(2)

MORE EFFICIENT BOTTOM-UP TREE PATTERN MATCHING

J. Cai
R. Paige
R. Tarjan

CS-TR-268-90

May 1990

Princeton Univ.
Dept. of Computer Science

More Efficient Bottom-Up Tree Pattern Matching

J. Cai and R. Paige [1] *and* *R. Tarjan* [2]

Dept. of Computer Science
NYU/Courant Institute
New York, NY 10012

Dept. of Computer Science and
Princeton University
Princeton, NJ 08540

NEC Research Institute
4 Independence Way
Princeton, NJ 08544

ABSTRACT

Pattern matching in trees is fundamental to a variety of programming language systems. However, progress has been slow in satisfying a pressing need for general purpose pattern matching algorithms that are efficient in both time and space. We offer asymptotic improvements in both time and space to Chase's bottom-up algorithm for pattern preprocessing. Our preprocessing algorithm has the additional advantage of being incremental with respect to pattern additions and deletions. We show how to modify our algorithm using a new decomposition method to obtain a space/time tradeoff. Finally, we trade a log factor in time for a linear space bottom-up pattern matching algorithm that handles a wide subclass of Hoffmann and O'Donnell's Simple Patterns.

1. Introduction

Pattern Matching in trees is fundamental to term rewriting systems [11], transformational programming systems [4, 7, 18, 22], program editing and development systems [6, 13], code generator generators [9, 17], theorem provers[14], logic programming optimizers that attempt to replace unification with matching[16], and compilers for ML[21], Haskell[12], and a variety of functional languages with equational function definitions. However, this problem seems to be extremely difficult. The best known space-efficient top-down algorithm to locate all occurrences of a pattern tree of size l in a tree of size n takes $O(nl^{.75}polylog(l))$ time, a recent result due to Kosaraju [15], which is barely better than the naive $O(nl)$ algorithm. Bottom-up pattern matching seems to be even more difficult than top-down matching and is of special practical importance. In a seminal paper Hoffmann and O'Donnell presented bottom-up tree pattern matching algorithms that were highly efficient in time but required excessive space [10] both in theory and practice (see Chase's empirical data [5]). Hoffmann and O'Donnell's work has stimulated a number of papers offering heuristic space improvements [2, 3, 5, 19], and Chase's method has aroused considerable attention [5]. However, none of these papers offer theoretical improvements or promising space/time tradeoffs.

1. Part of this work was done while Paige was a summer faculty member at IBM T. J. Watson Research Center. This work is also partly based on research supported by the Office of Naval Research under Contract No. N00014-87-0461.
2. Research at Princeton University partially supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center, grant NSF-STC88-09648, and the Office of Naval Research, contract N00014-87-K-0467.

In this paper we present two new theoretical results in bottom-up tree matching.

1. At the end of his CAAP '88 paper [3] Burghardt called for an algorithm that could preprocess pattern trees incrementally as worthwhile future research. Such an algorithm is needed in the RAPTS transformational programming system [4], because incrementally modifying systems of rewrite rules is a frequent activity, and preprocessing full sets of patterns is highly expensive.

In this paper we present a modification to Chase's algorithm so that its costliest task, preprocessing, can be achieved incrementally with respect to additions and deletions of patterns. When our algorithm is applied repeatedly to solve nonincremental preprocessing by adding one pattern at a time starting from the empty set, it runs asymptotically better in time and space than Chase's algorithm.

2. In bottom-up pattern matching, the main difficulty that sorely needs to be overcome is space utilization. We present an algorithm for a subclass of Hoffmann and O'Donnell's Simple Patterns that runs in $O(l)$ space overall and $O(\log l)$ time per step, where l is a parameter related to the number of input patterns. Previous bounds due to Hoffmann and O'Donnell are $O(l^2)$ time and space for an algorithm tailored to binary Simple Patterns (which our subclass properly includes) and $O(l^{kmax+1})$ space with $O(1)$ step time for an algorithm handling all Simple Patterns, where *kmax* is the maximum arity of a pattern. Thus, we offer a quadratic space improvement over the latter algorithm for binary patterns and even more dramatic improvement for patterns of greater arity. Our space compression is obtained by applying persistent data structures in a new way.

2. Bottom-up pattern matching

Hoffmann and O'Donnell [10] define patterns inductively as follows:

Definition: Given an alphabet of one distinguished variable v and function symbols f with fixed arity $A(f)$, then the set of patterns is the smallest set of terms that include (i) v, (ii) constant c if c is a function symbol with arity 0, and (iii) $f(p_1, \cdots, p_k)$, which we call an *f–pattern*, if f is a function symbol of arity k and $p_1, \cdots, p_k$ are patterns.

They also define pattern matching as follows:

Definition: Pattern p_1 is said to be more general than pattern p_2, denoted by $p_1 \geq p_2$, iff either (i) p_1 is v, or (ii) p_1 is $f(x_1, \cdots, x_k)$, p_2 is $f(y_1, \cdots, y_k)$ and $x_i \geq y_i$ for $i = 1, \cdots, k$.

If $p_1 \geq p_2$, we also say that p_1 *matches* p_2 or that $[p_1, p_2]$ is a *match*. The set of subexpressions, or subpatterns, of p is denoted by $sub(p)$. A slightly more general form of the main pattern matching problem considered by Hoffmann and O'Donnell is:

Problem(Multi-pattern matching): Given a set P of patterns and a pattern t called the *subject*, find the set $MPTM(t) = \{[p, q]: p \in P, q \in sub(t) \mid p \geq q\}$ of all patterns in P matching subpatterns of t.

As the preceding notation suggests, bottom-up solutions presented by Hoffmann and O'Donnell and Chase treat the set P of patterns as fixed and the subject t (which for them has no variables) as the only parameter that can vary. In a bottom-up strategy to solve the multi-pattern matching problem, a complete set $MPTM(q)$ of matches is found for each subpattern q of t without reference to any subpattern of t that properly encloses q.

In order to explain these algorithms, we need to first present a few definitions and notational conventions.

Definition: If P is a set of patterns, then the *pattern forest PF* of P is the set of subpatterns of all the patterns in P.

Definition: If PF is the pattern forest for a set P of patterns and t is the subject, then the *match set* $MS(t)$ for t is defined by the rule $MS(t) = \{q \in PF \mid q \geq t\}$.

The main idea underlying Hoffmann and O'Donnell's bottom-up algorithm is the following equivalent recursive definition:

$$MS(v) = \{v\}$$
$$MS(c) = \{v\}, \text{ when constant } c \notin PF$$
$$\{v,c\}, \text{ when constant } c \in PF$$

(1) $MS(f(t_1, \cdots, t_k)) = \{f(q_1, \cdots, q_k) \in PF \mid q_i \in MS(t_i), i = 1, \cdots, k\} \cup \{v\}$

After determining match sets for constants and variable occurring in the subject t, the main task of Hoffmann and O'Donnell's bottom-up algorithm is to identify the match set for each subpattern $f(t_1, \cdots, t_k)$ of t based on the match sets for t_i, $i = 1, \cdots, k$. This is achieved by solving expression (1), which we call the *Basic Bottom–Up Step*.

Consider a multi-pattern matching problem instance with pattern set P, pattern forest PF, and subject t. We will use the following parameters throughout this paper:

n = length of t
m = number of match sets for P
$l = |PF|$
$o = |MPTM(t)|$
$kmax$ = maximum arity of any function appearing in PF

In order to compute Step (1) and print the set $MS(f(t_1, \cdots, t_k)) \cap P$ of patterns that match $f(t_1, \cdots, t_k)$ in time $O(k + |MS(f(t_1, \cdots, t_k)) \cap P|)$, Hoffmann and O'Donnell preprocess the patterns in P to

i. encode each pattern in PF as a distinct integer from 1 to l, and represent patterns as trees in the obvious way (implemented in compressed form as dags);
ii. compute all match sets, and encode each such set as a distinct integer from 1 to m;
iii. compute the subset of patterns in P belonging to the i^{th} match set for $i = 1, ..., m$;
iv. compute a transition function τ_f for every k–*ary* function symbol f occurring in P so that $\tau_f(ms_1, \cdots, ms_k) = MS(f(t_1, \cdots, t_k)$ whenever $ms_j = MS(t_j)$ for $j = 1, ..., k$; $\tau_v = \{v\}$, and $\tau_c = \{v, c\}$ if c is any constant appearing in PF; transition maps τ_f are implemented as multi-dimensional arrays accessed using integer encodings of match sets.

After preprocessing the patterns in P, Hoffmann and O'Donnell's algorithm solves the multi-pattern matching problem by repeatedly solving Step (1) from innermost to outermost subpattern of t. Their worst case time is $O(n + o)$ after preprocessing P. The transition table τ_f for each k–*ary* function symbol f appearing in PF uses $\Omega(m^k)$ space, where the number m of match sets can be $\Omega(2^l)$, which is expensive in practice. Their rough bound on preprocessing time is $O(l^2 m^{kmax})$.

Several approaches seem reasonable to overcome the large preprocessing and space costs. Chase [5] saves space in the transition function by eliminating some redundancy. Hoffmann and O'Donnell restrict the class of patterns to the *Simple Patterns* for which m is always small - essentially $O(l)$. For Simple Patterns that are further restricted to have arity less than or equal to two, Hoffmann and O'Donnell give an algorithm where match sets can be avoided entirely.

In the next section we show how to make Chase's preprocessing algorithm incremental and asymptotically better in both time and space. We also present a general problem decomposition technique that allows the algorithm to be tailored according to a space/time tradeoff. After that, we show how to improve the space and time for Hoffmann and O'Donnell's algorithm for binary

Simple Patterns and how to extend the algorithm to a wide subclass of Simple Patterns with unrestricted arity.

3. Incremental preprocessing

Chase was able to improve Hoffmann and O'Donnell's method by exploiting the deeper structure of the pattern set P to reduce the size of transition functions[5]. Chase's heuristic preserves the $O(1)$ per step matching time.

Let PF be the pattern forest for P, and assume that it contains variable v. For each function f appearing in PF, define projection $\Pi_f^i = \{c_i : f(c_1, \cdots, c_k) \in PF\}$ to be the set of patterns appearing as the i^{th} parameter of some f-pattern in PF. Chase made the crucial observation that the basic Bottom-Up Step (1) could be rewritten equivalently as

$$(2)\quad MS(f(t_1, \cdots, t_k)) = \{f(c_1, \cdots, c_k) \in PF \mid c_i \in MS(t_i) \cap \Pi_f^i,\ i = 1 \,..\, k\} \cup \{v\}$$

and that the size of the finite function (i.e., number of pairs stored in its graph representation) defined by the rule $\theta_f(MS(t_1) \cap \Pi_f^1, \cdots, MS(t_k) \cap \Pi_f^k) = MS(f(t_1, \cdots, t_k))$ must be no greater than Hoffmann and O'Donnell's transition function τ_f. The essential idea may be simply put: for any two finite functions f and g where f is defined by the rule $f(h(x)) = g(x)$, we know that $|f| < |g|$ as long as h is not one-to-one. Chase also provided extensive empirical evidence to show that θ_f is much smaller than τ_f in practice.

Chase's bottom-up step involves two substeps. First each Hoffmann and O'Donnell match set $MS(t_i)$ is turned into the smaller Chase match set $\mu_f^i(MS(t_i)) = MS(t_i) \cap \Pi_f^i$ for $i = 1, .., k$. Next, Chase's transition function θ_f is used to obtain the Hoffmann and O'Donnell match set $\theta_f(\mu_f^1(MS(t_1)), ..., \mu_f^k(MS(t_k)))$. Chase's *implementation* uses integer encodings for both kinds of match sets.

We will give an abstract algorithm that incrementally constructs functions μ and θ and runs asymptotically faster than Chase's algorithm. Since our algorithm is specified in terms of set and map operations, it is useful to discuss some notations and implementation details. In addition to standard mathematical notations it will sometimes be convenient to use certain unconventional dictions. Expression A **with** x abbreviates set element addition $A \cup \{x\}$ (where in this context A is interpreted as the empty set if it is undefined), and assignment $A\ op := x$ abbreviates $A := A\ op\ x$. If f is a binary relation, then **domain** $f = \{x : [x,y] \in f\}$, **range** $f = \{y : [x,y] \in f\}$, $f(x)$ denotes function application (undefined if f is multivalued at x or if $x \notin$ **domain** f), and $f\{x\}$ denotes multivalued map application with value $\{y : [x,y] \in f\}$.

By a *Set Encoding Structure* (abbr. *SE–Structure*) we mean a triple (U, A, Q) with finite universe U, *primary* set $A \subseteq 2^U$, and *secondary* set $Q \subseteq U$. SE-structures support the following five operations:

1. (*create*) Add a new set $\{z\}$ to A, and possibly add z to Q, where $z \in U$, *i.e.*,

   A with:= $\{z\}$
   Q with:= z

2. (*replace*) Replace $a \in A$ by new set a **with** z, which is denoted by,

   a with:= z

3. (*add*) Add new set a **with** z to A, and perhaps add z to Q, where $a \in A$ and $z \in U$; that is,

   A with:= a with z

4. (*query*) Retrieve set $a \cap Q$, where $a \in A$.

5. (*index*) Retrieve set $\{a \in A \mid c \in a\}$, where $c \in U$.

We will implement SE-structures using the following data structure called an *SE–Tree* (see Fig. 1). Each set a_x belonging to primary set A is associated with a unique node x in the SE-tree; that is, x 'encodes' a_x. Each node x in the tree will be uniquely associated with a set $a_x \subseteq U$, which may or may not belong to A. If a_x and a_y are sets associated with tree nodes x and y, then x is a descendent of y in the tree only if $a_y \subset a_x$. Each node is implemented by a record with five fields: a right sybling pointer, a leftmost child pointer, a Q-list pointer to a subset of Q, a pointer to the nearest ancestor with a nonempty Q-list, and a membership bit indicating whether the node corresponds to a set belonging to A or not. For each node x, which represents set a_x, Q-lists for nodes along the path from x to the root are mutually disjoint, and their union stores the set $a_x \cap Q$. Sets U and Q are implemented by a list of records. The record corresponding to each element $c \in U$ has a bit indicating membership in Q and a pointer to a list (called the c-list) of tree nodes x closest to the root such that the associated set a_x contains c.

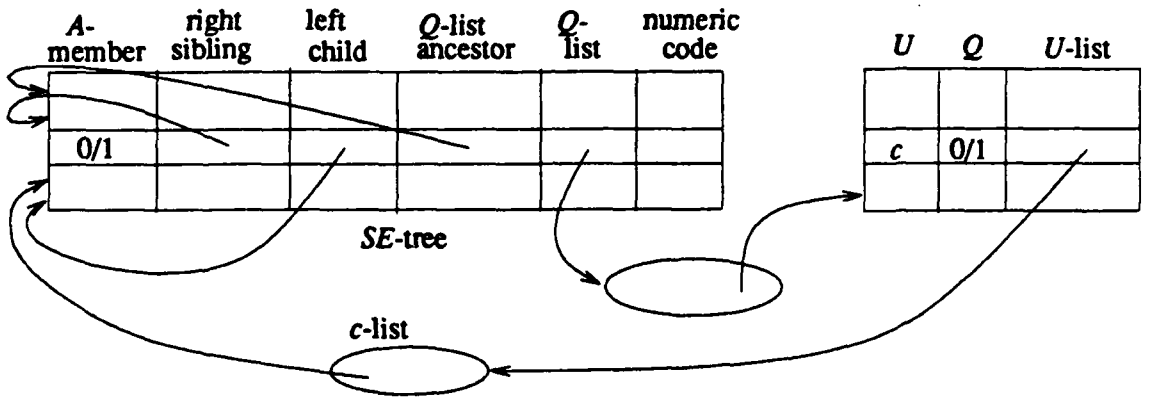

Fig.1. *SE*-structure(U, A, Q)

The *create* operation (A **with:=** $\{z\}$; Q **with:=** z) is implemented by adding a new tree root with empty sybling, child, and Q-list ancestor pointers, membership bit on, and Q-list containing z if it belongs to Q and empty if not. We also add a pointer to the newly created record in the z-list. This operation takes $O(1)$ time.

Implementation of *replace* a **with:=** z requires two cases to be considered. In the first case, called a *nondestructive replace*, the tree node x associated with a has a nonnull child pointer. Then (i) unset the membership bit in x and create a new tree node y as a child of x, (ii) if the Q-list in x is nonempty, then make the Q-list ancestor in y point to x; otherwise, make it point to the same record that the Q-list ancestor in x points to, and (iii) set the member bit in y. In the second case, where x has no children, we reuse x to represent the new set a **with** z. In this case, called a *destructive replace*, we assume that nodes x and y are the same. In either case, if z belongs to Q, add z to the Q-list for y. Finally, add y to the z-list. This operation takes $O(1)$ time.

To implement *add* A **with:=** a **with** z we let x be the tree node associated with set a. Create a new tree node y associated with set a **with** z, and make y a child of x. If the Q-list in x is nonempty, then make the Q-list ancestor in y point to x; otherwise, make it point to the same record that the Q-list ancestor in x points to, and set the member bit in y. If z belongs to Q, add z to the Q-list for y. Finally, add y to the z-list. This operation takes $O(1)$ time.

Operation *query* $a \cap Q$ is implemented as follows. If x is the tree node associated with a, then retrieve the elements in each Q-list along the path starting from x following Q-list ancestors.

The Q-lists along this path are disjoint. This operation takes $O(|a \cap Q|)$ time.

Finally, SE-trees support a straightforward implementation of *index* $\{a \in A \mid c \in a\}$. Form a list of records x, where set a_x belongs to A, occurring in subtrees rooted in nodes contained in the c-list. This operation takes $O(|\{a \in A \mid c \in a\}|)$ time, because the number of nodes x in these subtrees such that $a_x \notin A$ must be strictly less than the number of leaves in these subtrees (and all leaves represent sets belonging to A).

In order to analyze the complexity of SE-trees, we give the following definitions. For each node x in an SE-tree, define *path* (x) to be the set of nodes in the tree path from the root to x. Define *weight* (x) to be the number of elements $u \in U$ whose *u–list* contains x. Define $W_n(A) = \sum_{x \text{ is a tree node}} weight(x)$ to be the total weight of all the nodes in the tree that implements set A. Letting $des(x)$ denote the number of tree descendents of x, we can define $W_p(A) = \sum_{x \text{ is a tree node}} des(x) \times weight(x)$ to be the sum of the weights of every tree path. Clearly, $|A| \leq W_n(A) \leq W_p(A) < 2 \sum_{a \in A} |a|$. Usually, $W_n(A)$ is much smaller than $W_p(A)$.

The total space required by an SE-tree is $O(W_n(A))$, and any sequence of h of the first three operations above takes $O(h)$ time and space. Note that a naive representation of the set A will take $O(W_p(A))$ space.

We will consider useful variants of SE-structures that require minor alteration to the preceding implementation and do not affect the stated complexities. A *Simple* SE-structure is one with no secondary set. A *numeric* SE-structure is one in which the set elements of the primary set A are identified by natural numbers $1,\ldots,|A|$ (cf Fig. 1). Numeric SE-structures have special importance in connection with our second abstract datatype described next.

The main abstract datatype used in our pattern matching algorithm is the *SE–Map*, which is a partial function f: $A \rightarrow B$ from a domain set A to a range set B, where A and B are the primary sets of two SE-structures. SE-maps support the following two map operations:

1. (*modify range*) Given a set Δ and an element z, where $\Delta \subseteq A$, and z does not belong to any set in B, add z to $f(x)$ for each x belonging to Δ. This operation is denoted by,

   (for $x \in \Delta$**)**
   $f(x)$ **with:=** z
   end

2. (*modify domain*) Given a set x in the domain of f and an element z, form a new domain set x with z and map it under f to the old image $f(x)$. This operation is denoted by,

   $f(x$ **with** $z) := f(x)$

Our basic implementation of SE-maps f: $A \rightarrow B$ uses SE-tree implementations for A and B as described above. In addition, for each pair $[a, b]$ belonging to map f, if x and y are the records associated with sets a and b, then x stores a pointer to y, and y stores the size of the preimage set $f^{-1}\{b\}$. If A is part of a numeric *SE*-structure, it is sometimes useful to implement domain f as an array accessed using the numeric code of an A element as shown in Fig.2. We also make use of a *multi–dimensional* SE-map in which the domain is the cartesian product of primary sets of SE-structures.

To implement *modify range*, **(for** $x \in \Delta$**)** $f(x)$ **with:=** z **end**, we search through records associated with elements of Δ, and handle these elements according to three different cases. (1) If there are elements of Δ not belonging to the domain of f, we augment B with a new set $\{z\}$ using a *create*

operation and add the appropriate record pointers and counts.

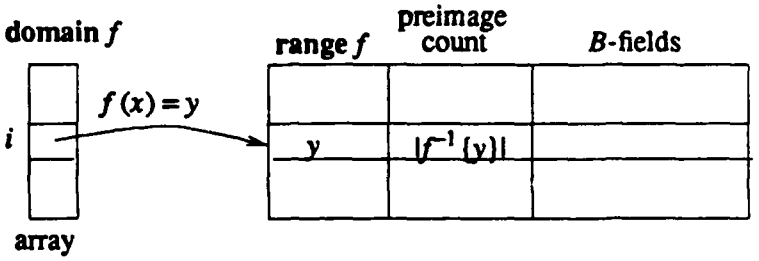

Fig.2. *SE*-map $f: A \rightarrow B$

(2) For each range element $y \in f[\Delta]$ whose preimage is entirely contained in Δ, we simply add z destructively to y using a *replace* operation. Nothing more is necessary, since B is modified implicitly. (3) For each element $y \in f[\Delta]$ not handled in case (2), we execute an *add* operation B **with**:= y **with** z, relink each element in $\Delta \cap f^{-1}\{y\}$ to the new set y **with** z, and modify preimage counts. The total cost of this operation is $O(|\Delta|)$.

The implementation of *modify domain* $f(x$ **with** $z) := f(x)$ depends on whether or not set x is modified destructively using a *replace* operation to obtain the new set x **with** z. If x is modified destructively, then the implementation is vacuous, since all operations including the modification to A are implicit. However, if x is modified nondestructively, then we need to link the new domain element x with z to the old range element $f(x)$ and increment the preimage reference count, which takes $O(1)$ time.

By an easy counting argument using the preceding analysis, we obtain the following result, which is central to the analysis of our incremental preprocessing algorithm.

LEMMA 1. *Any sequence of intermixed modify range and modify domain operations takes* $O(W_n(A)+W_n(B))$ *time and space, where A and B are at their final values.*

Let F be the set of function symbols appearing in PF. For each function $f \in F$, let $A(f)$ be its arity. Let Γ be the set of Hoffmann and O'Donnell match sets. From the above discussion, we know that the following equations hold:

$$\Gamma = \{\{v, s\}: s \in PF \mid s \text{ is a leaf}\} \cup \cup / \{\textbf{range}\ \theta_f: f \in F \mid A(f) > 0\}$$

$$\Pi_f^i = \{c_i: f(c_1, \cdots, c_k) \in PF\}$$

$$\mu_f^i = \{[m, m \cap \Pi_f^i]: m \in \Gamma\}$$

$$\theta_f = \{[[m_1, \ldots, m_k], m]: m_1 \in \textbf{range}\ \mu_f^1, \cdots, m_k \in \textbf{range}\ \mu_f^k\}$$
$$\textit{where } m = \{f(c_1, \cdots, c_k) \in PF \mid c_i \in m_i, i=1,\ldots,k\} \cup \{v\}$$

Because the preceding equations contain a cyclic dependency in which Γ depends on both PF and θ, μ depends on Γ, and θ depends on μ and PF, it would seem that a costly fixed point iteration is needed to maintain these equations when PF is modified. Fortunately, this can be avoided with careful scheduling.

The algorithm also depends on a careful logical organization of the data into SE-structures and SE-maps. Let C_f^i represent Chase match sets for $f \in F$ and $i=1,\ldots,A(f)$. Then (PF, Γ, P) is a numeric SE-structure and $(PF, C_f^i, .)$ are Simple numeric SE-structures for $f \in F$ and $i=1,\ldots,A(f)$. We also have the following SE-maps: μ_f^i: $\Gamma \rightarrow C_f^i$ for $f \in F$, $i=1,\ldots,A(f)$, and θ_f: $C_f^1 \times \ldots \times C_f^{A(f)} \rightarrow \Gamma$ for $f \in F$. Fig.3 describes the data structures used to access the main *SE*-

structures

and *SE*-maps shown in Fig.4.

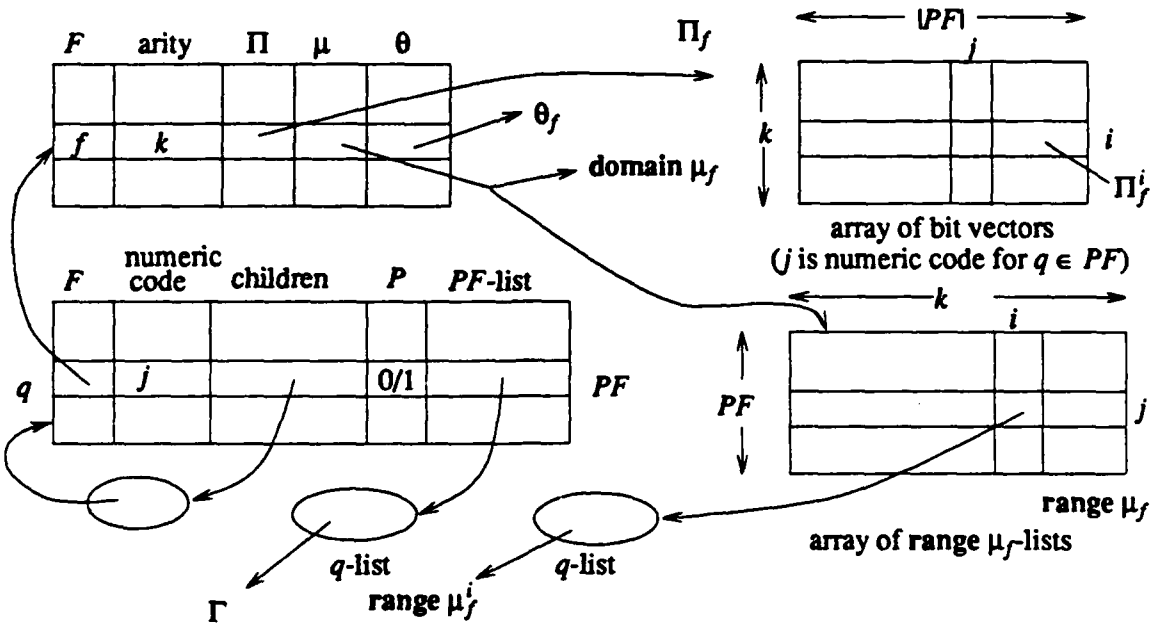

Fig.3. Core data structure

It is useful to explain our incremental algorithm in terms of three cases.

(case 1) Assume, first of all, that the set of patterns P is set to empty. It is also convenient to assume that pattern forest PF always contains v. Then we can initialize variables Γ, Π, μ, and θ as follows:

$PF := \{v\}$
$\Gamma := \{\{v\}\}$
$\Pi := \{\}$
$\mu := \{\}$
$\theta_v := \{v\}$

Next, suppose that P is augmented by a new pattern p. In order to reestablish PF, we add to PF those subpatterns of p not already in PF in an innermost-to-outermost order. Because of the order in which updates are scheduled, we know that immediately before a subpattern q of p is added to PF, either q is a leaf or all the subpatterns of q except for q itself are already in PF. More importantly we know that q is not the subpattern of any other pattern belonging to PF.

(case 2) Suppose PF is augmented with a constant symbol c. In this case, we can maintain the system of equations by executing the following code just before the modification PF with:= c:

```
Γ with:= {v} with c
COMMENT: Perform a modify domain operation on μ_f
(for [j,f,m] ∈ μ{{v}})
   μ_f(j)with c) := μ_f(j))
end
θ_c := {v,c}
```

To implement the loop efficiently, for each match set $m \in \Gamma$ we maintain a single doubly linked list threading each occurrence of m within domain μ^i_f for $f \in F, i=1,\ldots,A(f)$.

(case 3) The third and more difficult case to consider is when *PF* is augmented with pattern $f(t_1, \cdots, t_k)$, where $k>0$. Below we describe how to propagate modifications to each of the variables Γ, Π, μ, and θ separately. Recall that each of the sets Γ and range μ_f^i, $f \in F$, $i = 1..A(f)$, will be implemented as SE-trees.

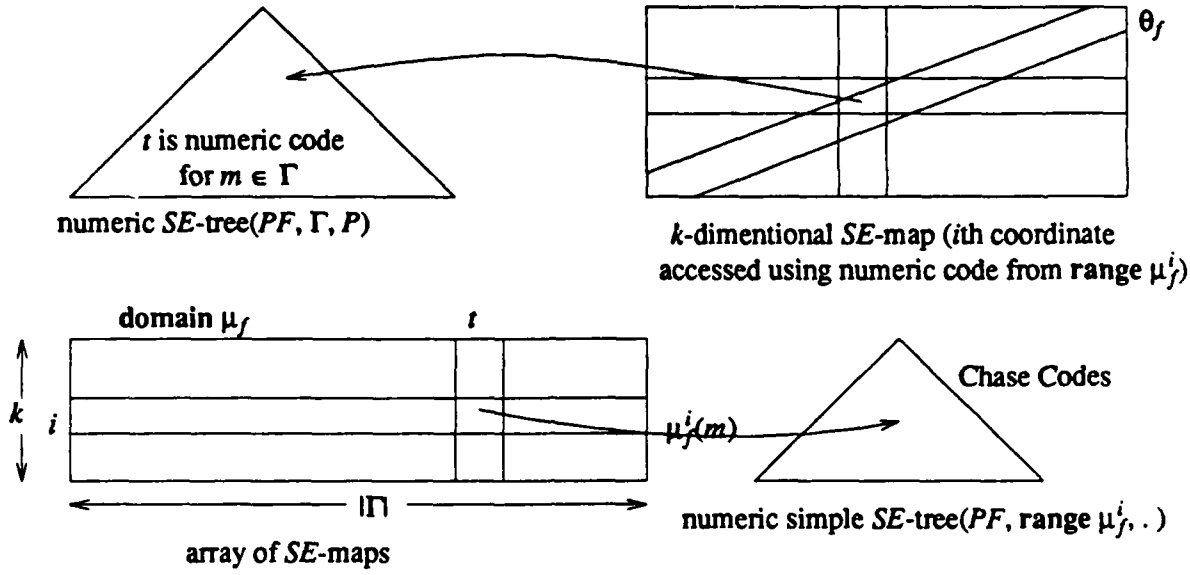

Fig.4. Data structure for θ_f and μ_f^i

1. Modify Π_f before the modification *PF* with:= $f(t_1, \cdots, t_k)$:

```
(for j = 1,..,k)
    if t_j ∉ Π_f^j then
        Π_f^j with:= t_j
    end
end
```

2. Perform a *modify range* operation on μ_f^j immediately prior to the modification Π_f^j with:= t_j of step 1:

```
(for m ∈ Γ | t_j ∈ m)
        μ_f^j(m) with:= t_j
end
```

As discussed in SE-tree operation 5, we can use the *t_j–list* to retrieve the set $\{m \in \Gamma \mid t_j \in m\}$.

3. Perform a *modify domain* operation on θ_f prior to the modification $\mu_f^j(m)$ with:= t_j of step 2 if the *modify range* in step 2 was nondestructive:

```
(for [m_1, ···, m_j, ..., m_k] ∈ domain θ_f | m_j = μ_f^j(m) )
        θ_f(m_1, ···, m_j with t_j, ···, m_k) := θ_f(m_1, ···, m_j, ···, m_k)
end
```

Here $\theta_f(m_1, \cdots, m_j \text{ with } t_j, \cdots, m_k) = \theta_f(m_1, \cdots, m_j, \cdots, m_k)$, because the pattern $f(t_1, \cdots, t_k)$ has not yet been added to *PF*, and so no *f*-pattern in *PF* has t_j as its j^{th} child. We can speedup the search by using the index $\{[m_i,[m_1, \ldots, m_k]]: [m_1, \ldots, m_k] \in \theta_f\}$. Maintaining all k such indexes for *f* along with θ_f does not change the overall asymptotic time or space.

4. Perform a *modify range* operation on θ_f just before the modification *PF* with:= $f(t_1, \cdots, t_k)$ and after the preceding three steps:

```
(for m1 ∈ range μf1, ..., mk ∈ range μfk | t1 ∈ m1, ..., tk ∈ mk)
    if [m1, ..., mk] ∉ domain θf then
        θf(m1, ..., mk) := {ν}
    end
    θf(m1, ..., mk) with:= f(t1, ···, tk)
end
```

It is important to observe that **range** μ_f^i is nonempty for i=1,...,k because of steps 1, 2, and 3. Again, we can use the t_j*–list* to search through the sets $\{m_j \in$ **range** $\mu_f^j \mid t_j \in m_j\}$ instead of the potentially much larger sets **range** μ_f^j, j=1,..,k. However, this step contains a new operation to create a k-tuple $[m_1, \ldots, m_k]$ and locate it in the domain of θ_f. Hashing is a practical solution that preserves the *Lemma* 1 space complexity but makes the time randomized. This would also make the Bottom-Up Step $O(1)$ randomized time. Our current implementation uses this approach. Another way of preserving space complexity at the expense of time is to use a balanced search tree; e.g., a red/black tree [23]. Access time is then O(**log** |**domain** θ_f|), and so is the Bottom-Up Step. Like Chase we can also use a large table to store θ_f, which doubles its size and reorganizes whenever it overflows. If each new array is allocated in unit time using the solution to exercise 2.12 of Aho, Hopcroft, and Ullman's book[1], then the *Lemma* 1 time complexity is preserved, but the run-time space requirements for θ_f are increased to be the same as Chase.

5. Modify Γ prior to the modification $\theta_f(m_1, \ldots, m_k)$ with:= $f(t_1, \cdots, t_k)$ of step 4 if the *modify range* operation of step 4 was nondestructive:

```
Γ with:= θf(m1, ···, mk) with f(t1, ···, tk)
```

Since $f(t_1, \cdots, t_k)$ is a new subpattern, no other subpattern in *PF* has $f(t_1, \cdots, t_k)$ as a subpattern. Thus no further modification is needed for Π.

6. Perform a *modify domain* operation on μ just before the modification Γ with:= $\theta_f(m_1, \cdots, m_k)$ **with** $f(t_1, \cdots, t_k)$ of step 5:

```
(for [j,g,m] ∈ μ{θf(m1, ···, mk)})
        μgj(θf(m1, ···, mk) with f(t1, ···, tk)) := μgj(θf(m1, ···, mk))
end
```

Observe that within the preceding code $\mu_g^j(\theta_f(m_1, \cdots, m_k)$ **with** $f(t_1, \cdots, t_k)) = \mu_g^j(\theta_f(m_1, \cdots, m_k))$, because $f(t_1, \cdots, t_k) \notin \Pi_g^j$. The implementation is the same as in case 2.

Now we compare the time and space complexity of Chase's algorithm and our algorithm .

THEOREM 1.

1. For each $m \in$ **domain** μ_f^j, where $f \in F$, j =1..$A(f)$, Chase's algorithm computes $\mu_f^j(m)$ in $\Omega(\min(|m|, |\Pi_f^j|))$ time, whereas our algorithm takes $O(|\mu_f^j(m)|)$ time.

2. For each $[m_1, \cdots, m_k]$ in **domain** θ_f, Chase's algorithm computes $\theta_f(m_1, \cdots, m_k)$ in $\Omega(\min(|PF|, |m_1 \times \cdots \times m_k|))$ time, whereas our algorithm takes $O(|\theta_f(m_1, \cdots, m_k)|)$ time.

3. We use $O(W_n(\Gamma))$ auxiliary space to represent the set Γ, whereas Chase uses $\Omega(W_p(\Gamma))$ space.

4. To represent the range of μ_f^j, we use $O(W_n$(**range** μ_f^j)) auxiliary space, whereas Chase uses $\Omega(W_p$(**range** μ_f^j)) space.

Proof Sketch) In both algorithms, the time complexity is dominated by the time needed to construct the tables μ^j_f and θ_f, where $f \in F$ and $j = 1..A(f)$.

1. For each $m \in$ **domain** μ^j_f, Chase's algorithm computes $\mu^j_f(m)$ by intersecting m and Π^j_f, and thus takes $\Omega(\min(|m|, |\Pi^j_f|))$ time. By Lemma 1, we spend $O(|\mu^j_f(m)|)$ time to establish the value of $\mu^j_f(m)$.

2. For each $[m_1, \cdots, m_k]$ in **domain** θ_f, Chase's algorithm computes $\theta_f(m_1, \cdots, m_k)$ by evaluating the set $\{f(c_1, \cdots, c_k) \in PF \mid [c_1, \cdots, c_k] \in m_1 \times \cdots \times m_k\}$ naively and thus takes $\Omega(\min(|PF|, |m_1 \times \cdots \times m_k|))$ time. In our algorithm , the initial value $\theta_f(m_1, \cdots, m_k)$ is $\{v\}$ by default. Then it gets new values in step 3 by copying, and increases one element at a time in step 4. Thus we spend $O(|\theta_f(m_1, \cdots, m_k)|)$ time to establish the value of $\theta_f(m_1, \cdots, m_k)$. Usually $\theta_f(m_1, \cdots, m_k)$ is much smaller than either PF or $m_1 \times \cdots \times m_k$.

3 and 4. Follows from Lemma 1.

We briefly mention that deleting patterns from P can be handled much like pattern addition, except that scheduling pattern deletion from PF is in an outermost-to-innermost subexpression order. Further, a pattern is deleted from PF only if its parent is not in PF. The deletion algorithm follows the same logic as the addition algorithm but in a backwards order to undo the effect of addition. Details will be provided in a fuller version of the paper.

4. Space/Time tradeoff

In Chase's algorithm, for each function symbol $f \in F$ of arity k, the space required for the θ_f table could be $\Omega(2^{lk})$. Here we give a method that decomposes θ_f into p tables with worst case overall space $O(p(2^{lk/p}))$ but leads to time $O(p)$ to solve the Basic Bottom-Up Step.

Let PF be partitioned into p disjoint equal size sets $PF_1, \ldots, PF_p$, and consider equations,

$$\Pi^i_{f,j} = \{c_i : f(c_1, \cdots, c_k) \in PF_j\}$$

$$\mu^i_{f,j} = \{[m, m \cap \Pi^i_{f,j}] : m \in \Gamma\}$$

$$\theta_{f,j} = \{[[m_1, \ldots, m_k], m] : m_1 \in \textbf{range}\ \mu^1_{f,j}, \cdots, m_k \in \textbf{range}\ \mu^k_{f,j}\}$$

$$\textit{where } m = \{f(c_1, \cdots, c_k) \in PF_j \mid c_i \in m_i, i = 1, \ldots, k\} \cup \{v\}$$

If $MS_i = \theta_{f,i}(\mu^1_{f,i}(MS(t_1)), \ldots, \mu^k_{f,i}(MS(t_k)))$, then we can compute disjoint unions $MS(f(t_1, \ldots, t_k)) = MS_1 \cup \cdots \cup MS_p$ in $O(p)$ time.

Consider the space required by this approach. If $r^i_{f,j} = |\textbf{range}\ \mu^i_{f,j}|$, then $r^i_{f,j} = O(2^{|\Pi^i_{f,j}|}) = O(2^{|PF_j|}) = O(2^{l/p})$, and $|\theta_{f,j}| = O(r^1_{f,j} \times \ldots \times r^k_{f,j}) = O(2^{lk/p})$. Thus, the total space storing the p match tables for function symbol f is $O(p(2^{lk/p}))$, which for $p > 1$ is asymptotically better than Chase's algorithm in the worst case.

The space required by each μ table is always $|\Gamma|$. Thus the total space for the tables $\mu^i_{f,j}$, $i = 1..k$, $j = 1..p$ is now $pk|\Gamma|$, and the total space for the μ and θ tables for function symbol f is $O(pk|\Gamma| + p(2^{lk/p}))$. When $p = \dfrac{lk}{\log(k|\Gamma|)}$, we obtain the approximate minimum $O\left(\dfrac{lk^2}{\log(k|\Gamma|)}|\Gamma|\right)$.

To further reduce the size of μ tables, we can split each $\mu^i_{f,j}$ table into p subtables $\mu^{i,t}_{f,j}$, $t = 1..p$, with **domain** $\mu^{i,t}_{f,j} = \{x \cap PF_t : x \in \Gamma\}$. Then for $x \in \Gamma$, we have $\mu^i_{f,j}(x) = \mu^{i,1}_{f,j}(x \cap PF_1) \cup \cdots \cup \mu^{i,p}_{f,j}(x \cap PF_p)\}$, which can be computed in p time. This increases the time per step to $O(p^2)$.

The total size of the p subtables is now bounded by $O(p2^{l/p})$, and the total space for the μ and θ tables for function f is $O(kp^2 2^{l/p} + p2^{lk/p})$. Since this approach is meaningful only for step time complexities better than $O(l)$, *i.e.*, $p = O(l^{1/2})$, the best upper bound we can get in this case is $O(l^{1/2}2^{ckl^{1/2}})$ for some constant c. This result also indicates that this approach is useful only when $|\Gamma| \gg 2^{l^{1/2}}$.

In a practical implementation it is not necessary for PF to be partitioned into disjoint equal size subsets. For example, we can let PF_1 be the set of patterns that are not children of any pattern, PF_i be the set of children of patterns in PF_{i-1} not contained in PF_j, where $i = 1..$*maximum height of patterns*, $j < i$. Then the tables $\mu^{i}_{j}{}^{t}_{,j}$ can be omitted for $t > j-1$. Alternatively, we can let PF_i be the set of all children of patterns in PF_{i-1}. Now the size of each subset may grow, but the tables $\mu^{i}_{j}{}^{t}_{,j}$ can be omitted for all $t \neq j-1$. It is an interesting question how to find a partition of PF that minimizes the table size for a fixed per step time bound.

5. Match set elimination

Hoffmann and O'Donnell [10] considered two subclasses of patterns for which the preprocessing and space costs for bottom-up multi-pattern matching are greatly reduced.

Definition: A set P of patterns is *Simple* if for every two distinct patterns p, $q \in PF$, either (1) $p < q$, (2) $q < p$, or (3) $\nexists$ subject $t \mid t \leq q$ and $t \leq p$.

For Simple Patterns P Hoffmann and O'Donnell observed that the partial ordering $(PF, <)$ could be represented by a directed tree (called a *subsumption* tree) with v at the root (assuming that v occurs in P). Each match set equals the set of patterns along some path in the subsumption tree from a node to the root. And every path from a node to the root determines a match set. Thus, there are only l match sets, and each one can be represented by its minimum pattern. For a function f of arity k, the transition table τ_f uses $O(l^k)$ space, a great improvement over the general case but still expensive. Hoffmann and O'Donnell also argue that most sets of patterns they have encountered in rewriting systems are Simple or can be turned into equivalent Simple sets.

Hoffmann and O'Donnell also looked at a subclass of binary Simple Patterns; i.e., Simple Patterns in which the maximum arity of any function is two. Although greatly restricted, this class is interesting, because conventional arithmetic and operations in combinatory logic have arity less than or equal to two. Also, Hoffmann and O'Donnell showed that naive transformation of patterns with arity greater than two into binary form sometimes but not always preserves the Simple Pattern property. For binary Simple Patterns they gave an algorithm requiring no transition tables, but uses $O(l^2)$ space, $O(lh^2)$ preprocessing time (h is the longest path in the subsumption tree), and $O(h^2)$ time instead of an $O(1)$ time for Step (1).

We will give a bottom-up algorithm for binary Simple Patterns (which extends to a subclass of Simple Patterns with arbitrary arity) with $O(l)$ space and $O(\log l)$ time per step. Our Preprocessing time is the same as that of Hoffmann and O'Donnell. The algorithm makes use of persistant search trees [20], and we expect it to be fast in practice.

Let PF be the pattern forest for the set P of patterns, and let T be its subsumption tree. Recall that for Simple Patterns each match set can be represented by the unique minimum pattern in the set. If p_i represents the match set for subpattern t_i of the subject, $i = 1 .. k$, then the match set for $f(t_1, \cdots, t_k)$ is represented by the pattern determined by the following formula:

(New Bottom-Up Step):

(3) $\min / (\{v\} \cup \{f(q_1, \cdots, q_k) \in PF \mid q_i \geq p_i, i = 1 .. k\})$

We call pattern $f(p_1, \cdots, p_k)$ the search argument for Step (3).

Consider any binary function f appearing in PF, and let $f(p_1, p_2)$ be the search argument for Step (3). (We will not discuss unary patterns and constants, which are simpler subcases.) We want to analyze (i) the worst case cost of performing Step (3); and (ii) the auxiliary space while executing Step (3).

An important observation is that, unlike patterns p_1 and p_2, search argument $f(p_1, p_2)$ may not belong to the subsumption tree T! Consequently, if we let **1** denote the unique maximum pattern, and if we define relation $R = \{[x, y]: f(x, y) \in PF\} \cup \{[\mathbf{1},\mathbf{1}]\}$, then we can replace Step (3) for search argument $f(p_1, p_2)$ more conveniently by,

(4) $\mathbf{min}\ / \{[x, y] \in R \mid x \geq p_1 \ \mathbf{and}\ y \geq p_2\}$

Expression (4) can be computed by locating the pair of nearest ancestors belonging to R of nodes p_1 and p_2 with respect to subsumption tree T. This characterization is meaningful because of the following proposition.

Proposition: If $[x_1, y_1]$ and $[x_2, y_2]$ are any two pairs in R and $x_1 < x_2$, then $y_2 \not< y_1$.

Proof Otherwise, P would not be Simple; i.e., we would have $f(x_1, y_2) < f(x_1, y_1)$ and $f(x_1, y_2) < f(x_2, y_2)$.

In order to compute (4) efficiently, the difficulties of two dimensional ancestor testing and searching within partially ordered sets need to be overcome. This is done by reducing the two dimensional nearest ancestor search in tree T to single dimensional searching through a totally ordered set. The essential idea is presented just below.

Let $R\{x\}$ denote the set $\{y: [x, y] \in R\}$, and let **domain** R denote the set $\{x:[x,y] \in R\}$. For each $x \in$ **domain** R, define set $S(x) = \cup_{y \geq x} R\{y\}$; for each $z \in S(x)$ define witness

$w(x, z)$ = minimum $y \geq x$ such that $[y, z] \in R$

Then we can compute (4) by performing these two queries:

(5) i. $q_0 = \mathbf{min}\ / \{x \in \mathbf{domain}\ R \mid x \geq p_1\}$
ii. $q_2 = \mathbf{min}\ / \{y \in S(q_0) \mid y \geq p_2\}$

If either q_0 or q_2 equals 1, then v is the answer; otherwise, we obtain $f(w(q_0, q_2), q_2)$.

The two queries (5) reduce computation (4) to finding single dimensional nearest ancestors and computing and storing sets $S(x)$. Nearest ancestors in trees can be computed efficiently based on the following idea. Let $pre(i)$ and $des(i)$ be the preorder number and descendent count of node i in tree T. Then node i is an ancestor of node j iff $pre(i) \leq pre(j) < pre(i) + des(i)$; also, if i and k are both ancestors of j, then i is nearer to j than k iff $pre(i) > pre(k)$.

Let Q be any subset of the nodes in T. Then for any node p in T, we can compute

(6) $\mathbf{min}\ / \{x \in Q \mid x \geq p\}$

whenever a solution exists by finding the node i in Q with maximum $pre(i)$ such that $pre(i) \leq pre(p) < pre(i) + des(i)$. To facilitate this computation we can preprocess Q as follows. For all i in Q define function $find(pre(i)) = i$ and $find(pre(i) + des(i)) = j$ such that $pre(j)$ is the maximum for which $pre(j) \leq pre(i) + des(i) < pre(j) + des(j)$ and $j \in Q$. Hence, (6) can be solved by computing $find(x)$, where x is the greatest element in **domain** *find* such that $x \leq pre(p)$.

We can store **domain** *find* as either a red/black tree [8,23] or Willard's variant of the Van Emde Boas priority queue[24,25] and obtain the following time/space bounds. Both data structures use space $O(|Q|)$. Computing query (6) costs $O(\log|Q|)$ with red/black trees, and $O(\log\log l)$ with priority queues (where l is the number of nodes in T).

Based on the preceding analysis, we can perform query (5), (i) with $O(l)$ space overall if we store all of the domains of relations R for each binary function f appearing in T either as red/black trees or Van Emde Boas priority queues. Query time is $O(\log l)$ using red/black trees, $O(\log\log l)$ with priority queues.

For query (5), (ii) we can store all of the sets $S(q_0)$ and their witnesses using a minor variant of the persistent search tree of Sarnak and Tarjan [20]. Recall that a persistent search tree can store a sequence $T_0, T_1, \cdots, T_r$ of sets, where T_0 is empty and T_i is formed from T_{i-1} by element addition or deletion for $i = 1, \cdots, r$. The data structure takes up $O(r)$ space and can support the nearest neighbor operation $pred(i, x) = \max / \{y \in T_i \mid y \le x\}$ in $O(\log r)$ worst case time.

In our application the sequence of sets is obtained by traversing the subsumption tree T in preorder, adding $R\{x\}$ as we arrive at node x from its parent, and deleting $R\{x\}$ when we go back from x to its parent. Hence, the sets $S(x)$ for x in **domain** R are included as a subsequence. Witnesses are stored using stacks inside the search tree. Since each set $R\{x\}$ is added and deleted once in forming the sequence, the size r of our sequence is just $|R|$, which is also the number of distinct patterns with root f appearing in PF. Thus, query (5), (ii) can be computed in $O(\log l)$ time, and the cumulative space for storing persistent search trees for all the binary functions f appearing in PF is just $O(l)$. Thus, we have

THEOREM 2. *Step (3) can be computed for binary Simple Patterns in $O(\log l)$ time and $O(l)$ space.*

Extending the preceding idea to functions of arbitrary arity is straightforward.

Definition: A k-ary function symbol f is *Very Simple* if there exists a k-permutation g such that for i=1,...,k-1 and every two distinct f patterns $f(x_1,...,x_k)$ and $f(y_1,...,y_k)$, $x_{g_j} \ge y_{g_j}$ j=1,..,i implies $x_{g_{i+1}} \not\le y_{g_{i+1}}$.

Any Very Simple function f in a Simple pattern forest can be handled without a transition map. Our algorithm runs in step time $O(kmax \log l)$ and total auxiliary space $O(kmax\ l)$ for all Simple functions together, where $kmax$ is the greatest arity of any Very Simple function appearing in PF.

6. Conclusion

We believe that a deeper analysis and exploitation of the structure of pattern matching can lead to further algorithmic improvements. In a subsequent paper we will report how to extend the algorithms presented here to a more complex pattern language, which is used to perform semantic analysis within RAPTS.

Acknowledgements We are grateful for stimulating discussions about pattern matching with David Chase, Chris Hoffmann, and Ken Perry. We also thank the CAAP referees for helpful comments.

References

1. Aho, A., Hopcroft, J., and Ullman, J., *Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. Borstler, J., Moncke, U., and Wilhelm, R., *Table Compression for Tree Automata*, Lehrstuhl fur Informatik III, Universitat des Saarlandes, 1987.
3. Burghardt, J., "A Tree Pattern Matching Algorithm with Reasonable Space Requirements," in *Proc. CAAP '88*, ed. M. Daudet and M. Nivat, Lecture Notes in Computer Science, vol. 299, pp. 1-15, Springer-Verlag, 1988.
4. Cai, J. and Paige, R., "The RAPTS Transformational System - A Proposal For Demonstration," in *ESOP '90 Systems Exhibition*, May 1990.

5. Chase, D., "An improvement to bottom-up tree pattern matching," in *Proc. Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 168-177, January, 1987.

6. Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B., "Programming environments based on structured Editors: the Mentor Experience," in *Interactive Programming Environments*, ed. D. Barstow, H. Shrobe, and E. Sandewall, McGraw-Hill, 1984.

7. Givler, J. and Kieburtz, R., "Schema Recognition for Program Transformations," in *ACM Symposium on LISP and Functional Programming*, pp. 74-85, Aug, 1984.

8. Guibas, L. and Sedgewick, R., "A dichromatic framework for balanced trees," in *Proc. 19th IEEE FOCS*, pp. 157-184, 1978.

9. Hatcher, P. and Christopher, T., "High-Quality Code Generation Via Bottom-Up Tree Pattern Matching," in *Proceedings 13th ACM Symposium on Principles of Programming Languages*, pp. 119-130, Jan, 1986.

10. Hoffmann, C. and O'Donnell, J., "Pattern Matching in Trees," *JACM*, vol. 29, no. 1, pp. 68-95, Jan, 1982.

11. Hoffmann, C. and O'Donnell, M., "Programming with Equations," *ACM TOPLAS*, vol. 4, no. 1, pp. 83-112, Jan., 1982.

12. Hudak, P., "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Survey*, vol. 21, no. 3, pp. 359-411, Sep. 1989.

13. Huet, G. and Lang, B., "Proving and Applying Program Transformations Expressed with Second-Order Patterns," *Acta Informatica*, vol. 11, pp. 31-55, 1978.

14. Knuth, D. and Bendix, P., "Simple Word Problems in Universal Algebras," in *Computational Problems in Abstract Algebra*, ed. Leech, J., pp. 263-297, Pergamon Press, 1970.

15. Kosaraju, S., "Efficient Tree Pattern Matching," in *Proc. FOCS '89*, Oct., 1989.

16. Maluszynski, J. and Komorowski, H. J., "Unification-free execution of logic programs," *IEEE Proceedings of symposium on logic programming*, Boston, 1985.

17. Pelegri-Llopart, E. and Graham, S., "Optimal Code Generation for Expression Trees: An Application of BURS Theory," in *Proceedings 15th ACM Symposium on Principles of Programming Languages*, pp. 294-308, Jan, 1988.

18. Pfenning, F. and Elliott, C., "Higher-Order Abstract Syntax," in *Proceedings SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pp. 199-208, June, 1988.

19. Purdom, P. and Brown, C., "Fast Many-to-one Matching Algorithm," in *Proc. RTA '85*, ed. J.-P. Jouannaud, Lecture Notes in Computer Science, vol. 202, pp. 407-416, Springer-Verlag, 1985.

20. Sarnak, N. and Tarjan, R., "Planar Point Location Using Persistent Search Trees," *CACM*, vol. 29, no. 7, pp. 669-679, July, 1986.

21. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1989.

22. Standish, T., Kibler, D., and Neighbors, J., "The Irvine Program Transformation Catalogue," Univ. of Cal. at Irvine, Dept. of Information and Computer Science, Jan, 1976.

23. Tarjan, R., *Data Structures and Network Algorithms*, SIAM, 1984.

24. Van Emde Boas, P., "Preserving Order in a Forest in Less Than Logarithmic Time and Linear Space," *IPL*, vol. 6, pp. 80-82, 1977.

25. Willard, D., "Log-Logarithmic Worst-Case Range Queries are Possible in Space O(N)," *IPL*, vol. 17, pp. 81-89, 1983.