Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

OPTIMIZATION BY NON-DETERMINISTIC, LAZY REWRITING

Sanjai Narain

November 1988
CSD-880092

Optimization by non-deterministic, lazy rewriting

Sanjai Narain
Computer Science Department
University of California
Los Angeles, CA 90024
narain@cs.ucla.edu

ABSTRACT

Given a set S and a condition C we address the problem of determining which members of S satisfy C. One useful approach is to set up the generation of S as a tree, where each node represents a subset of S. If from the information available at a node, we can determine that no members of the subset it represents satisfy C, then the subtree rooted at it can be pruned, or not generated. Thus, large subsets of S can be quickly eliminated from consideration. We show how such a tree can be simulated by interpretation of non-deterministic rewrite rules, and its pruning simulated by lazy evaluation.

1.0 INTRODUCTION

Given a condition C, and a set S, the problem is to compute those members of S which satisfy C. The obvious way of solving it is to simply generate all members of S, and test which of these satisfy C. However, a much more effective approach can be the following: set up the generation of S as a tree, each node in which represents a subset of S. The subset represented by an internal node is the union of the subsets represented by each of its immediate descendants. External, or leaf nodes represent either empty sets, or singleton sets consisting of individual members of S. From the information available at a node N, check if any members of the set represented by N satisfy C. If not, prune, i.e. do not generate, the subtree rooted at N. Thus, large numbers of members in S can quickly be eliminated from consideration, particularly if trees are deep and pruning occurs near the root.

For example, S could be the set of all lists of the form [X1,..,Xm], where each Xi is in {0,1,..,9}. C could be the condition that the sum of all numbers in a list be equal to some fixed number K. We could set up the generation of S as a tree where each node is of the form [A1,A2,..,Ak], 0=<k=<m, each Ai in {0,..,9}, and its immediate descendants are the nodes of the form [A1,..,Ak,Ak+1], Ak+1 in {0,..,9}. The subset of S it represents is all lists whose first k elements are, respectively, A1,..,Ak. Clearly, if it can be determined that the sum of A1,..,Ak is greater than K, then the tree rooted at [A1,..,Ak] need not be generated. In particular, none of the 10**(m-k) subsets that [A1,..,Ak] represents, need be generated.

A convenient way to generate sets is via non-deterministic algorithms [Kowalski 1979, Sterling & Shapiro 1986]. A non-deterministic algorithm SA for generating S is, typically, a definition of what it means for an object to belong to S. From it, an interpreter for SA can automatically compute all members of S. Interpretation of SA can usually be laid out in the form of a tree. Nodes in this tree

Work supported in part by the Tangram project, DARPA contract F29601-87-C-0072.

represent unfinished computations. Information available at these is that gathered in the computation along the branch from the root to these.

Now, we can try to write SA in such a way that the tree resulting from its interpretation is similar in structure, and information content at nodes, to the original tree for generating S. For the purpose of tree-pruning we can try to use this tree in place of the original one. To ensure that whenever a new node is created in this tree, the condition C is evaluated, if possible, using the information available at this node, we need to suitably interleave steps in SA with those of CA, the algorithm implementing C.

For example, the set of all lists of the form [X1,..,Xm], each Xi in {0,..,9}, can be generated by the following non-deterministic Prolog program:

```
tuple([X1,X2,..,Xm]):-d(X1),d(X2),..,d(Xm).

d(0).
d(1).
d(2).
d(3).
d(4).
d(5).
d(6).
d(7).
d(8).
d(9).
```

Now the query tuple([Y1,..,Ym]), Y1,..,Ym variables, will halt with each instantiation of Y1,..,Ym to members of {0,1,..,9}. Moreover, it can be easily seen that the structure of the SLD-search tree for this query, and information available at its nodes (in the form of partial answer substitutions [Lloyd 1984]), are similar to those with the tree defined in the second paragraph above.

A Prolog program for checking whether the sum of all members of a list is equal to K is:

```
sum([],S,S).
sum([U|V],S,K):-A is U+S,A=<K,sum(V,A,K).
```

Now, the query sum(L,0,K) will succeed if the sum of members of L is equal to K, otherwise it will fail. The second rule says that the sum of members of [U|V], given S, is K provided U+S=<K, and the sum of members of V, given U+S, is K. Thus, if U+S>K, no further members of V need be considered.

Tree-pruning would occur if we could suitably interleave steps in these two programs. In particular, we need to ensure that whenever the tuple program instantiated a new variable in [X1,..,Xm], the variables already instantiated are immediately tested by sum, and moreover, that sum never instantiates any variables. Of course, it is clear that pruning would not occur with the simple-minded Prolog query:

tuple(X),sum(X,0,K).

as tuple would generate an entire tuple of m digits, before passing it on to sum.

There are three possible approaches for interleaving steps in SA with those in CA. First, we can combine SA and CA into a single algorithm, in particular, by explicitly programming the appropriate interleaving. However, from a software engineering point of view, it is highly desirable to keep SA and CA separate, i.e. think about them, and develop them independently of each other. If not, the combined algorithm can become quite complex, especially if SA and CA are complex.

Second, we can keep SA and CA separate but interleave their steps by connecting them via a facility for coroutining. For example, one can use the variable annotations (?,^) of IC-Prolog [Clark & McCabe 1979]. However, if SA and CA are complex, connecting them together may not be simple, as intricate knowledge of their execution may be required. Also, there do not seem to be efficient enough implementations of languages such as IC-Prolog, in which one can write non-deterministic programs such as tuple above, and also do coroutining.

Third, we can develop SA and CA separately, but in such a way that the interleaving is accomplished transparently, in the natural course of interpreting SA and CA, without our having to program it. This seems possible in languages such as LEAF [Barbuti et al. 1986], or FUNLOG [Subrahmanyam & You 1984] which combine logic programming with lazy rewriting. A similar possibility also exists in Prologs which perform intelligent backtracking [Kumar & Lin 1988, Bruynooghe & Pereira 1984, Chang & Despain 1984]. However, implementations of these languages, efficient enough for practical programming, seem to be still under development.

We propose a realization of the third approach within the framework of rewriting. Specifically, we propose F*, a first-order, lazy non-deterministic rewrite rule system [Narain 1988]. We show how SA and CA can be developed independently of each other in F*, yet steps in them interleaved by non-deterministic, lazy rewriting in such a way that tree-pruning is accomplished.

F* programs can be compiled into Horn clauses in such a way that when SLD-resolution interprets these it directly simulates the behavior of the lazy F* interpreter. In particular, the non-determinism of F* is mapped on to the non-determinism of Horn clauses. Due to the nature of the clauses obtained by compilation, outlined in APPENDIX, we effectively obtain a lazy interpreter which operates at roughly the same speed as does Prolog. For problems in which lazy evaluation does not reduce lengths of computation, F* is somewhat slower than Prolog. Otherwise, F* is faster than Prolog by unbounded, even infinite amounts. Thus, F* seems to be efficient enough for practical programming.

Intuitively, tree-pruning is achieved in F* as follows: Let E be a ground term. In F*, as with other rewrite rule systems, there can be more than one reduction starting at E. However, in F* there can also be more than one *lazy* reduction starting at E. Due to non-determinism, E can possess more than one normal form. A *reduction-completeness* theorem states that each of these normal forms can be computed by generating all the lazy reductions starting at E. These can all be laid out in the form of a *lazy reduction-tree*. This tree is analogous to an SLD-search tree [Lloyd 1984], while the lazy reduction strategy is analogous to SLD-resolution.

Given the set S, we can define a non-deterministic algorithm in F* such that some ground term E possesses as normal forms, all the members of S. Moreover, we define this algorithm in such a way that the lazy reduction-tree rooted at E is similar to that we have in mind for generating S, and upon which we wish to perform the pruning.

Now, for conditions such as C, we take the point of view that *when they hold for objects, they return, not the truth-value true, but the objects themselves. Furthermore, if they do not hold, they do not return anything, but simply fail.* Thus, where CA is the function symbol defining condition C in F* with this point of view, the normal forms of the term CA(E) represent the members of S which satisfy C.

Given CA(E), the F* interpreter generates the lazy reduction-tree rooted at CA(E). *Due to laziness, reduction of* E *is interleaved with evaluation of* CA. In particular, whenever new information about E becomes available, an attempt is automatically made to evaluate CA. If at a node it is discovered that CA does not hold, no further descendants of it are generated. Thus, the tree-pruning we seek is automatically achieved. In particular, the lazy reduction tree rooted at CA(E) is, usually, much smaller than that rooted at E.

This paper discussess solutions to five problems illustrating the above idea. These have all been implemented and tested in the LOG(F) system, which is simply a logic programming system augmented with an F* compiler. Performance figures obtained for these seem to compare favorably with those obtained by languages such as CHIP [Dincbas et al. 1988, van Hentenryck & Dincbas 1987], which contain specialized capabilities for solving combinatorial problems.

F* can be viewed as a way of bringing to rewriting, one of the most powerful features of logic programming: the ability to develop non-deterministic algorithms. In this it is similar to EqL [Jayaraman & Gupta 1987]. Due to the compilation of F* in Prolog, it can also be viewed as a means of doing lazy evaluation in Prolog. Before we discuss the problems solved by our approach, we first define the F* system.

2.0 DEFINITION OF F*

F* is intended mainly for lazy reduction of ground terms. Nevertheless, it forms a sufficient basis for functional programming.

Function symbols are partitioned, *in advance*, into constructors and non-constructors. Thus, we do *not* adopt the convention e.g. [van Emden 1987] that any function symbol not defined by a program is a constructor symbol. For example, 0, 1, 2, 3.1415,..., true, false, [] are 0-ary constructor symbols and | a 2-ary constructor symbol.

A **term** is either a variable, or an expression of the form f(t1,..tn) where f is an n-ary function symbol, and each ti is a term. A ground term E is said to **match** another term F, with substitution α, if E=Fα. A **reduction rule** is of the form LHS=>RHS, where LHS and RHS are terms, satisfying the following restrictions:

> (a) LHS is of the form f(L1,..,Lm), f an m-ary non-constructor function symbol, and each Li either a variable, or a term of the form c(X1,..,Xn), c an n-ary constructor symbol, and

each Xi a *variable.*

(b) A variable occurs at most once in LHS.

(c) All variables of RHS occur in LHS.

Note that reduction rules with left-hand-sides of arbitrary depth can easily be expressed in terms of rules with left-hand-sides of depth at most two, as required by (a). For example, fib(s(s(X)))=>plus(fib(X),fib(s(X))) can be expressed as fib(s(A))=>g(A), g(s(X))=>plus(fib(X),fib(s(X)). An **F* program** is a set of reduction rules. Some examples of F* programs are:

```
append([],X)=>X
append([U|V],W)=>[U|append(V,W)]

int(N)=>[N|int(s(N))].

merge([A|B],[C|D])=>if(lesseq(A,C),[A|merge(B,[C|D])],[C|merge([A|B],D)]).

if(true,X,Y)=>X.
if(false,X,Y)=>Y.
```

Note that there is no restriction that F* programs be *Noetherian*, or even *confluent*. Thus infinite structures can be freely defined and manipulated in F*. Also, terms can be simplified in more than one way, a fact which we exploit for implementing tree-pruning.

Let P be an F* program and E and E1 be ground terms. We say $\mathbf{E}=>_P\mathbf{E1}$ if there is a rule LHS=>RHS in P such that E matches LHS with substitution σ and E1 is RHSσ. Where E,F,G,H, are ground terms, let F be the result of replacing an occurrence of G in E by H. Then we say **F=E[G/H]**. Let P be an F* program and E a ground term. Let G be a subterm of E such that $G=>_PH$. Let E1=E[G/H]. Then we say that $\mathbf{E}->_P\mathbf{E1}$. $-*>_P$ is the reflexive-transitive closure of $->_P$. The subscript P is dropped if clear from context.

A ground term is said to be in **simplified form**, or simplified, if it is of the form c(t1,..,tn) where c is an n-ary constructor symbol, n>=0, and each ti is a ground term. F is called a simplified form of E if E-*>F and F is in simplified form. Simplified forms can be used to represent finite approximations to infinite structures. For example, [0|int(s(0))] is a simplified form, and is a finite approximation to [0,s(0),s(s(0)),..].

A ground term is said to be in **normal form** if each function symbol in it is a constructor symbol. F is called a normal form of E if E-*>F and F is in normal form. Note that this notion of normal form is different from the usual one which has to do with non-reducibility. This does not lead to any loss of generality, at least for programming purposes. Moreover, it allows us to define the notion of *failure form* below, which is useful for tree-pruning.

Let P be an F* program. A **reduction** in P is a, possibly infinite, sequence E1,E2,E3... such that for each i, $Ei->_PEi+1$. A **successful reduction** in P is a reduction E0,..,En, n>=0, in P, such that En is

simplified.

Let P be an F* program. We now define a reduction strategy, $\mathbf{select}_P$ for P. Informally, given a ground term E it will select that subterm of E whose reduction is necessary in order that some => rule in P apply to the whole of E. *In this, is implicit its laziness.* The relation $select_P$, whose second argument is the subterm selected from E, is defined by the following pseudo-Horn clauses:

$select_P$(E,E) if $E=>_P X$.
$select_P$(E,X) if
    E=f(T1,..,Ti,..,Tn), and
    there is a rule f(L1,..,Li,..,Ln)=>RHS in P, and
    Ti does not match Li, and
    $select_P$(Ti,X).

The first rule states that if E is the given ground term, and there exists another term X such that E=>X, then E itself can be selected from E.

In the second rule, E=f(T1,..,Tn), and there is some rule f(L1,..,Ln)=>RHS, such that for some i, Ti in T1,..,Tn does not match Li in L1,..,Ln. In order to reduce E by this rule, it is necessary to reduce Ti. Thus, a term in Ti must be recursively selected for reduction. This rule is a schema, so that an instance of it is assumed written for each 1=<i=<n, and each non-constructor function symbol f. For example, where P is the set of reduction rules which appear above, we have the following:

select(merge(int(1),int(2)),int(1)).
select(merge(int(1),int(2)),int(2)).
select(merge([1,3],int(2)),int(2)).
select(merge([1,2],[3,4]),merge([1,2],[3,4])).
If E=[1|merge(int(1),int(2))] then select is undefined for E.

Note that select is non-deterministic, in that given E, there can be more than one F, such that select(E,F). Thus, starting at E, there can be more than one reduction computed by select. Also, select is more general than a leftmost-outermost strategy. For example, from the rules:

f(X,[])=>[].
a=>a.
b=>[].

the only leftmost-outermost reduction of f(a,b) is f(a,b),f(a,b),f(a,b),... However, repeated application of select yields only the finite reduction f(a,b),f(a,[]),[].

Let P be an F* program and E,G,H be ground terms. Suppose $select_P$(E,G) and $G=>_P H$. Let E1 be the result of replacing G by H in E. Then we say that E reduces to E1 in an **N-step** in P. The prefix N in N-step is intended to connote normal order. Let P be an F* program. An **N-reduction** in P is a reduction E1,E2,.... in P such that for each i, Ei reduces to Ei+1 in an N-step in P. We now have:

Theorem 1. Reduction-completeness of F* for simplified forms. Let P be an F* program and D0 a ground term. Let D0,D1,..,Dn be a successful reduction in P. Then there is a successful N-

reduction D0,E1,..,Em in P, such that Em-*>Dn.

Its proof can be found in [Narain 1988]. It states that to reduce a ground term to a simplified form it is sufficient to generate only the N-reductions starting at it.

Note that if a term E is already simplified, e.g. E=[1|append([],[])], then select is undefined for E. Thus, an N-reduction ending at E cannot be extended further. If we wish to compute normal forms of E, we need a reduction strategy more general than select. It turns out that this can be based upon repeated application of select. Specifically, where P is an F* program, we define a reduction strategy **select-r$_P$**, where r stands for repeated or recursive, by the following pseudo-Horn clauses:

select-r$_P$(E,F) if select$_P$(E,F).
select-r$_P$(c(T1,..,Ti,..,Tm),F) if
    c is a constructor symbol, and
    select-r$_P$(Ti,F).

Thus, select-r is like select except that if a ground term is in simplified form, it recursively calls select on one of the arguments of the outermost constructor symbol. So, its repeated use can yield normal-forms of ground terms. Again, the second rule is a schema so that an instance of it is assumed written for each 1=<i=<m, and each constructor symbol c.

Let P be an F* program and E,G,H be ground terms. Suppose select-r$_P$(E,G) and G=>$_P$H. Let E1 be the result of replacing G by H in E. Then we say that E reduces to E1 in an **NR-step** in P. NR is intended to connote normal-repeated.

Let P be an F* program. An **NR-reduction** in P is a reduction E1,E2,.... in P such that for each i, Ei reduces to Ei+1 in an NR-step in P. We now have:

Theorem 2. Reduction-completeness of F* for normal forms. Let P be an F* program and D0 a ground term. Let D0,D1,..,Dn be a reduction in P, where Dn is in normal form. Then there is an NR-reduction D0,E1,..,Em=Dn, in P.

Again, its proof can be found in [Narain 1988]. It states that to compute normal forms of a term, it is sufficient to generate only the NR-reductions starting at it. Of course, not all N- or NR-reductions are finite, even when normal forms exist. For example, with the program:

a=>a.
a=>[].
f([])=>[].

There is an infinite N-reduction f(a),f(a),f(a),.... However, there is also a finite N-reduction f(a),f([]),[]. Thus, some searching among alternative N- or NR-reductions may be required to compute simplified, or normal forms. However, laziness ensures that search paths are cutoff as soon as possible. In [Narain 1988] is studied D(eterministic)F*, a restriction of F*, in which simplified, or normal forms may be computed without any search. However, in DF*, every term has at most one normal form, and so it is not relevant to this paper.

Let P be an F* program, and E a ground term. An **NR-tree** rooted at E is constructed as follows: E is the root node. The immediate descendants of any node Q in the tree are Q0,Q1,..,Qk, where Q reduces to each Qi in an NR-step. NR-trees are also called **lazy reduction-trees.**

Let P be an F* program and E a ground term. E is said to be a **failure form** if E is not a normal form, and E cannot be reduced to another term in an NR-step in P. Note that E may still be reducible. Failure forms are analogous to failure nodes in SLD-search trees [Lloyd 1984], while normal forms are analogous to success nodes (or empty goals). If while generating a lazy reduction-tree, a failure form is encountered, the interpreter backs up and generates other parts of the tree. *It is for this reason that normal forms are not defined in terms of non-reducibility.*

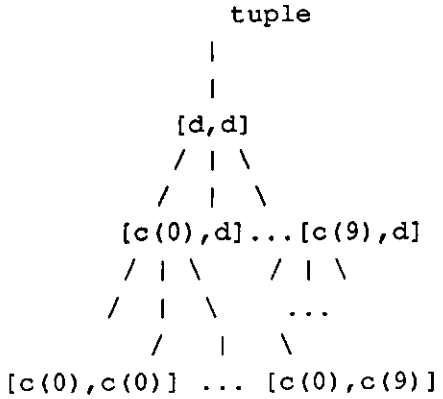
3.0 TUPLE SUM

We now show how the example in Section 1.0 can be programmed in F* to achieve the type of tree-pruning we desire. An F* program to compute the set S of all lists [X1,..,Xm], each Xi in {0,..,9} can be:

```
tuple=>[d,d,d,..,d].

d=>c(0).
d=>c(1).
d=>c(2).
d=>c(3).
d=>c(4).
d=>c(5).
d=>c(6).
d=>c(7).
d=>c(8).
d=>c(9).
```

Here c, !, [], and 0,1,..,9 are constructor symbols, while d and tuple are non-constructor symbols. Now, the term tuple, whose right hand side contains m ds, possesses 10**m normal forms, each of the form [c(X1),c(X2),..,c(Xm)], Xi in {0,..,9}. Each normal form can be taken to represent a member of S. Assuming that m=2, a subtree of the lazy reduction-tree rooted at tuple is:



Each internal node, except the root, has ten descendants. Each tip is of the form [c(X1),c(X2)], Xi in {0,..,9}. This tree can be seen to be similar to that defined in the second paragraph of Section 1.0.

We now define the condition that the sum of all members of a list be equal to K. *Again, we do this in such a way that when it hold for a list, it returns, not true, but the list itself. If it does not hold, it does not return anything, but simply fails.* A satisfactory definition is:

```
sum_eq([],c(S),c(K))=>cond(equal(S,K),[]).
sum_eq([c(U)|V],c(S),c(K))=>cond((U+S)=<K,[c(U)|sum_eq(V,c(S+U),c(K))]).

cond(true,X)=>X.
```

These rules exactly parallel the two Prolog clauses for sum, in Section 1.0. Here, true is a constructor symbol (along with [], |, and c), and sum_eq, cond, equal, +, and =< are non-constructor symbols. The last three are F* primitives and are evaluated eagerly. Now, where m=2, the term sum_eq(tuple) possesses as the only normal form, the term [c(0),c(0)]. Effectively, sum_eq is evaluated as soon as enough information becomes available about its first argument. If it does not hold, then the rest of the argument is not evaluated further. In particular, one branch in the lazy reduction-tree rooted at sum_eq(tuple,c(0),c(0)) is:

```
sum_eq(tuple,c(0),c(0))
     |
sum_eq([d,d],c(0),c(0))
     |
sum_eq([c(0),d],c(0),c(0))
     |
cond(0=<0,[c(0)|sum_eq([d],c(0),c(0))])
     |
cond(true,[c(0)|sum_eq([d],c(0),c(0))])
     |
[c(0)|sum_eq([d],c(0),c(0))]
     |
[c(0)|cond(0=<0,[c(0)|sum_eq([],c(0),c(0))])]
     |
[c(0)|cond(true,[c(0)|sum_eq([],c(0),c(0))])]
     |
[c(0),c(0)|sum_eq([],c(0),c(0))]
     |
[c(0),c(0)|cond(0=0,[])]
     |
[c(0),c(0)|cond(true,[])]
     |
[c(0),c(0)]
```

Some, minor steps involving + have been omitted. However, all other branches in this tree rapidly reach failure forms. For example:

```
sum_eq(tuple,c(0),c(0))
     |
sum_eq([d,d],c(0),c(0))
     |
sum_eq([c(1),d],c(0),c(0))
     |
cond(1=<0,[c(1)|sum_eq([d],c(0),c(0))])
     |
cond(false,[c(1)|sum_eq([d],c(0),c(0))])
```

Thus, the lazy reduction tree T2 rooted at sum_eq(tuple,c(0),c(0)) is much smaller than the tree T1, rooted at tuple, as was our objective. In particular, while T1 had 100 leaves, T2 has only 19. The gap widens as larger tuples are considered (for the same value of K).

Moreover, we are also able to define the generation algorithm, tuple independently of the testing algorithm, sum_eq. The interleaving between these two is naturally accomplished in the process of generating the lazy reduction-tree.

A similar program has been developed for computing all the ways in which a dollar can be changed using half-dollars, quarters, dimes, nickels, and pennies [Polya 1973].

4.0 SUBSET SUM

Subset sum is problem SP13 in Appendix: A list of NP-complete problems, [Garey & Johnson 1979]. Given a finite set A, and size $s(a) \in Z^+$ for each $a \in A$, and a positive integer K, the problem is to determine whether there is a subset $A1 \subseteq A$ such that the sum of the sizes of the elements in A1 is exactly K. We consider a special case of this problem where A is a list of positive integers, and s is the identity function. Sets are represented as lists. Subsets can be computed non-deterministically by the following F* program:

```
subset([])=>[].
subset([U|V])=>[U|subset(V)].
subset([U|V])=>subset(V).
```

Now the term subset([c(1),c(2),c(3)]) possesses as normal forms each of the eight subsets of [c(1),c(2),c(3)]. Each node Q in the lazy reduction-tree rooted at subset([a1,a2,..,am]) is of the form [U1,..,Un|subset([A1,..,Ap])], where each Ui and Aj is in {a1,..,am}, 0=<n,p=<m. The number of normal forms appearing at the leaves of the subtree rooted at Q is 2**p. The program to test whether sum of numbers in a subset is equal to K, is exactly the same as in the previous section.

In the lazy reduction-tree for sum_eq(subset([a1,a2,..,am]),c(0),c(K)), subset([a1,..,am]) is reduced to [U1,..,Un|subset([A1,..,Ap])], and whenever the sum of U1,..,Un is greater than K, none of the 2**p subsets of the form [U1,..,Un|X] are generated.

5.0 N-QUEENS

The problem is to place N queens on an NxN chess board so that no two queens attack each other. It

is easily seen that each queen must be in a distinct row and column, so that candidates for solutions can be represented by permutations of the list [1,2,..,N]. The position of the ith queen in a permutation p is [i,q] where q is is the ith element of p. The problem now reduces to generating all permutations of [1,2,..,N] and testing whether they are safe, or represent a solution. Permutations can be generated by the following F* program:

```
perm([])=>[].
perm([U|V])=>insert(U,perm(V)).

insert(U,X)=>[U|X].
insert(U,[A|B])=>[A|insert(U,B)].
```

In the lazy reduction-tree rooted at perm([1,..,N]) if there is a node Q of the form [U1,..,Up|Z], p>=1, each Ui in {1,..,N}, and Z unsimplified, then the subtree rooted at Q contains (N-p)! leaves, each representing a permutation of the form [U1,..,Up|_]. If at Q it is determined that [U1,..,Up] already form an unsafe configuration, then none of these (N-p)! permutations need be generated.

The condition safe, as usual, is defined in such a way that if it holds for a list, it returns that list itself:

```
safe([])=>[].
safe([U|V])=>[U|safe(nodiagonal(U,V,1))].

nodiagonal(U,[],N)=>[].
nodiagonal(U,[A|B],N)=>cond(noattack(U,A,N),[A|nodiagonal(U,B,N+1)]).

noattack(U,A,N)=>neg(equal(abs(U-A),N)).

cond(true,X)=>X.
```

Here neg, equal, and abs are F* primitives computing, respectively, logical inversion, syntactic equality, and real number modulus. Finally, queens can be computed by:

```
queens(X)=>safe(perm(X)).
```

In particular, queens([1,2,3,4]) yields [2,4,1,3], and [3,1,4,2].

6.0 SEND+MORE=MONEY

The problem is to assign the variables S,E,N,D,M,O,R,Y to distinct values in {0,1,..,9} so that SEND+MORE=MONEY is a correct equation, where for example, SEND is interpreted as 1000*S+100*E+10*N+D. Clearly, there are 10p8 assignments.

The main challenge here is designing an appropriate condition which, given an assignment A to the first k variables, can determine that A *cannot* be part of any correct assignment to all 8 variables. If the condition is simply to find the values of SEND, MORE, and MONEY, and check the addition, then it requires complete assignment before evaluation. This degenerates to checking each of the 10p8 assignments, and there is no tree-pruning.

A suitable condition is based upon checking the addition from right to left, the way a child would do it. In particular, given an assignment, check whether the list [D,E,Y,N,R,E,E,O,N,S,M,O,0,0,M] represents a correct sum, in that D+E yields Y as sum and the carry+N+R yields E as sum, and so on. Now, for example, all assignments with D=0,E=1,Y=2 can be discarded, and substantial pruning can take place. The following four F* rules construct the above list:

```
form_1([c(D),c(E),c(Y)|Z])=>[D,E,Y|form_2(D,E,Y,Z)].

form_2(D,E,Y,[c(N),c(R)|Z])=>[N,R,E|form_3(D,E,Y,N,R,Z)].

form_3(D,E,Y,N,R,[c(O)|Z])=>[E,O,N|form_4(D,E,Y,N,R,O,Z)].

form_4(D,E,Y,N,R,O,[c(S),c(M)|Z])=>[S,M,O,0,0,M].
```

Function admit below, takes as input, an initial carry of 0, and the above list, checks whether they represent a correct sum, and if so, return the list as output. Functions sum and carry are F* primitives yielding the obvious results:

```
admit(_,[])=>[].
admit(C,[A,B,D|Z])=>cond(equal(sum(C,A,B),D),[A,B,D|admit(carry(C,A,B),Z)]).

cond(true,X)=>X.
```

The program below computes permutations of a list of L items, taken N at a time, such that each item in a permutation is distinct. If N is zero, there is a single permutation []. Otherwise, some element from L is removed, and put at the front of a permutation of the resulting list of items, taken N-1 at a time. Note that the conditional function, if, is 3-ary, in contrast to cond, which was binary. The symbol pair is a binary constructor.

```
npr(L,N)=>if(equal(N,0),[],npr_aux(remove(L),N)).

npr_aux(pair(U,V),N)=>[U|npr(V,N-1)].

remove([U|V])=>pair(U,V).
remove([U|V])=>remove_aux(U,remove(V)).

remove_aux(U,pair(A,B))=>pair(A,[U|B]).

if(true,X,Y)=>X.
if(false,X,Y)=>Y.
```

Finally, instantiations of [D,E,Y,N,R,E,E,O,N,S,M,O,0,0,M] such that it is a correct sum are obtained by computing normal forms of soln:

```
soln=>admit(0,form_1(npr(digits,8))).

digits=>[c(0),c(1),c(2),c(3),c(4),c(5),c(6),c(7),c(8),c(9)].
```

7.0 ZEBRA PUZZLE

This is a well known problem, but we have taken it from [Dincbas et al. 1987], and our formulation of it for computer solution is essentially theirs. Of course, we use lazy rewriting, whereas they use logic programming augmented with a technique called forward checking. There are five houses in a row. With each house is associated a distinct color, nationality, profession, animal, and drink. The question is, who owns the zebra. More generally, it is to find the associations given the following constraints:

The Englishman lives in a red house.
The Spaniard owns a dog.
The Japanese is a painter.
The Italian drinks tea.
The Norwegian lives in the first house on the left.
The owner of the green house drinks coffee.
The green house is on the right of the white one.
The sculptor breeds snails.
The diplomat lives in the yellow house.
Milk is drunk in the middle house.
The Norwegian's house is next to the blue one.
The violinist drinks fruit juice.
The fox is in the house next to that of the doctor.
The horse is in the house next to that of the diplomat.

We draw the following 5x5 table:

Zebra Variables					
	1	2	3	4	5
N ation	England	Spain	Japan	Italy	Norway
C olor	green	red	yellow	blue	white
P rofession	painter	diplomat	violinist	doctor	sculptor
A nimal	dog	zebra	fox	snails	horse
D rink	juice	water	tea	coffee	milk

There are 25 variables, N1,..,N5, C1,..,C5, P1,..,P5, A1,..,A5, D1,..,D5, each standing for one of the 25 entries in the table. Each ranges over {1,..,5}. For example if N5=k, then Norwegian has house number k. Similarly, if C3=j, the color yellow is associated with house number j. Now, for example, the fact that the Englishman lives in a red house can be expressed as N1=C2. The above fourteen constraints can now be expressed as follows:

N1=C2,N2=A1,N3=P1,N4=D3,N5=1,D5=3,P3=D1,C1=D4,P5=A4,P2=C3,plusc(C1,C5,1), plusorminus(A3,P4,1),plusorminus(A5,P2,1),plusorminus(N5,C4,1).

where plusc(X,Y,C) means X is Y+C, and plusorminus(X,Y,C) means either X is Y-C, or X is Y+C.

In F*, we can represent the above table by a list of 5 lists, each containing five numbers, representing values of the 25 variables. To refer to any of these we can write the following five selector functions:

```
n(N,[Ns,Cs,Ds,Ps,As])=>nth(N,Ns).
c(N,[Ns,Cs,Ds,Ps,As])=>nth(N,Cs).
p(N,[Ns,Cs,Ds,Ps,As])=>nth(N,Ps).
d(N,[Ns,Cs,Ds,Ps,As])=>nth(N,Ds).
a(N,[Ns,Cs,Ds,Ps,As])=>nth(N,As).

nth(N,[X|Y])=>if(equal(N,1),t(X),nth(N-1,Y)).

if(true,X,Y)=>X.
if(false,X,Y)=>Y.
```

As usual, we will express constraints in such a way that if they hold, they return the table itself, instead of truth. Where CL is the above 5x5 table, and true, false, t, and c are constructor symbols, the last four constraints can be written as:

```
c_1(CL)=>cond(plusc(c(1,CL),c(5,CL),t(1)),CL).
c_2(CL)=>cond(plus_or_minus(a(3,CL),p(4,CL),t(1)),CL).
c_3(CL)=>cond(plus_or_minus(a(5,CL),p(2,CL),t(1)),CL).
c_4(CL)=>cond(plus_or_minus(n(5,CL),c(4,CL),t(1)),CL).

cond(true,X)=>X.

plusc(t(X),t(Y),t(C))=>if(equal(X,Y+C),true,false).
plus_or_minus(t(X),t(Y),t(C))=>or(equal(X,Y-C),equal(X,Y+C)).

or(true,true)=>true.
or(true,false)=>true.
or(false,false)=>false.
or(false,true)=>true.
```

We must now express the first 10 constraints. Clearly, we could express the first and fifth by, respectively:

```
c_5(X)=>cond(eqt(n(1,X),c(2,X)),X).
c_9(X)=>cond(eqt(n(5,X),t(1)),X).

eqt(t(X),t(Y))=>if(equal(X,Y),true,false).
```

Unfortunately, if all ten constraints are expressed this way, extreme inefficiency results. On the other hand constraints like these can be enforced extremely efficiently in Prolog: if there occur variables V1 and V2 in some structure S, and we wish to declare that V1 and V2 are the same, we can simply replace V1 by V2 in S, or vice versa.

We assume the existence of an F* primitive inst(L,B), where L is a list containing some variables, and B is a constant list. This can easily be defined in our implementation of F*. The output of inst is an instantiation of variables of L to distinct numbers in {1,..,5}, and distinct from the numbers in B. inst can enumerate all possible instantiations. Thus, we can now represent the table as follows:

```
[inst([N1,N2,N3,N4,N5],[]),
 inst([C1,C2,C3,C4,C5],[]),
 inst([D1,D2,D3,D4,D5],[]),
 inst([P1,P2,P3,P4,P5],[]),
 inst([A1,A2,A3,A4,A5],[])].
```

Clearly, the total number of instantiations are (5!)**5 or about 25 billion. The first 10 constraints can now be expressed simply by changing certain of these variables, so that the new table is:

```
[inst([N1,N2,N3,N4,1],[]),
 inst([C1,N1,P2,C4,C5],[]),
 inst([P3,D2,N4,C1,3],[]),
 inst([N3,P2,P3,P4,P5],[]),
 inst([N2,A2,A3,P5,A5],[])].
```

For example, N1=C2 is expressed by replacing C2 by N1, and N5=1 is expressed by replacing N5 by 1. Of course, we are departing from the framework of F* in that we are reducing non-ground terms. However, the gains are considerable, so reluctantly we propose this approach, while continuing to think of how to express such constraints efficiently within F*. Of course, this example still illustrates the basic thesis that non-deterministic, lazy rewriting can prune search spaces of problems of the form "given set S, and condition C, find those members of S which satisfy C". If we now define:

```
soln=>c_2(c_3(c_4(c_1([inst([N1,N2,N3,N4,1],[]),
                       inst([C1,N1,P2,C4,C5],[]),
                       inst([P3,D2,N4,C1,3],[]),
                       inst([N3,P2,P3,P4,P5],[]),
                       inst([N2,A2,A3,P5,A5],[])])))).
```

and obtain normal forms of soln, we obtain the following associations:

Zebra solution				
1	2	3	4	5
Norway	Italy	England	Spain	Japan
yellow	blue	red	white	green
water	tea	milk	juice	coffee
diplomat	doctor	sculptor	violinist	painter
fox	horse	snails	dog	zebra

Clearly, the zebra is owned by the Japanese.

8.0 PERFORMANCE FIGURES

These figures provide some idea of the performance we have been able to obtain with the above algorithms using our current implementation of F*. If we try to solve these problems in Prolog, using unintellligent generate-and-test, the time taken is far too much to be practical.

Time in seconds on SUN-3/50 with 4 MB main memory	
Changing dollar: all 292 solutions	230
Zebra: Only solution	35
SEND+MORE=MONEY: All 25 solutions	20
8-Queens: All solutions	20
9-Queens: All solutions	97
15-Queens: First solution	37

9.0 SUMMARY

A great deal of work in rewriting has focussed on *confluent* rewrite rule systems. Of course, these are very important because of their close relationship to equality theories. In particular, they can be used to solve the word problem in a computationally feasible manner, e.g. [Knuth & Bendix 1970].

We have tried to show that non-confluent rewrite rule systems such as F* can also exhibit interesting behavior. In particular, we can use these to generate sets using non-deterministic algorithms, which can be very compact. Moreover, if rewrite rules are interpreted lazily, we can achieve substantial reductions of search spaces of problems of the form "given set S, and condition C, find those members of S which satisfy C".

Whenever critical, algorithms achieving such reductions could always be developed, and in any language, declarative, or procedural. What we have proposed is a technique for simplifying their development. In particular, we can separate development of algorithms for generating search spaces, from development of algorithms for testing candidates, yet achieve the search space reduction transparently. Due to the efficient implementation of F*, and as the performance figures show, our technique could be viable for important cases of real problems.

REFERENCES

Barbuti, R., Bellia, M., Levi, G. [1986]. LEAF: A language which integrates logic, equations, and functions. In *Logic programming: functions, relations and equations* (eds.) D. DeGroot, G. Lindstrom, Prentice Hall, N.J.

Bruynooghe, M., Pereira, L.M. [1984]. Deduction revision by intelligent backtracking. In, *Implementations of Prolog*, ed. J.A. Campbell, Ellis Horwood.

Chang, J.-H., Despain, A.M. [1984]. Semi-intelligent backtracking of Prolog based on a static data dependency analysis. *Proceedings of IEEE symposium on logic programming*, Boston, MA.

Clark, K.L., McCabe F. [1979]. Programmer's guide to IC-Prolog. *CCD Report 79/7*, London: Imperial College, University of London.

Dincbas, M., Simonis, H., van Hentenryck, P. [1988]. Solving a cutting-stock problem in constraint logic programming. *Proceedings of fifth international conference and symposium on logic programming*, eds. R. Kowalski, K. Bowen, MIT Press, Cambridge, MA.

Garey, M.R., Johnson, D.S. [1979]. *Computers and intractability. A guide to the theory of NP-completeness*. W.H. Freeman & Co. New York, N.Y.

Huet, G., Levy, J.-J. [1979]. Call by need computations in non-ambiguous linear term rewriting systems. IRIA technical report 359.

Jayaraman, B., Gupta, G. [1987]. EqL User's Guide. Department of Computer Science, University of North Carolina at Chapel Hill, N.C.

Kahn, G., MacQueen, D. [1977]. Coroutines and Networks of Parallel Processes. *Information Processing-77*, North-Holland, Amsterdam.

Knuth, D.E., Bendix, P.B. [1970]. Simple word problems in universal algebras. *Computational problems in abstract algebra*, ed. J. Leech, Pergamon Press.

Kowalski, R. [1979]. *Logic for Problem Solving*, Elsevier North Holland, New York.

Kumar, V., Lin, Y.-J. [1988]. A data-dependency-based intelligent backtracking scheme for Prolog. *Journal of Logic Programming*, vol. 5, No. 2, June.

Lloyd, J. [1984]. *Foundations of logic programming*. Springer Verlag, New York.

Narain, S. [1986]. A Technique for Doing Lazy Evaluation in Logic. *Journal of Logic Programming*, vol. 3, no. 3, October.

Narain, S. [1988]. LOG(F): An optimal combination of logic programming, rewriting and lazy evaluation. Ph.D. Thesis, Department of Computer Science, University of California, Los Angeles.

O'Donnell, M.J. [1985]. Equational logic as a programming language. MIT Press, Cambridge, MA.

Peyton Jones, S. L. [1987]. *The implementation of functional programming languages*. Prentice Hall, Englewood Cliffs, N.J.

Polya, G. [1973]. *How to solve it*. Princeton university press. Princeton, N.J.

Sterling, L., Shapiro, E. [1986]. *The art of Prolog*. MIT Press, Cambridge, MA.

Subrahmanyam, P.A. and You J.-H. [1984]. Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming. *Proceedings of IEEE Logic Programming Symposium*, Atlantic City, N.J.

van Emden, M.H., Yukawa, K. [1987]. Logic programming with equations. *Journal of Logic Programming*, vol. 4, no. 4.

van Hentenryck, P., Dincbas, M. [1987]. Forward checking in logic programming. *Proceedings of fourth international conference on logic programming*, ed. J.-L. Lassez, MIT Press, Cambridge, MA.

APPENDIX. COMPILATION OF F*

For completeness, we outline the algorithm for compiling an F* program into Horn clauses. Let P be an F* program. The compilation of P into pure Prolog clauses proceeds in two steps:

Step 1. For each n-ary, n>=0, constructor symbol c in P, and where X1,..,Xn are distinct variables, generate the clause:

```
reduce(c(X1,..,Xn),c(X1,..,Xn))
```

Step 2. Let f(L1,..,Lm)=>RHS be a rule in P. Generate the clause:

```
reduce(f(A1,..,Am),Out):-Q1,Q2,..,Qm,reduce(RHS,Out).
```

where A1,..,Am,Out are distinct variables not occurring in the rule, and if Li is a variable, Qi is Ai=Li, otherwise, Qi is reduce(Ai,Li).

For example the F* rules:

```
append([],X)=>X
append([U|V],W)=>[U|append(V,W)]
intfrom(N)=>[N|intfrom(s(N))].
```

are compiled into:

```
reduce([],[]).
reduce([U|V],[U|V]).

reduce(append(A1,A2),Out):-reduce(A1,[]),A2=X,reduce(X,Out).
reduce(append(A1,A2),Out):-reduce(A1,[U|V]),A2=W,reduce([U|append(V,W)],Out).
reduce(intfrom(A),Out):-A=N,reduce([N|intfrom(s(N))],Out).
```

If we now type, in Prolog, reduce(append(intfrom(0),[]),Z), we obtain Z=[0|append(intfrom(s(0)),[])]. Thus, append(intfrom(0),[]) is only partially, or lazily reduced, and directly by Prolog, *not by some lazy interpreter implemented in Prolog.*

It can be seen that where reduce(f(A1,..,Am),Out):-Q1,..,Qm,reduce(RHS,Out) is the translation of f(L1,..,Lm)=>RHS, Q1,..,Qm represent the attempt to match some term f(t1,..,tm) with f(L1,..,Lm). If these succeed, the match succeeds with some substitution α. Now, reduce(RHS,Out) represents simultaneously, application of α to RHS, and recursive simplification of RHSα. In fact we have:

Theorem. Correctness of compilation of F*. Let P be an F* program and PC be its compilation. Let E0 and En be ground terms. Then there is a successful N-reduction beginning with E0 and ending with En iff PCl-reduce(E0,En).

To compute normal forms, we need to add, for each m-ary constructor symbol c, the clause:

```
nf(E,c(X1,..,Xm)):-reduce(E,c(T1,..,Tm)),nf(T1,X1),..,nf(Tm,Xm).
```

Now, to compute the normal form of a ground term E, we can execute nf(E,X), where X is a variable.