AN INTERPRETER GENERATOR USING TREE PATTERN MATCHING

Christoph H. Hoffman*  
Michael J. O'Donnell**  
Purdue University  
W. Lafayette, IN 47907

Abstract:

Equations provide a rich, intuitively understandable notation for describing nonprocedural computing languages such as LISP and Lucid. In this paper, we present techniques for automatically generating interpreters from equations, analogous to well-known techniques for generating parsers from context-free grammars. The interpreters so generated are exactly faithful to the simple traditional mathematical meaning of the equation—no lattice-theoretic or fixpoint ideas are needed to explain the correspondence. The main technical problem involved is the extension of efficient practical string matching algorithms to trees. We present new efficient table-driven matching techniques for a large class of trees, and point out unsolved problems in extending this class. We believe that the techniques of this paper form the beginnings of a useful discipline of interpreting, comparable to the existing discipline of parsing.

1. Introduction

Languages for computation may be classified into (a) procedural languages, e.g. ALGOL, PASCAL, which directly describe sequences of actions to be performed, and (b) descriptive languages, e.g. LISP, Lucid, which allow definitions of mathematical objects, functions and relations without direct reference to computational techniques. Since procedural languages have the advantage of allowing a programmer to ensure efficiency by specifying a computation in detail, they have received much at-

*This research was supported by NSF Grant NS 78-01812 and by a Purdue University Summer XL grant.
which may be used to deduce the equivalence of certain expressions in I. We restrict attention to axioms which may be written as equations. Goguen [Go77] claims that "any reasonable computational process can be specified purely equationality." Whether or not this thesis holds, equations provide a plausible starting point, which might be extended by further work, and which is already capable of expressing general purpose languages such as LISP and Lucid.

An interpreter satisfies a specification of the form I, O, A above whenever, given any expression E₀ in I for which there exists an equivalent expression E₂ in O, the interpreter produces such an E₂. For instance, LISP might be specified by letting I include all N-expressions, i.e., expressions formed from atomic symbols, cons, car, cdr, atom, eq, cond, and eval. O would contain exactly the S-expressions: those using only atomic symbols and cons. A would contain the defining equations from [Mc60]. An interpreter based on those specifications would take an N-expression, especially one of the form eval(exp, env), and return an equivalent S-expression if such exists.

Previous work on interpreters for purely descriptive languages includes two types of work: theoretical studies of equational definitions [BL77, Ca72, BS76, Ro75, Sta77, Vu74, O'77], all of which lack important implementation details; and specific studies of individual languages [JN76, JN76, Pay76, JN77] including actual implementations faithful to the precise semantics [Cn76, Ps77, Jo77]. This paper attempts to maintain the generality of the theoretical studies while providing some of the details needed to build practical implementations.

Using the subtree replacement systems of [O'77 and Ro75] as theoretical basis, we outline the steps necessary to apply that theory to specific languages, and develop algorithms for interpreters. Part 2 explains briefly how the theory of subtree replacement systems applies, and what restrictions are required by the present theory.

Part 3 sketches a specification language for interpreters which includes interacting tree replacements with simple (e.g., arithmetic) operations. Parts 4 and 5 treat the structure of the data and algorithms to be used in an interpreter.

We have applied the techniques of this paper to implement an interpreter generator, and have generated interpreters for several languages including LISP and Lucid. Because of the faithfulness to the mathematical semantics, the generator can be used to provide immediate implementations of defined data types from their specifications, as suggested by [Go76] and [Na76].

The central issue in mechanizing interpreter generation is how to extend efficient pattern matching algorithms from strings to trees. We outline briefly the technique employed in our present implementation. We believe that our approach provides techniques which may form the basis for a discipline of interpreter construction comparable to the present discipline of compiler construction.

2. Reduction Sequences Applied to Interpreting

Given I, O, A as in the introduction, a theoretical interpreter might work as follows: take an expression E₀ and enumerate expressions E₂ such that E₀ = E₂ until an E₂ in O is found. Such a scheme is obviously inefficient, unless the enumeration is done in a particularly clever manner. In many cases, equations may be ordered so that (*) the righthand side of each equation is in some sense simpler or clearer than the lefthand side, and so that expressions in O do not contain lefthand sides as subexpressions.

In such cases, a better interpreter might produce a sequence E₀, E₁, E₂, ... of progressively simpler expressions by replacing lefthand sides of equations which appear as subexpression in some E₂ by the corresponding righthand sides, until (hopefully) an E₂ in O is found. Such reduction sequences are studied in [BL77, Ca72, DS76, Ro75, Sta77, Vu74, O'77].

In a reduction sequence, each occurrence of a lefthand side of an equation is called a redex. An expression which contains no redexes is in normal form, and must be the last expression in the sequence. Under the (*) assumptions, every expression in O is in normal form. Since a single expression may contain several different redexes, there may be many different reduction sequences starting with the same E₀. In order to use reduction sequences for interpreters, we must know how to choose an appropriate reduction sequence:
one that terminates with an $E^g$ in normal form whenever $A$ entails $E^g = E^f$, and one that is not too long. To guarantee such behavior, we need a few reasonable restrictions on equations.

(1) No variable may be repeated on the left-hand side of an equation. For instance, if $X$ then $Y$ else $Y = Y$ is prohibited.

(2) If two different left-hand sides match the same expression, then the corresponding right-hand sides must be the same. So the pair of equations $g(0,x) = 0$ and $g(x,1) = 1$ is prohibited, since $g(0,1)$ could be replaced by 0 or 1.

(3) When two (not necessarily different) left-hand sides match two different parts of the same expression, the two parts must not overlap. E.g., the pair of equations $\text{first}(\text{pred}(X)) = \text{pred}$ and $\text{pred}(\text{succ}(X)) = X$ are prohibited, since the left-hand sides overlap in $\text{first}(\text{pred}(\text{succ}(0)))$.

[O'D77] shows that, with these three restrictions,

(1) For each expression $E^g$, there is at most one normal form $E^f$ which may be obtained by reducing $E^g$.

(2) Any strategy for choosing reduction sequences which guarantees that every possible outermost replacement in an expression is eventually done will produce $E^f$ in normal form such that $A$ implies that $E^g = E^f$, whenever $E^g$ exists.

We can use (2) above to prove that an interpreter satisfies its specifications. Strategies for choosing reduction sequences fall into two classes:

(a) Parallel strategies, in which several reducibles are reduced simultaneously (in practice, "simultaneous" reductions are scheduled sequentially according to some fair queueing discipline).

(b) Sequential strategies, in which a single reducible is chosen at each step.

The most common sequential strategy chooses reducibles in preorder, i.e., leftmost outermost first. Every set of equations satisfying the restrictions above may be handled by a parallel strategy. See [O'D77] and [Y74] for a general discussion of the additional restrictions needed to allow sequentiality. For LISP, the leftmost outermost strategy is correct and optimal, but for LUCID, because of the equations $\text{or}(T, X) = T$ and $\text{or}(X, T) = T$, a parallel strategy is required.

Using subtree replacement systems as a model, we may organize the task of implementing an interpreter into the following steps:

(1) Specify the language to be interpreted in terms of $I$, $O$, $A$, with $A$ in the form of equations.

(2) Convert the equations $A$ into a form satisfying $(*)$ and the additional three restrictions above.

(3) Pick a data structure to represent expressions and an algorithm for performing single reductions.

(4) Pick a strategy for choosing the next reducible to be replaced, and develop an efficient algorithm to find the reducible specified by that strategy.

Step (1) is inherently intuitive, but the other steps may be partially or fully automated. To automate Step (2) requires further research.

[KB70] gives automatic techniques which sometimes succeed in eliminating overlap in equations; but at present, a language designer must usually perform Step (2) intuitively. Section 3 of this paper gives a specification language in which (1) and (2) may be presented, and an example of such a presentation. Sections 4, 5 and 6 discuss two different methods for performing (3) and (4) automatically.

3. The Interpreter Specification Language

To specify the allowable input expressions one need only list a set of symbols with their priorities. The input expressions will be the usual terms composed of the listed symbols. Most interesting programming languages include large sets of standard symbols such as integer constants. To avoid excessively long specifications, there are standard primitive sets of symbols which may be specified by a single name. In this paper we will use the sets $\text{integer}$, containing all integer constants as term symbols, the zeroary boolean symbols $T$ and $F$, as well as the binary symbols $\ast$, $\cdot$, $\div$, $\mod$, $\text{eq}$, $\text{ne}$, $\text{lt}$, $\text{gt}$, $\text{le}$, $\text{ge}$. We also use the set $\text{unspecified}$ containing as zeroary symbols all alphanumerics strings not otherwise accounted for, on which $\text{eq}$ and $\text{ne}$ are defined.

These sets of primitive symbols containing con-
Constants are primitive domains, and symbols of higher
degree will be called standard functions. Now, a
useful subset of $N$-expressions may be specified as
follows:

**SYMBOLS**
- cons: 2; car: 1; atom: 1; eval: 2;
even: 2; integer: boolean;
cond: 3; cdr: 1; pair: 2; even: 2;
assoc: 2; unspecified;
quote: 0; atom: 0; eq: 0; cond: 0;
car: 0; cdr: 0; cons: 0; label: 0;
lamba: 0.

Many important equations may be given direc-
tly, e.g. $\text{car} (\text{cons}(X,Y)) = X$. Theoretically, such
equations are sufficient to define general purpose
programming languages. For practical purposes,
however, standard functions such as + defined on
primitive domains should be computed by program.
That is, the set of equations $\{0,0\} = 0; \{0,1\} = 1;
\{1,0\} = 1; \{1,1\} = 2; \ldots$ is implicitly speci-
fied by a program which, given that a subexpres-
sion $\text{+}(X,Y)$ where $X$ and $Y$ are integer constants is
to be reduced, replaces it with an integer constant
$Z$ such that $\text{+}(X,Y) = Z$. Whenever primitive sets of
symbols are introduced in the language specifi-
cation, these programs representing equations defini-
ing standard operations are also included. One
may wish to visualize the effect of these pro-
grams as specified by schemata, such as

$$\text{+}(X,Y) = Z \text{ where } X, Y, Z \text{ in integer, and } Z \text{ is }
\text{the sum of } X \text{ and } Y,$$

with which is associated some subroutine computing,
in this case, the value of $Z$ as the sum of $X$ and $Y$.
In this view we can implement predefined standard
functions by a small extension of the mechanisms
needed for user-defined reductions. It is tem-
ting to extend this approach and to allow the user
to associate subroutines with reductions in gen-
eral, analogous to the way in which so called
semantic processing is associated with syntactic
reduction steps performed by a parser. However, we
wish to limit our device to a few standard functions
defined on primitive domains, to ensure that the
language semantics is correctly specified, avoiding
tedious separate proofs.

A syntactic difficulty arises when a large
number of equations must mention each expression in
a large class. For instance, one specification of
LISP would include a separate equation atom(const) = \top
for each unspecified symbol and each integer con-
stant. We condense such sets of equations into one
by using a variable restricted to range over a
union of primitive domains. Thus, the LISP axioms
may be given as follows:

**AXIOMS**
- FOR ALL $X, Y$:
  - $\text{car}(\text{cons}(X,Y)) = X$
  - $\text{cond}(\text{cons}(X,Y)) = Y$
  - $\text{atom}(X) = T \text{ where } X \text{ in integer or }
    \text{unspecified+boolean;}
  - $\text{atom}(\text{cons}(X,Y)) = F; \ldots$

The complete set is given in the Appendix A. The phrase
FOR ALL $X, Y$ indicates that the symbols $X$ and
$Y$ are to be taken as variables, rather than as
unspecified constants, in the axioms.

The set of output expressions is given im-
plicitly as the set of normal form expressions as
determined by the axioms. A separate specification
of 0 could lead to a uniform treatment of certain
errors [see (O'd77) p. 83], but this possibility
has not yet been explored.

4. Reduction With Backpointers

The problem of reducing an expression $E_0$ to
normal form divides naturally into (a) finding
reducts, (b) choosing the next reduct to be replaced,
(c) performing a single reduction step.

Finding reducts is essentially a pattern match
problem with trees instead of strings. The choice
of a suitable algorithm is complicated by the fact
that each reduction step alters the expression tree
locally. It is unacceptable to recompute the entire
structure after each reduction step.

Expressions may be stored as daggs in which
each node is connected by a circular pointer structure
to each of its fathers, as sketched:

![Diagram of a dag representation]

In this way, from any node we can find any son or
father. Without sharing, we double the pointer
space required. Sharing (i.e., multiple fathers)
cuts down on this wasted space, and may reduce the
number of reduction steps needed to reduce $E_0$.
Since the technique for a single reduction step is
straightforward, we omit discussing it. The key idea for identifying reductions is to associate with each node a code indicating which part(s) of left-hand sides of equations match the subtree at that node. We compute these match states from the leaves up, using a precomputed table giving the state at node p as a function of the label at p and the states of p's sons. In this way, all reductions may be located in $E_0$ in time proportional to the size of the dag. The problem of matching tree patterns and of generating these tables is studied in detail in [Ho78]. For the purposes of this paper, we limit the discussion of the technicalities of tree matching to Section 5.

When a reduction is performed, match states must be computed for any new nodes which have been added, and for some of the ancestors of the redex. The back pointers make it easy to find all affected ancestors. The length of any path along which an update can occur is limited by the maximum depth of equation left-hand sides. During this local update, new redices may be discovered.

Given that every node in the dag representing an expression has been assigned the appropriate match state, a simple parallel strategy for choosing the next redex would be to keep all redices in a queue, reducing the first redex in it, and adding newly discovered redices at the rear. A standard reference count detects if any redex is removed from the expression as side effect of a redex preceding it in the queue. The strategy is correct but not optimal.

In cases where an a priori sequential ordering of redices is given, an appropriate depth-first traversal of the dag implements the reduction strategy. A single additional bit maintained at each node indicating whether the subexpression rooted at that node is in normal form will prevent useless rescanning of the same (shared) normal form subexpression repeatedly.

We have implemented a generator system based on these ideas in PASCAL. The implementation effort has been approximately 5 man weeks. We have generated no frills interpreters for LISP, LISP, and the combinator calculus. Actual reductions have been very fast.

5. Matching Tree Patterns

We wish to generate tables from a set of tree patterns (the axioms $H$) with which to drive the linear matching algorithm outlined in Section 4. All possible sets of partial matches need to be known, since they are used to index into the tables during the matching process. During the process of this generation, the restrictions on axiom lefthand sides of Section 2 will also be checked. Note that there is a straightforward matching algorithm which works on $O(n \cdot m)$ steps, where $m$ is the pattern size to be matched in a subject of size $n$. This algorithm requires no preprocessing, and works for all patterns. The algorithm of Section 4, in contrast, matches in $O(n)$ steps, after suitable tables have been generated as explained now.

Given a forest $F$ of tree patterns $t_1, \ldots, t_k$, a match set $M$ for $F$ is a set of (sub)trees in $F$ such that there is a subject tree $t$ such that every member of $M$ matches $t$ at the root, and every (sub)tree in $F$ which is not in $M$ does not match $t$ at the root. $M$ is thus the set of all (partial) matches at the root of $t$. A table of match sets may be generated straightforwardly in time $O(a^{k-1} \cdot m)$ where $a$ is the number of symbols, $k$ is the maximum arity of any symbol (so the table is size $O(s^k)$), and $m$ is the total size of all patterns. We discuss only more efficient methods.

Given distinct patterns $t$ and $t'$, we define two relations: $t$ subsumes $t'$, $t \triangleright t'$, if every match of $t$ always implies a match of $t'$ at the same node. For example, $a(b,c) \triangleright a(b,v)$, since $v$ matches whenever $c$ does. $t$ and $t'$ are independent, $t \perp t'$, if we can find subject trees $t_1$, $t_2$, and $t_3$ such that $t$ matches $t_1$ and $t_2$ at the root, but not $t_3$, whereas $t'$ matches $t_1$ and $t_3$ at the root, but not $t_2$. Thus, $a(b,v) \perp a(v,c)$, because of trees $a(b,c)$, $a(b,v)$, and $a(c,v)$.

It is not hard to show that each match set $N$ may be partitioned uniquely into a base set $H_0$ of pairwise independent trees, and a set $H_1$ of trees each of which is subsumed by some tree in $H_0$. Because of the transitivity of subsumption, each match set is completely determined by its base set.

Using the defined relations, [Ho78] shows that

1. The number of distinct match sets may grow exponentially with the pattern size.
2. If there are no (sub)trees in the pat-
tern forest $F$ which are independent, then the number of possible match sets is equal to the size of $F$.

Because of these results, we restrict axioms such that their l.h.s. form pattern sets in which no two (sub)trees are independent. Such pattern sets are called simple.

Define immediate subsumption, $\alpha \triangleright \beta$, by $\frac{\alpha}{\beta} \triangleright \frac{\beta}{\beta}$, iff $\alpha = \gamma$ and there is no (sub)tree $\gamma$ in the pattern forest $F$ such that $\alpha < \gamma$ and $\gamma < \gamma$.

The directed acyclic graph $G_s$ of the immediate subsumption relation is called the subsumption graph of $F$. For simple forests, [Ho78] shows that

1. $G_s$ is a tree.
2. The base set of each match set $M$ is a singleton.
3. The match set $N$ with base set $(t)$ is precisely the set of trees on the path from the base tree $t$ to the root of $G_s$.

Let $n$ be the size of the pattern forest $F$, and $d$ the depth of $G_s$. There is a straightforward algorithm for computing the transitive closure of $G_s$ in $O(n^3)$ steps. Using an indexing scheme, we can design an $O(n^2)$ algorithm, which is slower in the worst case, but can be expected to run significantly faster than the $O(n^3)$ algorithm, which is quadratic for all inputs. Both algorithms, at the same time, can check that $F$ is simple and that the restrictions of Section 2 are satisfied, without affecting the running time. The $O(n^2)$ algorithm is given in Appendix B.

If there are no function symbols with arity exceeding 2, then there is an $O(n)$ algorithm for computing $G_s$. The algorithm can be adapted to perform the actual matching too, leading to a matching algorithm of $O(n^2)$ steps in a subject of size $n$.

The algorithm can be adapted to compute, in the same time bound, $G_s$ for alphabets of higher degrees, but will then be unable to process certain simple forests. The details are covered in [Ho78].

Once $G_s$ has been computed, the tables to drive the $O(n)$ matching algorithm can be constructed easily. If $k$ is the highest occurring arity in the alphabet $A$, then the tables require $O(n^k)$ space and take $O(n^k)$ steps to construct. Unfortunately then, table generation is the bottleneck of the preprocessing. Since the maximum arity $k$ of alphabets affects the size and time of table generation so critically, it is useful to reduce $k$ by introducing a set of pairing functions. We have used this technique successfully to speed up the interpreter generation, but it should be noted that pairing sometimes transforms simple pattern forests into forests in which independence occurs. This phenomenon is also responsible for the failure of the adapted $O(n^d)$ algorithm to process all simple forests for alphabets of higher arities, since the intermediate graphs constructed by the algorithm conceptually imitate argument pairing.

Although table generation is the bottleneck of the preprocessing, it is well worth while to investigate ways to speed up the computation of $G_s$ further, because these algorithms can be adapted to perform the actual pattern matching without the need for generating large tables. This may best be understood by observing the analogy of tree pattern matching and string pattern matching in the style of [Kn77] and [AC75].

Consider a string pattern $a_1, \ldots, a_m$ as nonbranching tree $a_k(a_{k-1}(\ldots(a_2(a_1(t))))).$ The graph $G_s$ for a forest of such nonbranching trees is precisely the graph of the failure function of [Kn77] and [AC75]. For this, note that a subtree of a nonbranching tree is a pattern prefix. Now $t > t'$, for trees $t$ and $t'$, if $t'$ matches $t$ at the root, therefore, in the case of nonbranching trees, $t'$ is a pattern prefix which is, at the same time, a suffix of $t'$. Thus $t \triangleright t'$ iff $t'$ is the largest proper pattern prefix which is also suffix of $t$.

Thus, it is reasonable to look for matching algorithms which use principally $G_s$ as data structure. The adaptation of the $O(n^d)$ algorithm is designed in just that way.

6. Reduction Without Back Pointers

Up to half the pointers in the implementation of Section 4 may be eliminated by representing dags without back pointers. At present we do not have a nice algorithm for parallel strategies without back pointers. For sequential strategies, a simple implementation uses the match states and normal form of Section 4. A depth-first traversal is used to find the next node, skipping any subtrees which are marked as being in normal form. Whenever a node is found whose state or normal form bit may be changed by recomputing from its sons, the change is
propagated upwards, but only along the path by which
the node was reached (this path is known from the
standard stack used for the traversal).

**Informal Development of a More Powerful
Algorithm**

The simple algorithm outlined above does not
address the problem of finding an acceptable sequen-
tial strategy. We have a method which finds optimal
sequential strategies automatically. In addition,
this method generalizes a trick applied by Friedman
and Wise to LISP (PWM76) in which portions of an
expression which have become stable are output and
eliminated to save space. A fuller development of
the algorithm is being prepared for journal publi-
cation, with a proof that the method finds a sequen-
tial strategy whenever such success is possible
without considering the right hand sides of equa-
tions.

The match state idea from section 4 is suffi-
cient for recognizing a redex once we have scanned
the appropriate part of a tree. The additional
problem is to decide which parts of the tree to
scan. This decision must account for the possibility
that some match states are out of date, since a
shared subtree may be changed through one path
without the change being noticed on other paths.

The main new concept needed is that of a
possibility state. The match state M(n) at a node
n represents all partial matches known to hold at
n. The nonoverlapping property guarantees that
existing matches at n will never be destroyed by
reductions at descendants of n, but new matches
could be created. The possibility state P(n) for
n represents a set containing all partial matches
which might ever hold at n as the result of re-
ductions at descendants. At any given time, some
matches in P(n) \setminus M(n) may already hold at n due to
reductions not yet noticed by n. The true set of
matches holding at n must always be a superset of
M(n) and a subset of P(n). To avoid obvious unde-
cidable questions, P(n) is computed without know-
ledge of equation right hand sides by assuming that
a redex may be replaced by any tree.

Finally, we need one more kind of state, called
a search state, to keep track of those partial
matches which might be useful to the reduction.
Match states and possibility states are stored at
each node. Search states \( S \) are stored on the

traversal stack, and contain all partial matches
which might make a reduction possible at some node
on the stacked path from the root.

The algorithm is developed from the following
observations:

1) If M(n) contains a complete match,
then a reduction may be performed;

2) If M(n) \cap S \neq \emptyset then an interesting
change has occurred which should be
propagated up the tree;

3) If M(n) \cap S = P(n) \cap S = \emptyset then no node
presently on the stack will ever be
changed;

4) If M(n) \cap S = \emptyset but P(n) \cap S \neq \emptyset then
further processing of descendants is
needed, and the appropriate node to
visit may be recognized by his match
state and possibility state.

A precise statement of the algorithm is attached
as Appendix C.

The problem of precomputing states for the new
algorithm is even trickier than for the old. So
far we know that in some cases possibility sets are
exponential in number even though match sets are
very few. More study is needed to discover those
cases in which the total number of combinations is
not too big.
Appendix A -- LISP Equations

McCarthy's original LISP equations [McG60] have several apparent mistakes. The following equations represent a correction and reordering of McCarthy's definition.

AXIOMS FOR ALL V, W, X, Y, Z:

car (cons (X, Y)) = X;
cdr (cons (X, Y)) = Y;
atom (cons (X, Y)) = F;
atom (X) = T

where X is integer, boolean, unspecified

U (QUOTE, ATOM, EQ, COND, CAR, CDR, CONS, LABEL, LAMBDA);

cond (T, X, Y) = X;
cond (F, X, Y) = Y;

eval (X, Z) = assoc(X, Z)
    where X in unspecified;

eval (cons (X, Y), Z) = apply (eval (X, Z), evalis (Y, Z))
    where X in unspecified U (ATOM, EQ, COND, CAR, CDR, CONS);

eval (cons (cons (W, X), Y), Z) = apply (eval (cons (W, X), Z), evalis (Y, Z));

apply (eval (ATOM), Z, cons (X, Y)) = atom (X);
apply (eval (EQ), Z, cons (W, cons (X, Y))) = eq (W, X);
apply (eval (COND), Z, X) = condis (X);
apply (eval (CAR), Z, cons (X, Y)) = car (X);
apply (eval (CDR), Z, cons (X, Y)) = cdr (X);
apply (eval (CONS), Z, cons (W, cons (X, Y))) = cons (W, X);
apply (eval (cons (LABEL), cons (Y, cons (W, X))), Z, Y) =
    apply (eval (W, cons (cons (V, cons (cons (LABEL, cons (Y, cons (W, X))), NIL), Z), Y);
apply (eval (cons (LAMBDA, cons (Y, cons (W, X))), Z), Y) =
    eval (W, append (pair (V, Y), Z));

evalis (NIL, Z) = NIL;
evalis (cons (X, Y), Z) = cons (eval (X, Z), evalis (Y, Z));

append (NIL, Y) = Y;
append (cons (W, X), Y) = cons (W, append (X, Y));
pair (NIL, NIL) = NIL;
pair (cons (V, W), cons (X, Y)) = cons (cons (V, cons (X, NIL)), pair (W, Y));

condis (cons (cons (T, cons (X, NIL)), Y)) = X;
condis (cons (cons (F, cons (X, NIL)), Y)) = condis (Y);

assoc (X, Y) = cond (eq (car (car (Y)), X), car (cdr (car (Y))), assoc (X, cdr (Y))).
Appendix B -- Pattern Preprocessing Algorithm

The preprocessing of simple pattern forests for generating tables divides into the computation of the subsumption graph $G_S$ (which is a tree for simple forests), and the generation of tables from $G_S$. The computation of $G_S$ with suitable changes not indicated here, also verifies that the patterns presented form a simple forest. Verifying the nonoverlap property can also be incorporated.

Assume patterns $t_1, \ldots, t_k$ are given. Let $T$ denote the set of all (sub)trees of the $t_i$, and denote a directed edge from $t$ to $t'$ in $G_S$ by $f(t) = t'$ -- i.e., $t$ directly subsumes $t'$. The computation of $G_S$ is now as follows:

Algorithm A Compute Subsumption Graph $G_S$ for Linear Forest $F$

Input: Linear forest $F$ of patterns
Output: Tree $G_S$ (with edges pointing to ancestors)
Method:
1. Order all trees in $T$ by their depth.
2. For each $t = v \in T$ of depth $i$ do $f(t) = v$; 
   Comment: $f(t) = t'$ iff there is a directed edge from $t$ to $t'$;
3. For $p = 2$ to MAXDEPTH IN FOREST do
   For each $e = (t_{i_1}, \ldots, t_{i_k})$ in $T$ of depth $p$
   do begin
   4. $s := v$;
   5. For $i = 1$ to $k$ do begin
   6. $t' := f(t_i)$;
   7. while there is no tree $t''$ of (maximal)
   depth $\leq p$ which is subsumed
   by $t$ and has $t'$ as $i$-th sub-
   tree and $t'' \neq v$ do
   8. $t'' := f(t')$;
   9. For each tree $t''$ with $t'$ as $i$-th sub-
   tree which is subsumed
   by $t$ and of maximal depth $\leq p$
   do if $t'' > s$ then $s := t''$;
10. end;
11. end;
12. enter $f(t) = s$;
13. end;

Note that, since we process trees ordered by increasing depth, the cost $t$ subsumes $t''$ can be

done by verifying, for each immediate sub-tree pair
$t_1$ and $t_1'$, that $t_1$ subsumes $t_1'$. Since the depth
of both $t_1$ and $t_1'$ must be strictly smaller than $p$,
this test involves tracing through the existing
portion of $G_S$.

Since there may be, in some cases, up to $O(n)$
trees $t''$ with $i$-th subtree $t_i$, where $a$ is the card-
inality of $T$, and since tracing through the existing
portion of $G_S$ for testing subsumption may involve
up to $O(d)$ steps, where $d$ is the depth of $G_S$, the
algorithm requires $O(n^2 \cdot d)$ steps.

Given $G_S$, we can then construct tables in the
following manner.

Algorithm B

Input: Subsumption graph $G_S$ of linear forest $F$
Output: Tables driving the fast matching algorithm
of Section 2.
Method:
1. Traverse $G_S$ in post order. For each tree
   $t = e(t_1, \ldots, t_k)$ visited, $k > 0$, do the
   following:
2. Enter $t$ into all portions of the table for $a$
   which are not yet assigned and are indexed
   by tuples $< t_1', t_2', \ldots, t_k' >$ where, for
   $1 \leq i \leq k$, $t_i' > t_i$ if $t_i > t_i'$.
3. Enter $v$ into all remaining unassigned table
   positions in each table.

For linear forests, it can be proved that
Algorithm B cannot attempt assigning $t$ to an entry
already assigned $t'$, unless $t' < t$; hence the $O(n^2 \cdot d)$
time bound, where $k$ is the highest occurring arity
in $F$.

Appendix C -- Reduction Without Backpointers

Let $n$ be any node in a dag representing a tree.
Labels: $\lambda(n)$ is the alphabet symbol at $n$;
Match States: $M(n)$ is the set of all partial
matches known to hold for the subtree
rooted at $n$;
Possibility States: $P(n)$ is a set of partial
matches which might arise at
$n$ as the result of reductions
at proper descendants of $n$;
$P(n)$ is a set of partial matches

9
Louise C. Coval
which might arise as a result of reductions at n and its descendants;
C is the set of complete matches of patterns;
U is the set of all partial matches.

Note that
\[ P(n) = P'(n) \text{ if } P'(n) \cap C = \emptyset \]
\[ P(n) = U \text{ if } P'(n) \cap C \neq \emptyset \]

A stack of pairs \( < n, S > \) is used to control the traversal. The nodes on the stack form a branch from the root. The search state \( S \) is the set of all partial matches whose occurrence at \( n \) could produce a complete match at some node below \( < n, S > \) on the stack.

When \( T \) is a set of partial matches,
- \( \text{Son}_i(T) \) is the set of all \( i \)th subtrees of roots of members of \( T \);
- \( \text{Father}_i(T) \) is the set of all partial matches whose \( i \)th subtrees are in \( T \);
- \( U_i \) is the set of all partial matches with root labeled \( i \).

The Algorithm

Initially: \( H(n) = P'(n) = U_i(n) \) for each leaf \( n \);
\( M(n) = \bigcup \text{Father}_i(M(\text{Son}_i(n))) \bigcap U_i(n) \);
\( P'(n) = \bigcup \text{Father}_i(P(\text{Son}_i(n))) \bigcap U_i(n) \);
for every nonleaf \( n \);
\[ P(n) = \begin{cases} P'(n) \text{ if } P'(n) \cap C = \emptyset \\ U \text{ if } P'(n) \cap C \neq \emptyset \end{cases} \]
for every node \( n \);

The stack initially contains \( < \text{root}, C > \).

While the tree to be reduced has not been SPLIT Do
- If \( n \) is not a leaf then
  \[ H(n) := \bigcup \text{Father}_i(H(\text{Son}_i(n))) \bigcap U_i(n) \];
  \[ P'(n) := \bigcup \text{Father}_i(P(\text{Son}_i(n))) \bigcap U_i(n) \];
  \[ P(n) := \begin{cases} P'(n) \text{ if } P'(n) \cap C = \emptyset \\ U \text{ if } P'(n) \cap C \neq \emptyset \end{cases} \];
  End if;
- If \( H(n) \cap C \neq \emptyset \) then REDUCE
- Else if \( H(n) \cap S \neq \emptyset \) then POP
- Else if \( P(n) \cap S \neq \emptyset \) then SPLIT
- Else Choose \( i \) such that
  \( \text{Son}_i(P'(n) \cap C) \cap M(\text{Son}_i(n)) = \emptyset \);
  \[ \text{PUSH} (< \text{Son}_i(n), \text{Son}_i(P'(n) \cap C) >) \]
End While

REDUCE, PUSH AND POP have their intuitive meanings.
SPLIT is invoked when the nodes on the stack have all stabilized (i.e., future reductions cannot possibly change them). SPLIT outputs the nodes on the stack, freeing them for garbage collection, and initiates processing of the remaining subtrees in any order, or simultaneously.

References

CACM 18:6, 333-343.


[BL77] Berry, G. and J. Levy, Minimal and Optimal Computation of Recursive Programs.

[Cud72] Cudmore, J., Recursive Definitions of Partial Functions and Their Computations,
Ph.D. Diss., Comp. Science Dept., Stanford University.

[Car78] Cargill, T., Deterministic Operational Semantics for Lucid, Res. Rept. CS-76-19, Univ. of Waterloo.


[Fa77] Farah, H., Correct Compilation of a Useful Subset of Lucid, Ph.D. Diss., Dept. of
Comp. Science, Univ. of Waterloo.

[FW76,1] Friedman, D., and D. Wise, Cons should not evaluate its arguments, 3rd Int. Colloq.
on Automata, Languages and Programming, Edinburgh.

FW76,2] Friedman, D., and D. Wise, Output Driven Interpretation of Recursive Programs, or
(Gn76) Guttag, J., E. Horowitz and D. Musser, 


(Ho76) Henderson, P., and J. K. Morris, A Lazy 
Evaluator, 3rd ACM Symp. on POPL, 95-103.

(Ho78) Hoofkman, C., Matching Tree Patterns, 
Technical Rept. 291, Dept. of Comp. Sci., 
Purdue University, 1978.

(Jo77) Johnson, S.D., An Interpretable Model for a 
Language Based On Suspended Construction, 
Tech. Rept. 68, Dept. of Comp. Sci., 
Indiana University.

(Kn76) Knuth, D., and P. Bendix, Simple Word 
Problems in Universal Algebras. Compu- 
tational Problems in Abstract Algebra, J. 

(Kn77) Knuth, D., J. Morris, and V. Pratt, Fast 
Pattern Matching in Strings, SIAM J on 

(Me76) McCarthy, J., Recursive Functions of 
Symbolic Expressions and Their Compu- 
tation by Machine, CACM 3:4, 184-195.

(O'D77) O'Donnell, M., Computing in Systems 
Described by Equations, Springer Lecture 
Notes in Comp. Science #58.

(Sta77) Staples, J., A Class of Replacement Systems 
with Simple Optimality Theory, Bull. of 

(Wa76) Wand, H., First Order Entities as Defining 
Language. Techn. Rept. 26, Dept. of Comp. 
Science, Indiana University.

(Ro73) Rosen, B. K., Tree Manipulation Systems 
and Church-Rosser Theorems, JACM 20:1, 
160-187.

(Vu74) Vuillemin, J., Correct and Optimal 
Implementations of Recursion in a Simple 
Programming Language. JCSS 9:1, 332-354.