of sequential evaluation itself. We are interested in sequential evaluation because we believe it embodies the prerequisites for efficient evaluation strategies for FOES. As such, our definition excludes the possibility of using look-ahead and/or memory in choosing redices.

A (*redex*) *selection* algorithm A is a function of a base L and a ground term t such that, $A(L,t)$ either fails or returns some $q\varepsilon RO_L(t)$. If L is fixed then the specialized or "curried" version of A is denoted by A_L.

Definition 9: Given a selection algorithm A, and $t\varepsilon T_\Sigma$, the *evaluation sequence* for t produced by A is the reduction sequence $t_0 \longrightarrow .. \longrightarrow t_n \longrightarrow ..$ such that:

(1) $t_0 = t$

(2) if $A_L(t_n)$ fails then the sequence *terminates in* t_n, else

$A(L)(t_n)$ returns r and $t_n\xrightarrow{r}t_{n+1}$.

The sequence is denoted by $\sigma_{A,S}(t)$. If the sequence terminates in t_n for some n, then we write $\sigma_{A,S}(t)\downarrow = t_n$, else we write $\sigma_{A,S}(t)\uparrow$. ■

Definition 10: A selection algorithm A is said to be *safe* for a system S based on L iff given any $u\varepsilon NF_L$ and any t such that $u=NORM_S(t)$, $\sigma_{A,S}(t)\downarrow = u$. ■

<u>*Definition*</u> <u>*11*</u>: A base L is said to be *sequential* iff there is a selection algorithm A which is safe for <u>any</u> S based on L. In this case, A is said to *sequentialize* L. ■

So far, we have been speaking of confluent FOES in general. We now introduce the class of constructor FOES which forms the subject of our study in this paper.

<u>*Definition*</u> <u>*12*</u>: Suppose $L = \{l_i,\ m\geq i\geq 1\}$. Let $l_i = f_i(u_{i1},..,u_{in_i})$, $m\geq i\geq 1$, $\Gamma_L = \{f_i,\ m\geq i\geq 1\}$, and $\Delta_L = \Sigma-\Gamma_L$. Δ_L is called the set of *constructors* of L. A term $f(u_1,..,u_m)$ is said to be a Δ_L*-pattern* iff $f\varepsilon\Gamma_L$ and each u_i is a Δ_L-term, $m\geq i\geq 1$. L is said to be a *constructor base* iff all $l_i \varepsilon L$ are Δ_L-patterns. A system based on such an L is called a *constructor system*. ■

Notice that, by definition, applications of constructors are always root-stable. In the following discussion, we assume that our fixed base L is a constructor base, and the alphabet Σ is accordingly partitioned into Γ and Δ, where Δ is the set of constructors.

We now have a precise definition of our problem. We need

(1) A decision procedure to identify sequential constructor bases, and

(2) A redex selection algorithm ("computation rule") that sequentializes *all* sequential constructor bases.

We still need a precise formulation of the demand driven approach outlined in section 2. The critical notion is that of a necessary redex, which is intuitively a redex that helps narrow down the set of equations compatible with the given term. The identification of compatible equations is based on the *known* structure of a given term, which is inferred using the fact that applications of constructors are root-stable.

Given a ground term t, u = *KNOWN*(t) is the largest linear Δ-term or Δ-pattern such that $u \le t$. Note that the set of all linear Δ-terms or Δ-patterns w such that $w \le t$ is nonempty and finite, and hence contains a largest member by the properties of *UNIFY*. The variables in *KNOWN*(t) are assumed to be new. For a ground term t, define

$L_t = \{l \in L \mid UNIFY(l, KNOWN(t)) \text{ succeeds}\}$,

where L_t is the set of equations compatible with t.

Let $PDNO(t) = XPaths(KNOWN(t))$. Each $p \in PDNO(t)$ is said to be *potentially directly necessary* for t. If, in addition, $p \in \Sigma Paths(l)$ for each $l \in L_t$, then p is said to be *directly necessary* for t, denoted by $p \in DNO_L(t)$. The notion of necessary redex occurrences essentially iterates directly necessary occurrences until a redex is reached. The exception is the situation noted in section 2, when root-stable outermost applications of defined functions must be taken into account. The structure of the definition below caters more to the needs of later proofs by structural induction than to intuitive transparency.

Definition 13: $p \in RO_L(t)$ is said to be *necessary* for t, written as $p \in NRO_l(t)$, iff *one* of the following holds:

(1) $p = \Lambda$ or $p \in \mathit{DNO}_L(t)$

(2) $q \in \mathit{DNO}_L(t)$, $p=q.r$, and $r \in \mathit{NRO}_L(t/q)$

(3) $\exists q \in \mathit{DNO}_L(t)$ such that $\mathit{RO}_L(t/q)=\phi$, $r \in \mathit{PDNO}(t)$, $p=r.s$, and $s \in \mathit{NRO}_L(t/r)$

■

Example 8 illustrates these definitions.

Example 8.

$L = \{f(X,a,b),\ f(b,X,a),\ f(a,b,X)\}$

$t = f(a,f(b,a,f(a,a,b)),f(b,b,a))$

$u = t/2 = f(b,a,f(a,a,b))$

$\mathit{KNOWN}(t) = f(a,Y1,Y2)$	$\mathit{PDNO}(t) = \{2,3\}$
$\mathit{DNO}(t) = \{2\}$	$\mathit{RO}(t) = \{2.3,3\}$
$\mathit{KNOWN}(u) = f(b,a,Y3)$	$\mathit{PDNO}(u) = \{3\}$
$\mathit{DNO}(u) = \{3\}$	$\mathit{RO}(u) = \{3\}$
$\mathit{NRO}(u) = \{3\}$	$\mathit{NRO}(t) = \{2.3\}$

Our whole approach rests on the assumption that a necessary redex can always be found in a reducible expression. This is clearly not the case for all FOES, and our conjecture is that *only* those FOES which satisfy this assumption are sequential in the sense of Definition 11. We say that the bases of such FOES are strictly sequential.

Definition 14: L is said to be *strictly sequential* iff whenever $\mathit{RO}_L(t)$ is nonempty, $\mathit{NRO}_L(t)$ is also nonempty. ■

It is clearly straightforward to write an algorithm that, given a strictly sequential base and a reducible expression, will always return a necessary redex occurrence.

It remains to show that:

(1) Strict sequentiality is decidable, and

(2) Ordinary and strict sequentiality coincide for constructor bases.

3.2 The Decidability of Strict Sequentiality

Our decision procedure Check is given in *Algorithm 1*. Check uses the notation that for each $f \varepsilon \Sigma$, with arity k,

$z^f = f(x_1,..,x_k)$, all $x_i \varepsilon X$ are new.

$L^f = \{l \varepsilon L \mid z^f \leq l\}$ and $P^f = \{1,..,k\}$.

As the structure of Check indicates, strict sequentiality is a property of the group of equations defining each individual function. In fact, fcheck is simply an abstract version of a translator that transforms each such group into a *single* equation involving nested conditionals, selectors, and predicates. The path p chosen in each call of fcheck corresponds to the argument or part of argument whose structure needs to be queried next, given that the structure of the application of f discovered so far corresponds to z. The argument z plays no part in the decision procedure. Its presence is solely a device to facilitate statements and proofs of the properties of Check. an example of a base rejected by Check is found in *Example 9* below.

We first prove a few simple technical facts about Check. A call fcheck(L,P,z) is said to be *legitimate* iff it results from the call Check(L).

<u>*Proposition*</u> <u>15</u>: In any legitimate call fcheck(L,P,z) the

```
Function Check($\mathcal{L}$)
   Let $s_f$ = fcheck($\mathcal{L}^f$,$\mathcal{P}^f$,$z^f$)
   Return the conjunction of $s_f$, $f \in \Gamma$

Function fcheck(L,P,z)
  If $|L| \le 1$ Then Return True
  Else Let $Q = \{q \in P \mid q \notin \mathit{XPaths}(l), l \in L\}$
  If $Q = \phi$ Then Return False
  Else choose any $p \in Q$
    partition L into $L_c$, $c \in \Delta$
     where $l \in L_c$ iff $l/p \ge c(x_1,..,x_k)$, $k \ge 0$,
        and each $x_i$ is a new variable
    correspondingly, for each c,
    $P_c = P - \{p\} \cup \{p.j, 1 \le j \le k\}$
    $z_c = z[p \leftarrow c(y_1,...,y_k)]$
        where each $y_i$ is a new variable
    Return the conjunction of fcheck($L_c$,$P_c$,$z_c$), $c \in \Delta$
```

Algorithm 1

following statements hold:

(1) $P \subseteq \mathit{Paths}(l)$ for each $l \in L$.

(2) $z \le l$ for every $l \in L$.

(3) for each $l \in \mathcal{L}-L$, *UNIFY*(l,z) fails.

(4) $P=\mathit{XPaths}(z)$.

(5) $|L|>1$ implies $P \ne \phi$.

(6) if fcheck(L',P',z') is also a legitimate call, then

either *UNIFY*(z,z') fails, or z and z' are not independent.

Proof: Assertion (5) follows from (2) and (4). The proofs of the rest proceed by induction on the length of the calling chain resulting in the call fcheck(L,P,z).

Basis: The length is 1, and the call is fcheck(L^f,P^f,z^f) for some f. All assertions except (6) are obviously true. For (6), it is sufficient to note that if a call fcheck(L_1,P_1,z_1) results eventually from a call fcheck(L_2,P_2,z_2), then $z_1>z_2$, and also that *UNIFY*(z^f,z^g) fails whenever $f\neq g$.

Induction: The inductive assumption is that the assertions hold for all calls with calling chains of length $n\geq 1$. Suppose the call fcheck(L,P,z) has a chain of length n+1. Clearly, the call results immediately from another call with chain length n, and all assertions hold for the latter by the inductive assumption. The inductive step is now straightforward for all assertions except (6). For (6), the same observations as in the basis case suffice. ■

Proposition 16: Check(*L*) terminates.

Proof: Let zlength(z) = $\sum$length(p), $p\in\Sigma$*Paths*(z). By (2) in Proposition 15, zlength(z) has a finite upper bound in any legitimate call fcheck(L,P,z) unless $L=\phi$. Moreover, zlength(z_c)>zlength(z) by (4) in Proposition 15. Hence, each legitimate call fcheck(L,P,z) terminates, either because $L=\phi$ or because each chain of recursive calls issuing from it is bounded by the limit on the monotonic increasing quantity zlength(z). ■

The reason Check guarantees strict sequentiality is that, whenever we have a term t that is neither irreducible nor a redex, the path p chosen in the call fcheck(L,P,z) with the largest z compatible with t reaches a directly necessary subterm of t. This is the intuitive idea in the next Proposition.

*Proposition* _17_: Check(L), t$\notin NF_L$, and $\Lambda\notin RO$(t) implies DNO(t)$\neq\phi$.

*Proof*: Suppose the antecedents hold. If $L_t=\phi$, then condition (2) in the definition of DNO(t) is vacuous, and it is easy to see that in that case, RO(t)$\neq\phi$ implies DNO(t)$\neq\phi$. Suppose $L_t\neq\phi$. Let Z be the set of Δ-patterns u such that u$\leq$t and fcheck(M,R,u) is a legitimate call for some M and R. Since $L_t\neq\phi$, Z is clearly nonempty, and also finite. By (6) in Proposition 15, Z is linearly ordered, and hence contains a maximal element z, which occurs in some call fcheck(L,P,z).

Let Q = {qϵP | q$\notin$*XPaths*(l) for any lϵL}. By (3) in Proposition 15, L$\supseteq L_t$. By the maximality of z, Q$\subseteq$*XPaths*(*KNOWN*(t))=*PDNO*(t). By (1) in Proposition 15, and the above considerations, we have Q$\subseteq$*DNO*(t). It remains to show that Q$\neq\phi$. If |L|>1 then since Check(L) we have Q$\neq\phi$. Since $L_t\neq\phi$, the only other possibility is that L={l}. Since $\Lambda\notin RO$(t), l$\notin$*KNOWN*(t). Since P$\subseteq$*Paths*(l) and P=*XPaths*(z) by Proposition 15, if Q=ϕ then the only way z$\leq$l is if z=l. However, by (2) in Proposition 15, z$\leq$l, hence Q=ϕ implies

$z=1$ and $1\leq t$, contradicting the assumption that $\Lambda\notin RO(t)$. ■

It only remains to iterate this fact by structural induction.

Lemma 18: Check(L) implies L is strictly sequential.

Proof: We must show that whenever $RO(t)$ is nonempty, $NRO(t)$ is also nonempty. The proof proceeds by induction on the structure of t.

Basis: $t\in\Gamma$, where $RO(t)=NRO(t)=\{\Lambda\}$.

Induction: The inductive assumption is that the required property holds for all proper subterms of t. If t is a redex then $\Lambda\in NRO(t)$. Suppose it is not, and $r\in RO(t)$. By Proposition 17 above we have $DNO(t)\neq\phi$. Suppose $p\in DNO(t)$. If $RO(t/p)\neq\phi$ then by the inductive assumption $\exists q\in NRO(t/p)$, and $p.q\in NRO(t)$. If $RO(t/p)=\phi$ then let $s\in Paths(t)$ be the nonempty path such that $r=s.w$ and $s\in PDNO(t)$. By the inductive assumption $NRO(t/s)\neq\phi$, and therefore by (3) in Definition 13, $NRO(t)\neq\phi$. ■

We show that Check is a *complete* decision procedure by defining a function "strange" which constructs a counter example to prove that L is not strictly sequential, whenever Check(L) fails.

Definition 19: If a legitimate call fcheck(L,P,z) *directly* returns false, being unable to find a suitable $p\in P$, then strange(L,P,z) is a term defined as follows. Let $L = \{l_0,..,l_m\}$, $m\geq 1$. Partition P into $P_1,..,P_m$, such that $P_i \subseteq XPaths(l_i)$. This is possible by the failure condition in fcheck. Let $V_i=\{z/p \mid p\in P_i\}$. By (4) in Proposition 15, each $V_i \subseteq X$. Let $nx(i)=(i+1)\mathrm{mod}(m+1)$. Define a substitution α such that:

$\alpha(x)$ = if $x \varepsilon V_i$ then a simple ground instance of $l_{nx(i)}$
else x

where a simple ground instance of a term is an instance in which each variable is replaced by a normal form. Note that α is well-defined since V_i are disjoint by the construction of z. **strange(L,P,z)** $=_{\mathbf{def}}$ $z\alpha$. ■

Example 9.

Let *L* be as in *Example 8*. Clearly, fcheck(L^f,P^f,z^f) fails directly, and thus strange(L^f,P^f,z^f) is well defined. We have,

$P = P^f = \{1,2,3\}$, $z = z^f = f(Z0,Z1,Z2)$

$l_0 = f(X,a,b)$, $l_1 = f(b,X,a)$, $l_2 = f(a,b,X)$

Therefore, $P_0 = \{1\}$, $P_1 = \{2\}$, $P_2 = \{3\}$

$V_0 = \{Z0\}$, $V_1 = \{Z1\}$, $V_2 = \{Z2\}$

With straightforward choices for simple ground instances,

strange(L,P,z) = f(f(b,a,a),f(a,b,a),f(a,a,b)).

Proposition 20: Whenever w = strange(L,P,z) is well-defined,

(1) w is a ground term.

(2) $P = RO(w) \neq \phi$

(3) $DNO(w)=\phi$

(4) $NRO(w)=\phi$

Proof: (1) is a consequence of (4) in Proposition 15. The $P=RO(w)$ part of (2) merely asserts that $\Lambda \notin RO(w)$. To see this, note that since w is well-defined $|L|>1$. Moreover,

$\Lambda \varepsilon$ *RO*(w) would mean $\exists l \varepsilon L$ such that $l \leq z$, and by (2) in Proposition 15, l=z. This would imply, again by (2) in Proposition 15, that L is not an independent set, contradicting (1) in Definition 1. $P \neq \phi$ follows from (5) in Proposition 15 since $|L|>1$. By the construction of w, *KNOWN*(w)=z, hence by (2) and (3) in Proposition 15, $L=L_w$. We know that each $q \varepsilon$ *PDNO*(w)=*XPaths*(z) reaches a variable in some $l \varepsilon L = L_w$ by the construction of w, hence *DNO*(w)=ϕ. Since *PDNO*(w)=*XPaths*(z)=P=*RO*(w) by (4) in Proposition 15 and (2) above, *NRO*(w)=ϕ is obvious. ■

Lemma 21: Check(*L*)=false implies that *L* is not strictly sequential.

Proof: Check(*L*)=false implies that some legitimate call fcheck(L,P,z) fails directly, and therefore w=strange(L,P,z) is well-defined. Assertions (2) and (4) in Proposition 20 then imply that *L* is not strictly sequential. ■

This concludes the demonstration of the decidability of strict sequentiality.

Theorem 22: Check(*L*) iff *L* is strictly sequential, i.e., Check is a decision procedure for strict sequentiality of constructor bases.

Proof: Immediate consequence of Lemmas 18 and 21. ■

3.3 The Equivalence of Ordinary and Strict Sequentiality

The first part of this proof shows that every strictly sequential system is sequential. The proof proceeds in two steps. The first step is to show that all demand driven computations for normalizable terms are essentially the same. The second step is to show that the normal form equivalent of each normalizable term can be obtained by some demand driven computation. Together, the two steps show that *any* demand driven computation obtains the normal form equivalent of a normalizable term.

We first need to establish certain important properties of necessary redex occurrences:

(1) A necessary redex is always an outermost redex (Proposition 25),

(2) Necessary redex occurrences persist as such until they are reduced (Lemma 29), and

(3) In a strictly sequential system, the last redex to be reduced before reaching normal form is always a necessary one (Lemma 31).

Note that (3) is really a corollary of (2). In proving (1) and (2), we start with properties of *DNO*, and extend them to *NRO* by structural induction.

Proposition 23: $\Lambda \varepsilon RO(t)$ implies $DNO(t)=\phi$.

Proof: $\Lambda \varepsilon RO(t)$ implies $KNOWN(t) \geq l$ for some $l \varepsilon L_t$, therefore $PDNO(t) \cap \Sigma Paths(l)$ is clearly empty. ∎

Corollary 24: $\Lambda \ \varepsilon \ RO(t)$ implies $NRO(t) = \{\Lambda\}$. ∎

Proposition 25: If $p \varepsilon NRO(t)$ and $s \varepsilon RO(t)$ then $s \not< p$.

Proof: By induction on the structure of t. The basis case is that $t \varepsilon \Sigma$, which is trivial. For the inductive step, assume that the proposition holds for all proper subterms of t. There are three cases, according to Definition 13.

Case 1: $p=\Lambda$ or $p \varepsilon DNO(t)$. Recall that if $p \varepsilon DNO(t)$ then by

Proposition 23, $\Lambda\notin RO(t)$, hence $s\neq\Lambda$. The rest follows from the fact that $p\varepsilon PDNO(t)$.

Case 2: $q\varepsilon DNO(t)$, $p=q.r$, and $r\varepsilon NRO(t/q)$. We have $\Lambda\notin RO(t)$ as before. By Corollary 24, if $q\varepsilon RO(t)$ then $p=q$. Therefore, $s\neq\Lambda$ and $s\neq q$. The rest follows by the inductive assumption.

Case 3: Similar to Case 2. ■

Corollary 26: $NRO(t)\varepsilon ISRO(t)$. ■

Proposition 27: Suppose $p\varepsilon DNO(t)$, $A{:}t\xrightarrow{r}u$, and $p\neq r$, then $p\backslash A=\{p\}$, and $p\varepsilon DNO(u)$.

Proof: Since $DNO(t)\neq\phi$, by Proposition 23, $r\neq\Lambda$. By the antecedents and the definition of DNO, $r\not\leq p$, hence $p\backslash A = \{p\}$. Moreover, since $r\neq\Lambda$, $L_u\subseteq L_t$, and $p\varepsilon DNO(u)$. ■

Proposition 28: $p\varepsilon NRO(t)$, $A{:}t\xrightarrow{s}u$, and $p\neq s$ implies that $p\backslash A=\{p\}$ and $p\varepsilon NRO(u)$.

Proof: By Proposition 25, $s\not\leq p$, therefore $p\backslash A=\{p\}$, and moreover, $s\neq\Lambda$. $p\varepsilon NRO(u)$ is then straightforward by analysing the cases of Definition 13, using Proposition 27. ■

A sequence $B{:}t\xrightarrow{*}u$ is said to *preserve* $p\varepsilon RO(t)$ iff $p\backslash B=\{p\}$.

Lemma 29: Suppose $A{:}t\xrightarrow{*}u$ and $r\varepsilon NRO(t)$. Then either A preserves r and $r\varepsilon NRO(u)$, or $A=A_1.A_2.A_3$, where $A_1{:}t\xrightarrow{*}v$ preserves r, $A_2{:}v\xrightarrow{r}w$, and $A_3{:}w\xrightarrow{*}u$.

Proof: By induction on $|A|$. The basis $|A|=0$ is trivial. Suppose $|A|=n+1$. Let $A=A_1.A_2$, where $A_1{:}t\xrightarrow{s}t_1$ is the first reduction in the sequence. By Proposition 28, either $r=s$, or $r\backslash A_1=\{r\}$ and $r\varepsilon NRO(t_1)$. The rest follows by the inductive assumption. ■

Corollary 30: If $q\varepsilon NRO(t)$, $A:t\xrightarrow{Q}u$ is a multireduction, and $q\not\in Q$, then $q\backslash A=\{q\}$ and $q\varepsilon NRO(u)$. ■

Lemma 31: In a strictly sequential system, if $A:t\xrightarrow{r}u$ and $u\varepsilon NF_L$ then $r\varepsilon NRO(t)$.

Proof: By strict sequentiality, $NRO(t)$ is not empty. By Lemma 29, each member of $NRO(t)$ is either reduced or preserved, and since u is irreducible, it must be the former. ■

A reduction sequence $t_0\longrightarrow t_1\longrightarrow..\longrightarrow t_i\longrightarrow..$ is said to be *normal* iff each simple reduction is a necessary one, i.e., iff in each reduction $t_i\xrightarrow{s}t_{i+1}$ in the sequence, $s\varepsilon NRO(t_i)$. Λ is normal by convention. Normal sequences are simply the formal version of demand driven computations in our context. Using the properties of necessary sequences, we now establish the properties of normal sequences.

Proposition 32: If $A:t\xrightarrow{*}u$ is normal, $u\varepsilon NF_L$, $r\varepsilon NRO(t)$, and $t\xrightarrow{r}t'$, then there is a normal sequence $B:t'\xrightarrow{*}u$ such that $|B|=|A|-1$.

Proof: Since $u\varepsilon NF_L$, A does not preserve r. Therefore, by Lemma 29, $A=A_1.A_2.A_3$ such that $A_1:t\xrightarrow{*}v$ preserves r, and $A_2:v\xrightarrow{r}w$. Proceed by induction on $|A_1|$.

Basis: $|A_1|=0$. Trivial.

Induction: Assume for $|A_1|=n$. Suppose $|A_1|=n+1$. Let $A_1=A_{11}.A_{12}$ where $A_{11}:t\xrightarrow{*}v'$, $A_{12}:v'\xrightarrow{s}v$. Clearly, $|A_{11}|=n$. Since A_1 preserves r, $r\varepsilon NRO(v')$, and since A_1 is normal $s\varepsilon NRO(v')$, therefore r and s are independent. The order of reductions using r and s may therefore be reversed without affecting the result, i.e., $v' \xrightarrow{r} v'' \xrightarrow{s} w$. The rest follows by applying the inductive assumption to A_{11}. ∎

Proposition *33*: If $A{:}t \xrightarrow{*} u$ and $B{:}t \xrightarrow{*} w$ are normal sequences, $u \varepsilon NF_L$, and $|A|=|B|$, then $u=w$.

Proof: Straightforward by induction on $|A|$ using Proposition 32. ∎

Lemma *34*: If $A{:}t \xrightarrow{*} u$ and $B{:}t \xrightarrow{*} w$ are normal sequences in a strictly sequential system, and $u \varepsilon NF_L$, then there is a normal sequence D = B.C such that $|D|=|A|$, and $D{:}t \xrightarrow{*} u$.

Proof: By Proposition 33, $|B|>|A|$ is impossible, and whenever $|B|<|A|$, $w \not\varepsilon NF_L$, hence $NRO(w) \neq \phi$. B can therefore be extended as a normal sequence to the same length as A. Let this extension be C, and D=B.C. By Proposition 33, $D{:}t \xrightarrow{*} u$. ∎

Lemma 34 shows that all normal sequences for normalizable terms are essentially the same. The next Proposition contains our only real use of multireductions and the so-called parallel moves lemma (Lemma 8). The Proposition states that in a strictly sequential system, any sequence which starts with an arbitrary multireduction, and reaches normal form by a demand driven computation thereafter, can be rearranged to an entirely demand driven computation. The essential idea is to migrate the redices in the initial multireduction forward as residuals until they either become necessary or disappear. The parallel moves lemma guarantees that the migration can take place without changing the final result. The upper bound on the length of terminating normal sequences is crucial in ensuring that the migration process terminates.

Proposition 35: In a strictly sequential system, If $A_1:t\xrightarrow{S}u$ is a multireduction, $A_2:u\xrightarrow{*}y$ is normal, and $y\varepsilon NF_L$, then there is a normal sequence $B:t\xrightarrow{*}y$.

Proof: The proof proceeds by induction on $|A_2|$.

Basis: $|A_2|=0$. In this case, since S is an independent set, and hence may be reduced in any order, Lemma 31 implies that $S\subseteq NRO(t)$, and $B=A_1$.

Induction: Suppose $|A_2|=n+1$. We may assume without loss of generality that $S\cap NRO(t)=\phi$, since normal redices in S could be included in A_2. By strict sequentiality, $NRO(t)\neq\phi$. Let $r\varepsilon NRO(t)$. By Corollary 30, $r\backslash A_1=\{r\}$, and $r\varepsilon NRO(u)$. By Proposition 32, if $A_3:u\xrightarrow{r}u'$, there is $A_4:u'\xrightarrow{*}y$ such that A_4 is normal and $|A_4|=n$. Let $A_5:t\xrightarrow{r}t'$, $R=S\backslash A_5$, $P=R\cap NRO(t')$, and $Q=R-P$. By Lemma 8, $t'\xrightarrow{R}u'$. Therefore $t'\xrightarrow{P}t''$ and $A_6:t''\xrightarrow{Q}u'$. The rest follows by applying the inductive assumption to A_6 and A_4. ∎

Lemma 36: In a strictly sequential constructor system, if $u=NORM(t)$ then there is a normal sequence $B:t\xrightarrow{*}u$.

Proof: Suppose $A:t\xrightarrow{*}u$. If $|A|=0$ then B=A. Suppose $|A|>0$. Let A_2 be the longest normal suffix of A, and $A=A_1.A_2$. By Lemma 31, $|A_2|>0$. Proceed by induction on $|A_1|$.

Basis: $|A_1|=0$, $B=A_2=A$.

Induction: Assume the result for $|A_1|=n$. Suppose $|A_1|=n+1$. Let $A_1=A_{11}.A_{12}$, $A_{11}:t\xrightarrow{*}v$, $A_{12}v\xrightarrow{r}w$. By Proposition 35, there is a normal sequence $A_3:v\xrightarrow{*}u$. Applying the inductive assumption to A_{11} yields the rest. ■

N is said to be a *normal selection algorithm* iff whenever $NRO_L(t)\neq\phi$, $N(L,t)\varepsilon NRO_L(t)$, and $N(L,t)$ fails otherwise.

Lemma 37: Any normal selection algorithm N sequentializes *all* strictly sequential constructor bases.

Proof: Suppose L is strictly sequential, S is based on L and $u=NORM_S(t)$. By Lemma 36, there is a normal sequence $B:t\xrightarrow{*}u$. From Lemma 34, it follows that $|\sigma_{N,S}(t)|=|B|$, and $\sigma_{N,S}(t)\downarrow=u$. ■

Finally, if the decision procedure for strict sequentiality rejects a base, that base is necessarily nonsequential. The function strange is again used in constructing the necessary counter example.

Lemma 38: If L is a constructor base and Check(L)=false, then L is not sequential.

Proof (sketch): Check(L)=false implies that w=strange(L,P,z) is well-defined for some $L=\{l_0,..,l_m\}$, $P=P_0\cup..\cup P_m$, and z, where $m\geq1$. Now suppose A sequentializes L. Clearly, w can be normalized in some system based on L, hence $A(L,w)$ must succeed and return some path $q\varepsilon P$. Suppose $q\varepsilon P_i$. We now construct a system S based on L for which A is not safe. Let $k=(i+1)\bmod(m+1)$, and let $<l_k,l_k>\varepsilon S$. Since t/q is an instance of l_k, $\sigma_{A,S}(w)\uparrow$. By the construction of w, $q\varepsilon XPaths(l_i)$. Let $<l_i,r>\varepsilon S$, where r is any normal form. Clearly, w can be made an instance of l_i by suitably "filling in" S (including a suitable choice for r), since all instances of l_k in w are reached by paths that reach variables in l_i, and the right-hand sides corresponding to left-hand sides other than l_i and l_k are unconstrained given (3) in Definition 1. Thus, $r=NORM_S(w)$, and for A to be safe for S, we must have $\sigma_{A,S}(w)\downarrow=r$. ■ *Example 10* illustrates the construction suggested in the proof above.

Example 10.

Let w=strange(L,P,z) as constructed in *Example 9*.

$RO(w) = \{1,2,3\}$. Suppose $A_L(w)=1\varepsilon P_0$. Then i=0 and k=1.

The required counterexample *S* is:

f(X,a,b) = b, f(b,X,a) = f(b,X,a), f(a,b,X) = a

<u>*Theorem*</u> <u>*39*</u>: A constructor base is sequential iff it is strictly sequential.

<u>*Proof*</u>: Straightforward consequence of Theorem 22, and Lemmas 37 and 38. ■

<u>*Corollary*</u> <u>*40*</u>: Check is a decision procedure for the sequentiality of constructor bases. ■

<u>*Corollary*</u> <u>*41*</u>: Any normal selection algorithm sequentializes all sequential constructor bases. ■

4. Applications

The obvious application of the results of the previous section is an efficient evaluator for sequential FOES. A first effort in that direction is the (nondeterministic) normal selection algorithm Select given in *Algorithm 2*.

Although Select can be used to produce normal sequences

```
Function Select($L$,t)
  Return get($\phi$,{$\Lambda$},t)

Function get(L,P,t)
  Let Q = {$q \varepsilon P \mid q \not\varepsilon \mathit{XPaths}(l), l \varepsilon L$}
  If Q=$\phi$ Then Fail, Else Choose any p $\varepsilon$ Q
  Let u = t/p
  If u is a redex then return p
  Else If u = $g(u_1,..,u_k)$, $k \geq 0$, $g \varepsilon \Gamma$
       Then Let q = get($L^g$,$P^g$,u)
            If get is successful Then Return p.q
            Else Return get($\phi$,P-{p},t)
  Else Let u = $c(u_1,..,u_k)$, $k \geq 0$, $c \varepsilon \Delta$
       Return get($L_c$,$P_c$,t)
       where $L_c$, $P_c$ are defined as in fcheck
```

Algorithm 2

which are optimal in the abstract world where only the reduction steps in a noncopying reducer count, its actual use is obviously very expensive since the cost of *finding* a necessary redex far outweighs the cost of the reduction step. The search cost can be reduced substantially if the search does not begin at the root of the entire expression each time. Instead, once a directly necessary subexpression is identified, it should be worked on until it is reduced to a root-stable form. Such a strategy reduces search cost by increasingly localizing the search for the next redex. The function Evaluate in *Algorithm 3* realizes this modification. Note that Evaluate is *not* a selection algorithm, it is an actual evaluator that returns the normal form equivalent of the given expression. The sequential system involved is S.

```
Function Evaluate(t)                 Function Root_Stabilize(t)
 Let w = Root_Stabilize(t)               Let t = $f(t_1,..,t_n)$
 Let Q = $PDNO$(w)                       Return Spin($L^f$,$P^f$,t)
 For q$\varepsilon$Q Do
   w := w[q$\leftarrow$Evaluate(w/q)]
 Return w


      Function Spin(L,P,t)
          Let Q = {q$\varepsilon$P | q$\not\varepsilon$$XPaths$(l), l$\varepsilon$L}
          If Q=$\phi$ Then t=l$\alpha$, <l,r>$\varepsilon$$S$ (see Proposition 17)
                      Return Root_Stabilize(r$\alpha$)
          Else Choose some q $\varepsilon$ Q
               Let w = Root_Stabilize(t/q)
                     = $c(w_1,..,w_m)$
               Let v = t[q$\leftarrow$w]
               If c $\varepsilon$ $\Gamma$ Then Return v
               Else Return Spin($L_c$,$P_c$,v)
                 where $L_c$, $P_c$ are defined as in fcheck
```

Algorithm 3

The nondeterministic choice of an occurrence from set Q in both Root_Stabilize and Evaluate suggests the possibility of safe parallel evaluation. Safe parallelism is usually associated with the notion of strictness. Conventionally, the strictness of a function with respect to an argument is understood to mean that it is safe to reduce that argument to *normal form* before applying the function. In our case, a finer-grained notion of strictness is appropriate. A function is strict with respect to an argument in this sense if it is safe to reduce that argument to *root-stable form* before attempting to apply the function. This notion of strictness is applicable to parts of arguments as well. We believe that our notion of strictness and the parallelism it offers may be practically significant, since root-stabilization is clearly a much more substantial operation than single step reduction, coinciding with complete evaluation in the case of scalar types.

Other possible applications include translation of groups of equations to a single logically equivalent one, in preparation for optimization as described by Hudak (1984), and safe translation of lazy equational programs to logic programming languages such as PROLOG (Subrahmanyam and You, 1983). Our results can also be used to derive a *complete* "semantic unification algorithm" for use in integrating logic and functional programming as suggested by Subrahmanyam and You (1984).

Although the formal results presented in section 3 are applicable to constructor FOES only, they can be extended to cover all (regular) bases, using the simulation technique described by Thatte (1984). The basic idea of the technique is to treat the outermost function symbol in every root-stable term as a constructor. It is clearly possible to extend our Evaluate function to accommodate this change.

5. Related Work

Safe and optimal computations with recursive equations have been extensively investigated (Vuillemin, 1974, Berry and Levy, 1979, Cadiou, 1972, Wadsworth, 1971). Confluence and other properties of FOES are discussed by Huet (1977), O'Donnell (1977), and Rosen (1973). Much of this work has been influenced by the seminal study of confluence by Knuth and Bendix (1970). The issue of sequential evaluation with confluent FOES has been addressed by Hoffmann and O'Donnell (1979,1984), and by Huet and Levy (1979). Hoffmann and O'Donnell's methods are applicable to a restricted class of FOES for which a preorder examination of redices without backtracking is safe. Huet and Levy have independently arrived at results similar to ours for regular FOES using a much more abstract approach. In contrast to their work, we have chosen to tackle the simpler problem of constructor FOES using a direct constructive approach which we feel is more easily amenable to efficient implementation.

6. Conclusions

We have presented abstract algorithms for checking sequentiality, and for safe sequential, i.e., demand driven evaluation of expressions, in the context of constructor FOES. Clearly, much work remains to be done on methods for efficient implementation before the approach becomes a serious practical alternative. We believe that a suitable concrete implementation of sequential FOES must also be amenable to smooth integration with an implementation of a language with higher-order functions, since the two paradigms bring complementary strengths to applicative programming. We are currently investigating the possibility of using a generalization of the implementation technique based on combinators due to Turner (1979).

More work is also needed to explore the applications discussed in Section 4, especially the possibility of using our refined notion of strictness in parallel evaluation. Finally, our results need to be extended to cover all regular FOES.

I am grateful to John Wiersba for suggesting a crucial element of the current version of the decision procedure Check.

References

Ashcroft, E.A., and Wadge, W.W. (1977), "Lucid, a non-Procedural Language with Iteration," Communications of the ACM **20**(7).

Backus, John (1978), "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM **21**(8).

Barendregt, H.P. (1981), "The Lambda Calculus: Its Syntax and Semantics," North-Holland, Amsterdam.

Berry, G., and Levy, J-J. (1979), "Minimal and Optimal Computations of Recursive Programs," Journal of the ACM **26**(1).

Burstall, R.M., MacQueen, D.B., and Sanella, D.T. (1980), "HOPE: An Experimental Applicative Language," Tech. Rep. CSR-62-80, Univ. of Edinburgh, Scotland.

Cadiou, J-M. (1972), "Recursive Definitions of Partial Functions and Their Computations," Ph.D. Dissertation, Stanford Univ.

Curry, H.B., and Feys, R. (1958), "Combinatory Logic," North-Holland, Amsterdam.

Friedman, D.P., and Wise, D.S. (1976), CONS Should Not Evaluate Its Arguments, *in* "Proc. 3rd International Conference on Automata Languages and Programming," Univ. of Edinburgh, Scotland.

Goguen, J.A., and Tardo, J. (1979), An Introduction to OBJ: A Language for Writing and Testing Software Specifications, *in* "Proc. IEEE Conf. on Specifications of Reliable Software," pp170-189.

Hudak, P., and Kranz, D. (1984), A Combinator-based Compiler for a Functional Language, *in* "Proc. 11th ACM Symp. on the Principles of Programming Languages," Salt Lake City.

Henderson, P., and Morris, J.M. (1976), A Lazy Evaluator, *in* "Proc. 3rd ACM Symp. on the Principles of Programming Languages," Atlanta.

Hoffmann, C.M., and O'Donnell, M.J. (1979), An Interpreter Generator using Tree Pattern Matching, *in* "Proc. 5th ACM Symp. on the Principles of Programming Languages," San Antonio.

Hoffmann, C.M., and O'Donnell, M.J. (1982), "Programming with Equations," ACM TOPLAS Vol. 4(1).

Hoffmann, C.M., and O'Donnell, M.J. (1984), Implementation of an Interpreter for Abstract Equations, *in* "Proc. 11th ACM Symp. on the Principles of Programming Languages," Salt Lake City.

Huet, G. (1977), Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems, *in* "Proc. 18th IEEE Conf. on Foundations of Computer Science," Providence, RI.

Huet, G., and Levy, J-J. (1979), "Computations in Nonambiguous Linear Term Rewriting Systems," Tech. Rep. 359, INRIA, Le Chesney, France.

Kahn, G., and MacQueen, D.B. (1977), Coroutines and Networks of Parallel Processes, *in* IFIP 77 (B. Gilchrist, Ed.), North-Holland, Amsterdam.

Knuth, D.E., and Bendix, P.B. (1970), Simple Word Problems in Universal Algebras, *in* "Computational Problems in Abstract Algebra" (J. Leech, Ed.), Pergammon Press.

Knuth, D.E. (1973), "The Art of Computer Programming, Vol. 3, Sorting and Searching," Addison-Wesley.

Kowalski, R. (1979), "Logic for Problem Solving," North-Holland, Amsterdam.

MacQueen, D.B. (1981), Structure and Parameterization in a Typed Functional Language, *in* "Conf. Record of Symposium on Functional Languages and Computer Architecture," Lab. for Programming Methodology, Univ. of Goeteborg, Sweden.

Milner, R. (1978), "A theory of Type Polymorphism in Programming," Journal of Computer and Systems Sciences, Vol. 17, pp. 348-374.

O'Donnell, M.J. (1977), "Computing in Systems Described by Equations," Lecture Notes in Computer Science 58, Springer-Verlag.

Robinson, J.A. (1965), "A Machine-oriented Logic Based on the Resolution Principle," Journal of the ACM 12, pp. 23-41.

Rosen, B.K. (1973), "Tree-manipulating Systems and Church-Rosser Theorems," Journal of the ACM 20(1).

Subrahmanyam, P.A., and You, J-H. (1983), "FUNLOG = Functions + Logic: A Computational Model Integrating Logic Programming and Functional Programming," Tech. Rep. UTEC-83-040, Univ. of Utah.

Subrahmanyam, P.A., and You, J-H. (1984), Pattern Driven Lazy Reduction: a Unifying Evaluation Mechanism for Functional and Logic Programs, *in* "Proc. 11th ACM Symp. on the Principles of Programming Languages," Salt Lake City.

Thatte, S.R. (1984), "On the Correspondance Between Two Classes of Reduction Systems," to appear in Information Processing Letters.

Turner, D.A. (1976), "SASL Language Manual," Tech. Rep., St. Andrews Univ., UK.

Turner, D.A. (1979), "A New Implementation Technique for Applicative Languages," Software Practice and Experience, Vol. 9.

Vuillemin, J. (1974), "Correct and Optimal Implementations of Recursion in a Simple Programming Language," Journal of Computer and Systems Sciences, Vol. 9.

Wadsworth, C. (1971), "The Semantics and Pragmatics of the Lambda Calculus," D.Phil. Dissertation, Oxford University, England.