-- --

More Efficient Bottom-Up Multi-Pattern Matching in Trees [1]

J. Cai [2] and R. Paige [3] and R. Tarjan [4]

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

Dept. of Computer Science, Princeton University, Princeton, NJ 08540
and 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. A preliminary implementation of our algorithm runs ten times faster than Chase's implementation on the hardest problem instances. Our preprocessing algorithm has the advantage of being on-line with respect to pattern additions and deletions. It also adapts to favorable input instances, and on Hoffmann and O'Donnell's class of Simple Patterns, it performs better than their special purpose algorithm tailored to this class. 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 [21], transformational programming systems [4, 15, 18, 26, 30, 35], program editing and development systems [10, 23, 32], code generator generators [14, 17, 19, 29], theorem provers [24], logic programming optimizers that attempt to replace unification with matching [27], and compilers for functional languages such as ML [34], and Haskell [22] that have equational function definitions.

---

1. An earilier version of this paper appeared in[5].
2. The research of this author was partially supported by National Science Foundation grant CCR-9002428.
3. Part of this work was done while this author was a summer faculty member at IBM T. J. Watson Research Center. The research of this author was partially supported by Office of Naval Research grant N00014-90-J-1890.
4. The research of this author at Princeton University was 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 by the Office of Naval Research contract N00014-87-K-0467.

This paper describes new solutions to a simple, basic kind of pattern matching problem of wide application. The problem is specified formally in terms of a partially ordered pattern language. Given an alphabet $\Sigma = F \cup \{v\}$ with one distinguished variable v and a finite set F of function symbols, where each such symbol $f \in F$ has arity $A(f)$, then the *linear* pattern language for Σ is the smallest set of terms that include (i) v, (ii) constant c if c is a function symbol with arity 0, and (iii) $f(p_1, ..., p_k)$, which we call an f-pattern, if f is a function symbol of arity $k>0$ and its arguments $p_1, ..., p_k$ are patterns in the language.

The set of subpatterns $sub(p)$ of a pattern p is the smallest set that contains p, and, if p is an f-pattern with $A(f) > 0$, then it also contains the subpatterns of the arguments of p. If q and p are two different patterns and q is a subpattern of p, then p is said to *properly enclose* q. The *size* of a pattern p is the number of occurrences of alphabet symbols in p.

Linear pattern matching is defined as follows. 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 $f(x_1, ..., x_k)$, p_2 is $f(y_1, ..., y_k)$ and $x_i \geq y_i$ for $i = 1, ..., k$. If $p_1 \geq p_2$, we also say that p_1 *matches* p_2 or that $[p_1, p_2]$ is a *match*. A *subsumption dag* for a set of patterns P is a directed acyclic graph that represents the reflexive transitive reduction of the partial ordering $(P, \geq)$. See the example illustrated in Fig. 1, where a is a constant and f is a binary function symbol.

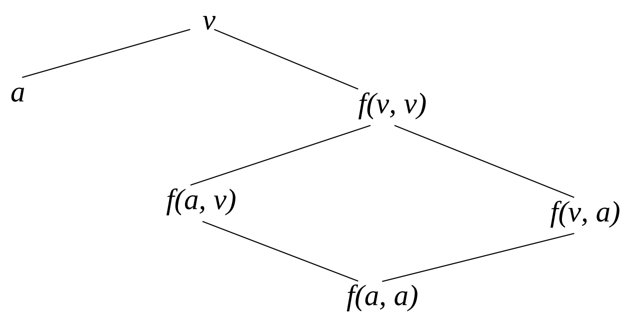

Fig. 1 Subsumption Dag

By the preceding definition variable v serves as a place holder during matching. Thus, testing whether pattern p matches pattern q is equivalent to testing whether q can be formed from p by replacing occurrences of v in p by patterns, each of which may be different.

In order to gauge performance of different pattern matching algorithms, it is useful to consider the following basic problem:

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

This paper is concerned with linear pattern matching and with solutions to the Multi-Pattern Matching Problem on a uniform cost sequential RAM [1, 28]. More complex kinds of pattern matching can be solved by extensions to our algorithms.

However, even for linear pattern matching, solving $MPTM(P,t)$ efficiently seems to be extremely difficult. The current best space-efficient top-down algorithm to solve $MPTM(P,t)$, where P contains a single pattern of size l and subject t is of size n, takes $O(n \sqrt{l}$ polylog$(l))$ time, a recent result due to Dubiner, Galil, and Magen[12], which improves Kosaraju's earlier $O(n\ l^{.75}$polylog$(l))$ time bound [25].

Bottom-up pattern matching seems to be even more difficult than top-down matching and is of special practical importance. In a seminal paper [20] Hoffmann and O'Donnell presented bottom-up linear pattern matching algorithms to solve $MPTM_P(t)$ for fixed P and subjects t without variable occurrences. They broke up the problem into two parts - (1) preprocessing P, and (2) solving $MPTM_P(t)$. Their bottom-up solution to $MPTM_P(t)$ was further broken up into repeated solutions to the following subproblem:

Bottom-Up Subproblem: Given solutions to $MPTM_P(t_i)$ $i = 1, ..., k$, solve $MPTM_P(f(t_1, ..., t_k))$.

Of course, an efficient solution to the Bottom-Up Subproblem is important to bottom-up tree rewriting, an application that concerned Hoffmann and O'Donnell. They sacrificed time and space in preprocessing P in return for an $O(k)$ time solution to the Bottom-Up Subproblem (not counting the time to produce output). Consequently, they obtained a $O(n{+}o)$ time solution to $MPTM_P(t)$, where o is the number of pairs in $MPTM_P(t)$, and n is the size of t. However, auxiliary space during computation of $MPTM_P(t)$ was excessive [20] both in theory and in practice (see Chase's empirical data [7]).

Hoffmann and O'Donnell's work has stimulated a number of papers offering heuristic space improvements [2, 3, 7, 31], and Chase's method has aroused considerable attention [7]. However, none of these papers gave proofs of theoretical improvements or promising space/time tradeoffs.

In this paper we present three new theoretical results in bottom-up linear pattern matching.

1. At the end of his CAAP '88 paper [3] Burghardt called for an efficient algorithm for preprocessing patterns P on-line with respect to additions and deletions of patterns. 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 an efficient pattern preprocessing algorithm that builds the data structures used in Chase's pattern matching algorithm in a new way. Our algorithm implements these data structures on-line with respect to additions and deletions of patterns. When our algorithm is applied repeatedly to solve batch preprocessing by adding one pattern at a time starting from the empty set, it runs asymptotically better in time and space than Chase's batch algorithm.

Hoffmann and O'Donnell obtained a worst case time bound of $O(l^2 2^{l\,(k_{max}+1)})$ for preprocessing P and a worst case auxiliary space bound of $O(l 2^{l\,k_{max}})$ both for preprocessing P and for computing $MPTM_P(t)$, where k_{max} is the greatest arity of any function symbol appearing in P. Based on our coarse analysis, Chase's algorithm improved these bounds to $O(l\;k_{max}\;2^{l\,k_{max}})$ time for preprocessing P, $O((k_{max} + 2^{k_{max}})\;2^{l\,k_{max}})$ space for preprocessing P, and $O(k_{max}\;2^{l\,k_{max}})$ space for computing $MPTM_P(t)$. Based on the same parameterization, our algorithm has the same space bounds as Chase but an improved $O(l\;2^{l\,k_{max}})$ time bound for preprocessing P. Based on a more accurate parameterization and deeper analysis of the problem, our algorithm can be observed to have a more striking theoretical advantage over Chase's algorithm.

Hoffmann and O'Donnell presented a special purpose algorithm tailored to the class of Simple patterns with polynomial worst case preprocessing time and space. Our algorithm adapts to input instances in this class and performs better in both worst case asymptotic time and space than their special purpose batch algorithm.

A prototype implementation of our algorithm is currently being used in the RAPTS transformational programming system [6] as the basis for searching, conditional rewriting, and static semantic analysis. A preliminary C implementation of our algorithm outperforms Chase's implementation of his algorithm on same data, machine, and compiler [7]; on the hardest problem instances we obtain a ten-fold speedup. We believe that a more careful implementation of our algorithm would show a more dramatic improvement.

2. Our first result is modified to obtain a general space/time tradeoff. Roughly speaking, for parameter $q \geq 1$, we trade $O(q^2)$ in time to solve the Bottom-Up Subproblem in return for auxiliary space $O(l\;k_{max}\;q^2\;2^{l/q} + q 2^{l\,k_{max}/q})$.

3. 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 and $O(\log l)$ time to solve the Bottom-Up Subproblem. A theoretical improvement to $O(\log\log l)$ time for the Bottom-Up Subproblem is obtained using Dietz's persistent form[8] of the Van Emde Boas priority queue[37]. 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 $O(l^{k_{max}+1})$ space with $O(k)$ Subproblem time for an algorithm handling all Simple Patterns. 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.

This paper is organized as follows. In the next section we discuss Hoffmann and O'Donnell's and Chase's solutions to multi-pattern matching. After that we present our on-line preprocessing algorithm, its adaptation to Simple Patterns, handling deletions, and a general space/time tradeoff. In the final section we present our third result, which deviates significantly from the earlier strategies of either Hoffmann and O'Donnell or Chase.

2. Algorithms for Bottom-up pattern matching

2.1. Notation

In addition to standard mathematical notation it will sometimes be convenient to use certain unconventional terminology. We let expression A **with** x abbreviate set element addition $A \cup \{x\}$ (where in this context A is interpreted as the empty set if it is undefined). Likewise, A **less** x represents set element deletion $A - \{x\}$. If f is a binary relation, then **domain** $f = \{x\colon [x,y] \in f\}$, **range** $f = \{y\colon [x,y] \in f\}$, and f^{-1} denotes the inverse map $\{[y,x]\colon [x,y] \in f\}$. Also, $f(x)$ denotes function application (undefined if f is multi-valued at x or if $x \notin$ **domain** f), $f\{x\}$ denotes multi-valued map application with value $\{y\colon [x,y] \in f\}$, and $f[S]$ denotes the image of set S under f with value $\{y\colon [x,y] \in f \mid x \in S\}$. The number of elements in a finite set S is denoted by $|S|$. If f is a binary relation (perhaps a function), then the number of pairs in its graph representation is denoted by $|f|$. If op is any binary, associative, and commutative operator, and $S = \{x_1, ..., x_k)$ is a set, then the APL-like reduction notation op/S denotes expansion $x_1\ op\ ...\ op\ x_k$ with an arbitrary ordering of arguments. For example, $\cup/S = \bigcup_{T \in S} T$. If S is a set, we use the **for**-loop notation **for** $x \in S$ **loop** $block(x)$ **end** to execute $block(x)$ repeatedly for each value $x \in S$ without repetition. Finally, assignment $A\ op := x$ is used to abbreviate $A := A\ op\ x$.

2.2. Hoffmann and O'Donnell's Bottom-Up Algorithm

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 $MPTM_P(q)$ of matches is found for each subpattern q of t without reference to any subpattern of t that properly encloses q.

Hoffmann and O'Donnell explain their multi-pattern matching algorithm in terms of the following two notions. 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. 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 $MS(t) = \{q \in PF \mid q \geq t\}$.

Hoffmann and O'Donnell use an equivalent recursive definition of match sets (but restricted to subjects without variable occurrences) to obtain an efficient bottom-up algorithm. The recursive rules shown below add a new rule for $MS(v)$ to Hoffmann and O'Donnell's rules so that match sets can be defined for arbitrary patterns.

$$MS(v) = \{v\}$$

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

$$\{v,c\}, \text{ when constant } c \in PF$$

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

Surprisingly, this new rule is merely a formalism, since it gives rise to the exact same collection of match sets as derived by Hoffmann and O'Donnell. This is true, because the match set $MS(p)$ for an arbitrary pattern p is identical to the match set $MS(t)$ for any pattern t formed from p by replacing occurrences of v in p by occurrences of arbitrary constants that do not belong to PF.

After determining match sets for constants and variable occurrences in subject t, Hoffmann and O'Donnell's algorithm solves the Bottom-Up Subproblem by identifying the match set for each subpattern $f(t_1, ..., t_k)$ of t based on the match sets for t_i, $i = 1, ..., k$. This task, which we call the *Bottom-Up Step*, computes expression (1) by an $O(k)$ time lookup in a k-dimensional array storing transition map τ_f, where $\tau_f(MS(t_1), ..., MS(t_k)) = MS(f(t_1, ..., t_k))$.

For consistency, throughout this paper we consider an instance of the Multi-Pattern Matching Problem with pattern set P, pattern forest PF, and subject t. We also use the following parameters:

n = size of t
Γ = the set of match sets for P
$l = |PF|$
$o = |MPTM_P(t)|$
k_{max} = maximum arity of any function symbol appearing in PF

In order to compute Step (1) and print the set $MS(f(t_1, ..., t_k)) \cap P$ of patterns that match $f(t_1, ..., t_k)$ in time $O(k + |MS(f(t_1, ..., t_k)) \cap P|)$, 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 $|\Gamma|$;
iii. compute the subset of patterns in P belonging to the i^{th} match set for $i = 1, ..., |\Gamma|$;
iv. compute a transition map τ_f for every k-ary function symbol f occurring in P so that $\tau_f(MS(t_1), ..., MS(t_k)) = MS(f(t_1, ..., t_k))$; $\tau_v = \{v\}$, and $\tau_c = \{v, c\}$ if c is any constant appearing in PF; transition maps τ_f are implemented as k-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 array storing the transition map τ_f for each k-ary function symbol f appearing in PF uses $\Omega(|\Gamma|^k)$ space, where the number $|\Gamma|$ of match sets can be $\Omega(2^l)$, which is expensive in practice. Their rough bound on preprocessing time is $O(l^2 \, |\Gamma|^{k_{max}+1})$.

2.3. Chase's Improvement

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 maps [7]. Chase's heuristic is slower by a constant factor but preserves the $O(k)$ asymptotic time for solving the Bottom-Up Subproblem.

Let PF be the pattern forest for P, and assume that PF contains variable v. For each k-ary function f appearing in PF and each $i = 1, ..., k$, Chase introduced projection $\Pi_f^i = \{q_i : f(q_1, ..., q_k) \in PF\}$ containing the set of patterns appearing as the i^{th} argument of some f-pattern in PF. Chase made the crucial observation that identity (1) could be replaced by

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

which gives rise to a modified Bottom-Up Step with improved auxiliary space.

Chase's Bottom-Up Step to compute (2) involves two substeps. First a conversion map μ_f^i is used to turn each Hoffmann and O'Donnell match set $MS(t_i)$ into a Chase match set $\mu_f^i(MS(t_i)) = MS(t_i) \cap \Pi_f^i$ for $i = 1, ..., k$. If any of these Chase match sets are empty, then $MS(f(t_1, ..., t_k)) = \{v\}$. Otherwise, Chase's transition map θ_f is used to obtain the Hoffmann and O'Donnell match set $\theta_f(\mu_f^1(MS(t_1)), ..., \mu_f^k(MS(t_k))) = MS(f(t_1, ..., t_k))$. Chase's implementation uses integer encodings for both kinds of match sets, one-dimensional arrays to implement each conversion map μ_f^i, and a k-dimensional array for θ_f.

A straightforward set theoretic argument can be used to explain why Chase's transition map utilizes space better than Hoffmann and O'Donnell's. Whenever every Chase match set $\mu_f^i(MS(t_i))$ is nonempty $i = 1, ..., k$, we know that identity $\theta_f(\mu_f^1(MS(t_1)), ..., \mu_f^k(MS(t_k))) = \tau_f(MS(t_1), ..., MS(t_k))$ holds. Consequently, if μ_f^i is not one-to-one for some i, we know that $|\theta_f| < |\tau_f|$. The essential idea may be simply put: for any two finite functions h and g where g is not one-to-one and **domain** $h \subseteq$ **range** g, then $|h| < |h \circ g|$.

Chase also provided extensive empirical evidence to show that θ_f is much smaller than τ_f in practice. Consider the example in Fig. 2. The Chase match sets associated with the first component of f are $c_1 = \{1\}$ and $c_2 = \{1,2\}$; the Chase match sets associated with the second component of f are $d_1 = \{1\}$ and $d_2 = \{1,2\}$. The Chase conversion and transition maps store 16 entries compared with 36 entries in Hoffmann and O'Donnell's transition map τ_f.

PF:	v	a	$f(a,v)$	$f(v,a)$	$f(v,v)$
Encoding:	1	2	3	4	5

Γ:	{1}	{1,2}	{1,3,5}	{1,3,4,5}	{1,4,5}	{1,5}
Encoding:	m_1	m_2	m_3	m_4	m_5	m_6

μ_f^1	
m_1	c_1
m_2	c_2
m_3	c_1
m_4	c_1
m_5	c_1
m_6	c_1

μ_f^2	
m_1	d_1
m_2	d_2
m_3	d_1
m_4	d_1
m_5	d_1
m_6	d_1

θ_f	d_1	d_2
c_1	m_6	m_5
c_2	m_3	m_4

Fig. 2 Chase's Data Organization

3. Incremental preprocessing

We will present a preprocessing algorithm that incrementally constructs maps μ and θ and is on-line with respect to modifications to P by adding or deleting patterns. When used to solve the batch preprocessing problem for fixed P, our algorithm performs asymptotically better in time and space than Chase's. It is convenient to specify our algorithm in terms of two abstract datatypes.

3.1. Abstract Sets

The first abstract datatype is called a *Set Encoding Structure* (abbr. *SE_Structure*), which is a 4-tuple (U, D, Q, τ) with finite universe U, *primary* set $D \subseteq 2^U$, *secondary* set $Q \subseteq U$, and *top* element $\tau \in U$, where $\{\tau\} \in D$, and every set within D contains τ. For simplicity we will assume for now that U and Q are fixed in order to focus on the more difficult problem of updating D. Later when we show how *SE*_structures are used by our preprocessing algorithm, details on how they are initialized and how to update U and Q will be supplied. Five operations on *SE*_structures are described below. A sixth operation *deletion* will be described later in a separate section.

1. *create*: Initialize D. This operation is performed only once for an *SE*_structure before any of the other operations below.

   $D := \{\{\tau\}\}$

2. *replace*(d,z): Replace $d \in D$ by new set d **with** z, where $z \in U$, for which we write,

   d **with**:= z

3. *add*(d,z): Add new set d **with** z to D, where $d \in D$ and $z \in U$; that is,

   D **with**:= d **with** z

4. *query*(d): Retrieve set $d \cap Q$, where $d \in D$.

5. *index*(c): Retrieve set $\{d \in D \mid c \in d\}$, where $c \in U$.

We will implement *SE*_structures using a data structure called an *SE*_tree (see Fig. 3), whose nodes correspond to distinct subsets of the universe U. Each set d_x belonging to primary set D is associated with a node x in the *SE*_tree; that is, x 'encodes' d_x. The root encodes set $\{\tau\}$. However, the set d_x associated with a node x in the tree may not necessarily belong to D. If d_x does not belong to D, it is called a *gap*. If d_x and d_y are sets associated with tree nodes x and y, then x is a descendant of y in the tree only if $d_y \subset d_x$.

*SE*_trees are implemented with two kinds of records - a *node*_record for each node in the tree and a *U*_record for each symbol in U. We will sometimes avoid distinguishing a node from its *node*_record implementation. The *node*_record for node x contains five fields: 1. a D field containing 1 if the node is not a gap and 0 if it is, 2. a *sibling* field with a pointer to the right sibling of x, 3. a *succ* field with a pointer to the leftmost child of x, 4. a *Q_query* field storing a possibly empty subset of Q, and 5. a *Q_ancestor* field with a pointer to the nearest ancestor in the tree with a nonempty *Q_query*.

For each node x the value of the subset of Q stored in the *Q_query* field is denoted by *Q_query*(x). The set is implemented by a pointer to a list of pointers to *U*_records for each symbol in *Q_query*(x). If d_x represents the set associated with node x, then the value of the collection of sets *Q_query*(y) for nodes y along the path from x to the root are mutually disjoint, and their union has the value *query*$(x) = d_x \cap Q$.

The *U*_record for symbol c has three fields: 1. a U field containing symbol c, 2. a Q field with a bit indicating whether c belongs to Q, and 3. a *D_index* field storing the subset of tree nodes x closest to the root such that the associated set d_x contains c.

We denote the subset of nodes associated with the *D_index* field in the *U*_record for symbol c by *D_index*(c). It is implemented by a pointer to a list of pointers to *node*_records for each node in *D_index*(c). Thus, the set of tree descendents of nodes belonging to *D_index*(c) has the value computed by operation *index*$(c) = \{d \in D \mid c \in d\}$.

Fig. 4 illustrates how *SE*_trees compress the space needed to store match sets. Chase's algorithm stores fifteen pattern entries to represent the collection of match sets Γ in the example shown in Fig. 2; our algorithm stores these same match sets in an *SE*_tree using only nine pattern entries.

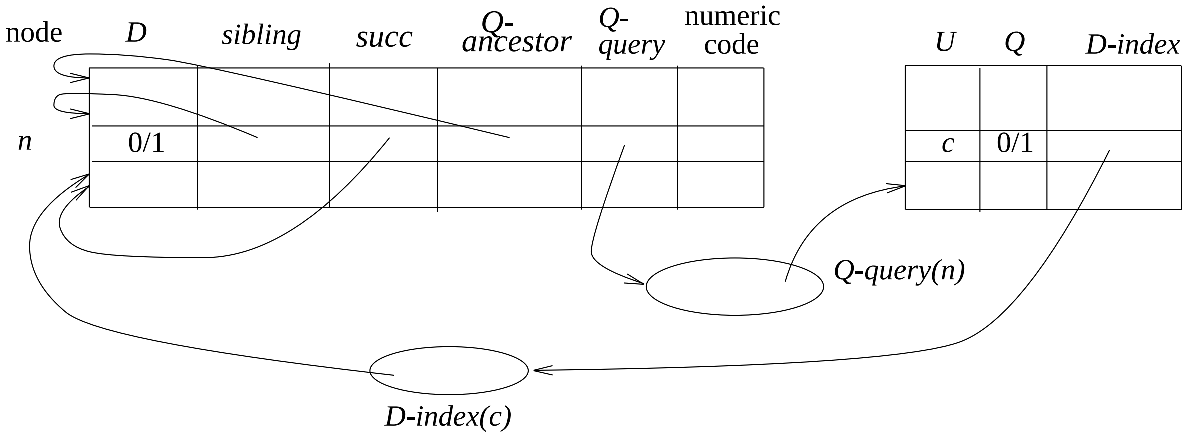

Fig.3 *SE*-Tree implementing *SE*-structure(*U*, *D*, *Q*)

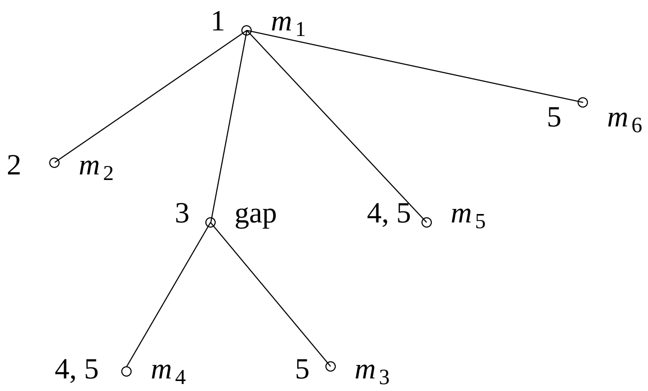

Fig. 4 *SE*-tree for (*PF*, Γ, ., *v*)

The *create* operation $D := \{\{\tau\}\}$ is implemented by adding a new tree root with empty *sibling*, *succ*, and *Q_ancestor* fields, *D* bit on , and *Q_query* containing τ if $\tau \in Q$ and empty if not. Within the *U*_record for τ we initialize *D_index* to a singleton set containing the newly created root.

Implementing the *replace* operation *d* **with**:= *z* has two cases. In the first case, called a *nondestructive replace*, the tree node *x* associated with *d* is not a leaf (i.e. *succ* is nonempty). In this case (i) unset the *D* bit in *x* (which makes *x* a gap), and create a new tree node *y* as a child of *x*, (ii) if *Q_query*(*x*) is nonempty, then make the *Q_ancestor* in *y* point to *x*; otherwise, make it point to the same record that the *Q_ancestor* in *x* points to, and (iii) set the *D* bit in *y*. In the second case, where *x* is a leaf, we reuse *x* to represent the new set *d* **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 *Q_query*(*y*). Finally, add *y* to the *D_index*(*z*).

To implement the *add* operation D **with**:= d **with** z we let x be the tree node associated with set d. Create a new tree node y associated with set d **with** z, and make y the child of x. If $Q\_query(x)$ is nonempty, then make the *Q_ancestor* in y point to x; otherwise, make it point to the same record that the *Q_ancestor* in x points to, and set the D bit in y. If z belongs to Q, add z to $Q\_query(y)$. Finally, add y to $D\_index(z)$.

The *query* operation $d \cap Q$ is implemented as follows. If x is the tree node associated with d, then retrieve the elements in each *Q_query* set along the path starting from x following *Q_ancestors*. Recall that the *Q_query* sets along this path are disjoint.

Finally, *SE*_trees support a straightforward implementation of the *index* operation $\{d \in D \mid c \in d\}$. Form a list of records x (where set d_x belongs to D) occurring in subtrees rooted in nodes belonging to $D\_index(c)$.

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 $u \in U$ such that $D\_index(u)$ contains x. Define $wn(D) = \sum_{x \text{ is a tree node}} weight(x)$ to be the total weight of all the nodes in the tree that implements set D. Letting $des(x)$ denote the number of tree descendants of x, we can define $wp(D) = \sum_{x \text{ is a tree node}} des(x) \times weight(x)$ to be the sum of the weights of every tree path. Clearly, $|D| \le wn(D) \le wp(D) < 2 \sum_{d \in D} |d|$. Usually, $wn(D)$ is much smaller than $wp(D)$.

LEMMA 1.

1. If $D\_index(c)$ is nonempty for every $c \in U$, then the total space required by an SE_tree to implement an SE_structure (U,D,Q,τ) is $O(wn(D))$. (Note that a naive representation of D can require $O(wp(D))$ space.)

2. Operations create, replace, and add each take unit time and space. A sequence of j of these operations requires $\Theta(j)$ space in the worst case.

3. Operation query $d \cap Q$ takes $O(|d \cap Q|)$ time.

4. Operation index $\{d \in D \mid c \in d\}$ takes $O(|\{d \in D \mid c \in d\}|)$ time.

Proof

1. The total space required by an *SE*_tree is dominated by the space $O(wn(D))$ needed to store all of the *D_index* sets.

2.-3. Trivial.

4. Within every subtree of an *SE*_tree the number of gaps is less than the number of nodes that are not gaps. This follows from the fact that only a nondestructive *replace* can create a gap, and this gap always has at least two children. Thus, no leaf can be a gap, and there are more leaves

than internal nodes with at least two children. ■

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 in the primary set *D* are identified by natural numbers 1, ..., $|D|$ (cf Fig. 5). Numeric *SE*_structures have special importance in connection with our second abstract datatype described next.

3.2. Abstract Maps

The second abstract datatype used in our pattern matching algorithm is the *SE_map*, which is a partial function f: $D \rightarrow R$ from a domain set *D* to a range set *R*, where *D* and *R* are the primary sets of two *SE*_structures. Let τ be the top element of *R*'s *SE*_structure, so that $\{\tau\} \in R$. It is convenient to postpone saying how *f* is initialized until later, and focus on the following two map operations:

1. *modify range*(Δ,z): Given a set Δ and an element *z*, where $\Delta \subseteq D$, and *z* does not belong to any set in *R*, add *z* to $f(x)$ for each *x* belonging to Δ. This operation can modify *SE*_set *R* as well as map *f*. It is denoted by,

```
for x ∈ Δ loop
  if x ∉ domain f then
    f(x) := {τ}
  end
  f(x) with:= z
end
```

2. *modify domain*(x,z): Given a set *x* in the domain of *f*, where (*x* **with** *z*) belongs to *D* but not to the domain of *f*, map the new set *x* **with** *z* under *f* to the old image $f(x)$. This operation modifies *f* but not *SE*_sets *D* or *R*. It is denoted by,

```
f(x with z) := f(x)
```

Our basic implementation of *SE*_maps f: $D \rightarrow R$ uses *SE*_tree implementations for *D* and *R* as described above. In addition, whenever $f(d) = r$, if *x* and *y* are the *node*_records associated with sets *d* and *r*, then in addition to the *node*_record fields previously described, *x* also stores a pointer to *y*, and *y* also stores the size of the preimage set $f^{-1}\{r\}$.

To implement *modify range*(Δ,z), we assume that the sets belonging to Δ are represented by a linked list of nodes in the *SE*_tree. In a single scan through Δ, we compute the subset Δ_1 of nodes that do not belong to the domain of *f*. For each node $x \in \Delta_1$, we store a pointer in *x* to the node *y* associated with $\{\tau\} \in R$, and increment the preimage count in *y*. Next, in a second scan through Δ, we form buckets $\Delta \cap f^{-1}\{y\}$ and bucket-counts for each $y \in f[\Delta]$. This allows us to process the elements of Δ efficiently, and to modify *SE*_set R according to two different cases. (1) For each range element $y \in f[\Delta]$ whose preimage is entirely contained in Δ (which occurs when the bucket-count for y equals the preimage count for y), we execute a *replace* operation y **with**:= z on *SE*_set R. (2) For each element $y \in f[\Delta]$ not handled in case (1), we execute an *add* operation R **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 *modify domain*(x,z) operation is only executed immediately after a set x in the domain of f is modified by either operation *add*(x,z) or *replace*(x,z). The implementation is different in each of these two cases. If x is modified by *replace*(x,z), then deleting x from D implicitly removes x from the domain of f. Hence, in this case, which we call an *implicit modify domain*, the implementation is vacuous. However, if x is modified by *add*(x,z), then we need to explicitly modify f by linking the new domain element x **with** z to the old range element $f(x)$ and increment the preimage reference count.

Analysis of the preceding implementation of *SE*_maps is straightforward and follows immediately from Lemma 1

LEMMA 2.

1. The time to execute modify range is O($|\Delta|$*).*

2. Implicit modify domain operations cost nothing. A modify domain operation that is not implicit takes O(1) time.

If D is the primary set of a numeric *SE*_structure, it is sometimes useful to implement **domain** f as an array, accessed using the numeric code of a D element as shown in Fig. 5. This idea is extended to multi-dimensional arrays used to implement the domain of a *multi-dimensional SE*_map f: $\overset{A(f)}{\underset{i=1}{\times}} D_i \rightarrow R$, where, for $i=1,..,A(f)$, D_i is the primary set of an *SE*_structure. In this case, where f has arity $k>1$, we include a dimension parameter i in operation *modify domain*$_i([x_1,...,x_k], z)$ to map f under $[x_1,..,x_i$ **with** $z,..,x_k]$ to the old image $f(x_1,...,x_k)$. We also assume a precondition that $[x_1,...,x_k] \in$ **domain** f, $[x_1,..,x_i$ **with** $z,..,x_k] \notin$ **domain** f, and x_i **with** $z \in D_i$.

The preceding algorithms adapt readily to these array implementations. However, since the domains of *SE*_maps can be augmented, we must account for overhead costs in maintaining these arrays dynamically. We implement dynamic multi-dimensional arrays by generalizing the method of unit-time array initialization found in the solution to exercise 2.12 of Aho, Hopcroft, and Ullman's book[1]. Their method permits a one-dimensional array of size s to double its size in unit time if growth space exists. If there is no growth space, we can initialize a new array of size $2s$ in unit time and then copy the old array into the new array in s steps. A multi-dimensional array that needs to double the size of one of its dimensions can be reduced to the one-dimensional case. However, if the dimension that doubles can vary, then we cannot assume that growth space is ever available.

Consider a k-dimensional array Q, where index values in dimension i for $i=1,...,k$ range from 1 to r_i, and Q is filled with entries (i.e. from the *SE*_map domain) only for index values from 1 to $e_i \le r_i$. Thus, Q has size $r_1 \times \cdots \times r_k$ and is filled with $e_1 \times \cdots \times e_k$ entries. Consider a single operation $extend_i$, which is implemented by the following two steps:

1. If $e_i = r_i$, then reallocate Q with double the range of the i^{th} dimension; i.e., assign $2r_i$ to r_i.
2. Add one to e_i.

Consider an arbitrary sequence of *extend* operations starting from an initial array with $e_i = r_i = 1$, $i=1,...,k$. The overhead in executing this sequence is the total reallocation cost in Step 1. The amortized overhead per array element is the maximum over all such sequences s of the overhead for s divided by the number of elements in the array after s is executed.

LEMMA 3. *The amortized overhead per array element in a k-dimensional array due to executing an arbitrary sequence of extend operations starting from the unit array is* $\Theta(k)$.

Proof Whenever the range of a dimension is doubled in Step 1 of an *extend* operation, we need to allocate twice the space of the current array (a unit-time operation by the method of Aho, Hopcroft, and Ullman) and to copy every entry in the old array into the new array (which can be done in time proportional to the number of entries copied by using strength reduction to access and copy an array element in unit time).

Let a *segment* be a maximal contiguous subsequence of a sequence of *extend* operations in which the last *extend* in the subsequence doubles the range of some dimension, but no other *extend* involves any such doubling. Since the last *extend* operation in a worst case sequence must double the range of some dimension, we limit our analysis to sequences of segments instead of sequences of *extend* operations. Let f_i be the number of entries in an array just after the i^{th} segment is executed; let c_i be the overhead cost due exclusively to the i^{th} segment. Clearly, $c_i < f_i$. Since doubling the range of one dimension doubles the size of the array, the array size after execution of the i^{th} segment is 2^i. Hence, we also know that $c_i \le 2^{i-1}$. Since $e_j > \frac{r_j}{2}$, $j=1,...,k$, holds after every *extend* operation, we know that $f_i > 2^{i-k}$ holds after every segment is executed $i=1,2,....$ Thus, the overhead from executing the first i segments is,

$$\sum_{j=1}^{i} c_j < k\, f_i + \sum_{j=1}^{i-k} 2^{j-1} = k\, f_i + 2^{i-k} - 1$$

and an upper bound on the overhead per array element is,

$$\frac{k\, f_i + 2^{i-k} - 1}{f_i} = k + \frac{2^{i-k} - 1}{f_i} < k + \frac{2^{i-k} - 1}{2^{i-k}} < k + 1$$

Next, we show that this bound is realizable. Starting from an initial array Q of unit size, we perform $(i+1)k$ segments as follows. First, for each dimension $j=1,...,k$ perform i segments, each doubling dimension j. Begin a new segment by performing successive *extend* operations until the entire array is filled, so that it contains 2^{ik} entries. The total overhead to this point is $2^{ik} - 1$. Next, perform one *extend* operation in each dimension, causing additional overhead costing at least $k\,2^{ik}$ for a cumulative total overhead of at least $(k+1)2^{ik}$. Thus, we obtain a lower bound $\Omega(k)$ on the overhead per array element. ■

Hoffmann and O'Donnell did not consider dynamic arrays, and their pessimistic analysis suggests that they simply preallocated enough space to accomodate worst case instances. Although Chase used algorithms that required dynamic multi-dimensional arrays, he did not analyze this cost, nor did he make use of unit-time initialization. In the next section we will use Lemma 3 to show that the overhead due to array doubling accounts for only a fraction of the total time for full pattern preprocessing. However, we do pay a price in space. Based on the proof of Lemma 3, the final space allocation of a dynamic k-dimensional array can be 2^k times the number of entries in the array. Of course, any overallocation during preprocessing is not needed for matching and can be shed.

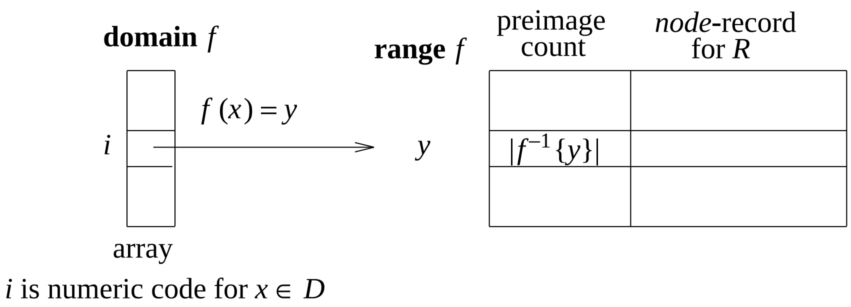

Fig.5. Implementation of *SE*_map $f: D \to R$

3.3. Abstract Algorithm

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:

$$(3)\quad \Gamma = \{\{v,\ s\}: s \in PF \mid s \textit{ is a leaf}\} \cup \cup/\{\mathbf{range}\ \theta_f: f \in F \mid A(f) > 0\}$$

$$\Pi^i_f = \{c_i: f(c_1, ..., c_k) \in PF\}$$

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

$$\theta_f = \{[[m_1, ..., m_k],\ m]: m_1 \in \mathbf{range}\ \mu^1_f, ..., m_k \in \mathbf{range}\ \mu^k_f\}$$

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

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. Recall that sets **range** μ_f^i represent Chase match sets for $f \in F$ and $i = 1, ..., A(f)$. We will use numeric *SE*_structure (PF, Γ, P, v), Simple numeric *SE*_structure (PF, **range** μ_f^i, ., v) and *SE*_map μ_f^i: $\Gamma \to$ **range** μ_f^i for $f \in F$ and $i = 1, ..., A(f)$, and multi-dimensional *SE*_map θ_f: $\times_{i=1}^{A(f)}$ **range** $\mu_f^i \to \Gamma$ for each $f \in F$. Fig. 6 describes the data structures used to access the main *SE*_structures and *SE*_maps shown in Fig. 7 (with array implementations indicated). Note that all of the *SE*_maps μ_f^i are defined on a shared *SE*_set Γ and are accessed through an array shown in Fig. 7. Note also that the *PF*_records for the *SE*_structure(PF, Γ, P, v) (see Fig. 6) spread the standard *PF* field into two fields - an *F* field for the function symbol and a *succ* field for the arguments of the function. For example, a pattern $f(t_1,...,t_k) \in PF$ would have a pointer to symbol f in the *F* field and pointers to arguments $t_1,...,t_k$ accessible from the *succ* field.

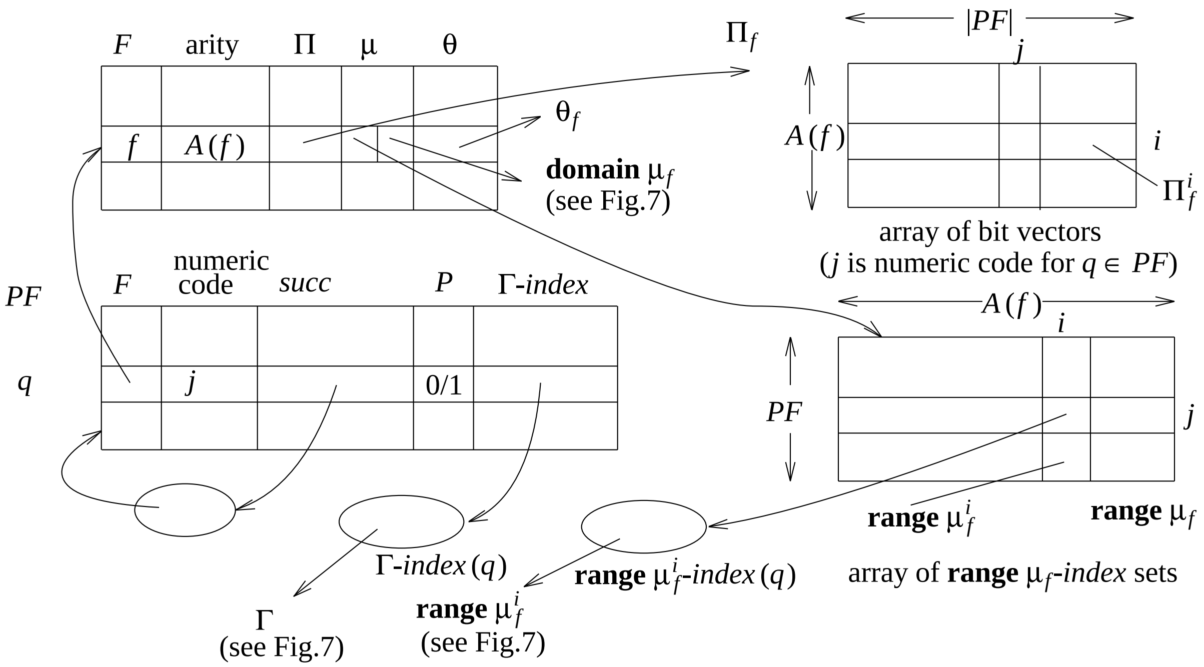

Fig.6. Core data structure

It is useful to explain our incremental algorithm in terms of three cases. Our analysis of individual operations will ignore overhead costs involving dynamic arrays. Overhead will be considered afterwards.

(case 1) Assume, first of all, that the set of patterns P is initially empty. It is also convenient to assume that pattern forest PF (but not P) always contains v. Then in $O(1)$ time and space 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 re-establish 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 already belong to 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 (3) by executing the following code just before the modification PF **with**:= c:

$\theta_c := \theta_v$
Γ **with**:= θ_c **with** c
θ_c **with**:= c
for $g \in F,\ j=1,...,A(g)$ **loop**
  if $\theta_v \in$ **domain** μ_g^j **then**
    $\mu_g^j(\theta_v$ **with** $c) := \mu_g^j(\theta_v)$
  end
end

In effect the preceding code can be implemented by performing a *modify domain*(θ_v, c) operation on μ_g^j for each $g \in F$ and $i = 1, ..., A(g)$ such that $v \in \Pi_g^j$. (Recall that $\theta_c = \theta_v$ if $c \notin PF$.) In order to implement the **for**-loop efficiently, we can use an index μ*_thread* $= \{[x,[j,g]]: g \in F,\ j=1,...,A(g),\ x \in$ **domain** $\mu_g^j\}$, which maps Hoffmann and O'Donnell match sets $m \in \Gamma$ (where in this case $m = \theta_v$) to conversion maps μ_g^j whose domain contains m. In order to update the conversion maps efficiently, we implement μ*_thread* by maintaining a single doubly linked list for every $m \in \Gamma$ threading each occurrence of m within every set **domain** μ_g^i over all $[j,g] \in \mu$*_thread*$\{\theta_v\}$. For example, in Fig. 7 the thread for match set $m \in \Gamma$ passes through entries in column t of arrays implementing **domain** μ_g^i for each $g \in F$ and $i=1,..,A(g)$ such that $m \in$ **domain** μ_g^i. Since this operation augments Γ, the arrays implementing the domains of the conversion maps can double their size. Double links allow the thread to be adjusted in unit time whenever an element in the thread is added, deleted, or moved (which occurs during array doubling). By Lemma 2 the time to perform the preceding *modify domain* operation (not including the cost of array doubling) is $O(|\mu\_thread\{\theta_v\}|)$.

(case 3) The third and more difficult case to consider is when PF is augmented with pattern $f(t_1, ..., t_k)$, where $k>0$. Below we describe how a two-stage cascade of updates can be used to propagate modifications to each of the variables Γ, Π, μ, and θ in order to re-establish equations (3). Recall that set Γ and each of the sets **range** μ_f^i, $f \in F$, $i = 1, ..., A(f)$, will be implemented as *SE*_trees.

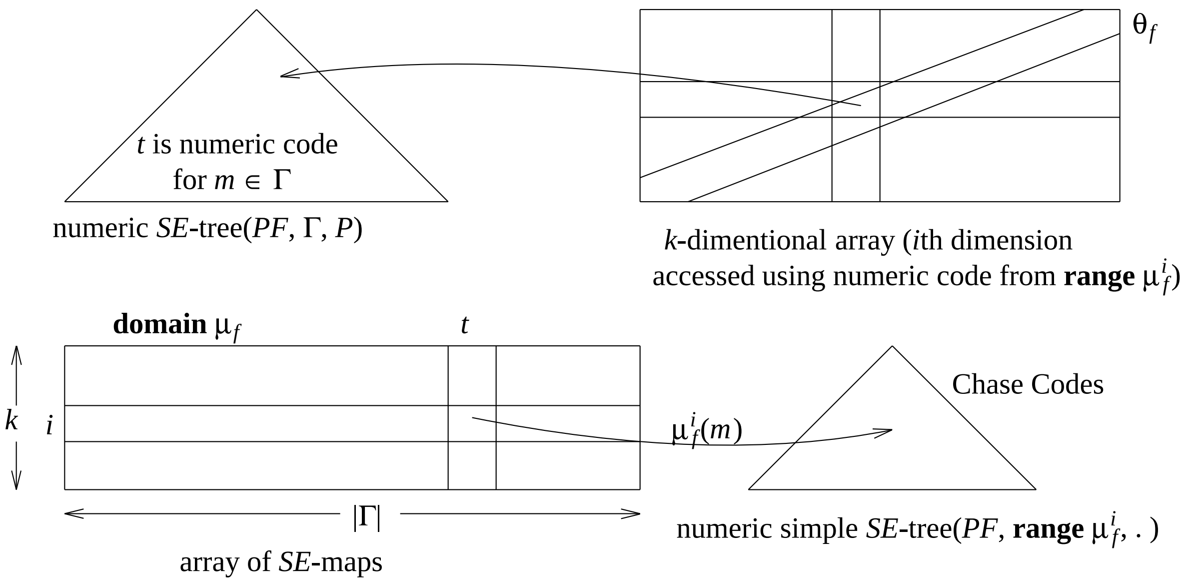

Fig.7. Data structure for θ_f and μ^i_f

1. In $O(k)$ time update Π_f before the modification PF **with**:= $f(t_1, ..., t_k)$. (Note that the array implementing Π_f can double when PF is augmented.)

```
for j = 1, ..., k loop
    if t_j ∉ Π^j_f then
        Π^j_f with:= t_j
    end
end
```

The preceding code gives rise to Stage-One updates. Each modification Π^j_f **with**:= t_j to projection Π^j_f makes the equation for Π^j_f hold for the new value of PF, but falsifies the equation for μ^j_f. In order to re-establish the equation for μ^j_f with respect to the new value of Π^j_f (but not the new value of PF), we perform a *modify range* operation on μ^j_f. However, modification to the range of μ^j_f falsifies the equation for θ_f. We re-establish this equation for the new value of Π^j_f (but not the new value of PF) by executing *modify domain* operations on θ_f. The Stage-Two updates establish all equations for the new value of PF. Details for Stage-One are given just below.

2. Perform a *modify range*($\{m \in \Gamma \mid t_j \in m\}$, t_j) operation on μ^j_f immediately prior to the modification Π^j_f **with**:= t_j of Step 1:

```
for m ∈ Γ | t_j ∈ m loop
    μ^j_f(m) with:= t_j
end
```

As discussed in SE_tree operation 5, set $\Gamma\_index(t_j)$, which is obtained from the PF_record for symbol t_j (see Fig. 6), is used to retrieve the subset $\{m \in \Gamma \mid t_j \in m\}$ of *node*_records in the numeric *SE*_tree(PF, Γ, P) (see Fig. 7). The numeric codes in these *node*_records are used to access the array for **domain** μ_f^j (see Fig. 7). By Lemmas 1 and 2, the cost of executing this step is $O(|\{m \in \Gamma \mid t_j \in m\}|)$. Although $add(\mu_f^j(m), t_j)$ operations used to implement *modify range* can cause the range of the j^{th} dimension of the array storing θ_f to double, we will charge such overhead to construction costs for θ_f.

3. Perform a $modify\ domain_j([m_1,...,m_k], t_j)$ operation for each $[m_1,...,m_k] \in$ **domain** θ_f, where $m_j = \mu_f^j(m)$, prior to each $add(\mu_f^j(m), t_j)$ operation used to implement the *modify range* of Step 2, but just after any doubling of multi-dimensional array θ_f that might result from augmenting **range** μ_f^j. Recall that the *modify domain* is implicit (i.e., implemented at no cost) whenever the *modify range* of Step 2 is implemented using *replace*.

   for $[m_1, ..., m_j, ..., m_k] \in$ **domain** $\theta_f \mid m_j = \mu_f^j(m)$ **loop**

   $\theta_f(m_1, ..., m_j$ **with** $t_j, ..., m_k) := \theta_f(m_1, ..., m_j, ..., m_k)$

   end

   Here $\theta_f(m_1, ..., m_j$ **with** $t_j, ..., m_k) = \theta_f(m_1, ..., m_j, ..., m_k)$, because the pattern $f(t_1, ..., t_k)$ has not yet been added to PF, and so no f-pattern in PF has t_j as its j^{th} argument. Since **range** θ_f is unchanged, Γ is unchanged also. Hence, the three preceding steps establish all equations relative to the new value of Π_f^i for $i=1,..,A(f)$.

This operation can be implemented naively by an exhaustive search in which every entry in a k-dimensional array implementation of θ_f with value m_j in the j-th dimension is copied to a new position differing only from the old position by index value m_j **with** t_j in the j^{th} dimension. Alternatively, if the domain of θ_f is sparse, we can speed up the search by using k indexes $\{[m_i,[m_1, ..., m_k]]: [m_1, ..., m_k] \in$ **domain** $\theta_f\}$ $i = 1, ..., k$. However, the indexes do not need to store k-tuples explictly. Each index can be implemented efficiently as lists threading elements of θ_f. That is, each Chase match set $m \in$ **range** μ_f^j has a pointer to a threaded list of entries $\theta_f(m_1, ..., m_k)$ such that $[m_1, ..., m_k] \in$ **domain** θ_f and $m_j = m$. A simple address calculation can then be used for copying. After each copy we need to update k threads for the k indexes in $O(k)$ time. Thus, our sparse implementation together with Lemma 2 lets us perform this operation in time proportional to the number of copy operations times k.

The Stage-Two updates result from modification to PF. First we execute a *modify range* operation on θ_f in order to re-establish the equation for θ_f relative to the new value of PF. Because updating the range of θ_f falsifies equations (3) for certain of the conversion maps μ_g^j, we need to perform *modify domain* operations on these maps. Consequently, after Stage-Two all of the equations (3) hold relative to the new value of PF. Details for Stage-Two are given below.

4. Perform a *modify range*($\{[m_1, ..., m_k] \in \overset{k}{\underset{i=1}{\times}}$ **range** $\mu^i_f \mid t_1 \in m_1, ..., t_k \in m_k\}$, $f(t_1, ..., t_k)$) operation on θ_f just before the modification PF **with**:= $f(t_1, ..., t_k)$ and after the preceding three steps:

> **for** $m_1 \in$ **range** μ^1_f, ..., $m_k \in$ **range** $\mu^k_f \mid t_1 \in m_1, ..., t_k \in m_k$ **loop**
>   **if** $[m_1, ..., m_k] \notin$ **domain** θ_f **then**
> $\theta_f(m_1, ..., m_k) := \theta_v$
>   **end**
> $\theta_f(m_1, ..., m_k)$ **with**:= $f(t_1, ..., t_k)$
> **end**

Whenever a new k-tuple is added to the domain of θ_f, we also need to update the k threaded indexes used in the sparse implementation for Step 3. Fortunately, this $O(k)$ maintenance operation is performed only once for each element in **domain** θ_f. We can use set **range** $\mu^i_f\_index(t_j)$ to search through the sets $\{m_j \in$ **range** $\mu^j_f \mid t_j \in m_j\}$ (which must be nonempty because t_j was previously added to some match set in Γ, and because Step 2 added t_j to **range** μ^j_f) instead of the potentially much larger sets **range** μ^j_f, $j = 1, ..., k$. However, this step contains a new operation to create a k-tuple $[m_1, ..., m_k]$ and locate it in the domain of θ_f. Hashing is a practical solution with good space utilization and good expected time. This would also make the Bottom-Up Step $O(k)$ expected time. Our current implementation uses this approach. Another way of keeping space costs down at the expense of time is to use a balanced search tree; e.g., a red/black tree [36]. Accessing the domain of the transition map θ_f then takes $O(k \log(|$**domain** $\theta_f|))$ time, and so does the Bottom-Up Step. Like Chase we can also use a k-dimensional array to store θ_f, which doubles its size and reorganizes whenever it overflows. In this case the running time for this operation is proportional to the number of times θ_f is updated by Lemmas 1 and 2. A constant factor k is avoided in each array access by using strength reduction.

5. Add a new code for a match set to Γ prior to each *add* operation that results from executing $\theta_f(m_1, ..., m_k)$ **with**:= $f(t_1, ..., t_k)$ within the *modify range* of Step 4. The old match set code is reused when the *modify range* of Step 4 is implemented using *replace*.

> Γ **with**:= $\theta_f(m_1, ..., m_k)$ **with** $f(t_1, ..., t_k)$

This operation can cause the arrays implementing the domains of conversion maps to double. Since pattern $f(t_1, ..., t_k)$ is newly added to PF, it is not a subpattern of any other pattern in PF. Thus no further modification is needed for Π.

6. Just before each *add*($\theta_f(m_1, ..., m_k)$, $f(t_1, ..., t_k)$) operation used to implement the *modify range* of Step 4, perform a *modify domain*($\theta_f(m_1, ..., m_k)$, $f(t_1, ..., t_k)$) operation

on μ_g^j for $g \in F$ and $j = 1, ..., A(f)$ such that Chase match set $\theta_f(m_1, ..., m_k)$ belongs to the domain of μ_g^j. An implicit *modify domain* is performed (at no cost) for each *replace* used to implement *modify range* in Step 4.

> **for** $g \in F,\ j = 1, ..., A(f)$ **loop**
>
>   **if** $\theta_f(m_1, ..., m_k) \in$ **domain** μ_g^j **then**
>
> $\mu_g^j(\theta_f(m_1, ..., m_k)$ **with** $f(t_1, ..., t_k)) := \mu_g^j(\theta_f(m_1, ..., m_k))$
>
>   **end**
>
> **end**

Observe that within the preceding code $\mu_g^j(\theta_f(m_1, ..., m_k)$ **with** $f(t_1, ..., t_k))$ = $\mu_g^j(\theta_f(m_1, ..., m_k))$, because $f(t_1, ..., t_k) \notin \Pi_g^j$. Since the range of μ_g^i is unchanged, the equation for θ_f remains satisfied, and no further updates are necessary. The **for**-loop is implemented efficiently using the μ*_thread* index described in case 2. By Lemma 2 the time to perform this operation is $O(|\mu\_thread\{\theta_f(m_1, ..., m_k)\}|)$.

The preceding discussion combines the correctness proof with the design description. However, we still need to analyze the performance of full batch processing, and compare our results with Chase's. In both Chase's and our algorithms the time complexity is dominated by the time needed to construct the maps μ_f^j and θ_f, where $f \in F$ and $j = 1, ..., A(f)$. However, since Chase[7] did not provide complete data structuring for an implementation and analysis, the comparison is based in part on our own data structures (not included in this paper) and analysis for his algorithm. In the following theorem we let l_g represent the total number of distinct g-patterns in PF for $g \in F$.

THEOREM 4.

1. For each $m \in \Gamma$, $f \in F$, and $j = 1, ..., A(f)$ Chase's algorithm computes $\mu_f^j(m)$ in $\Omega(\min(|m|, |\Pi_f^j|))$ time, which is improved by our algorithm to $O(|\mu_f^j(m)|)$ time when $m \in$ **domain** *μ_f^j and $O(1)$ time otherwise. By coarser analysis the total preprocessing time contributed by μ is $O(|\Gamma|\, k_{max}\, l)$ for both Chase and us.*

2. Let function symbol f have arity $k > 0$. For each $[m_1, ..., m_k] \in \times_{i=1}^{k}$ **range** *μ_f^i Chase's algorithm computes $\theta_f(m_1, ..., m_k)$ in $\Omega(\min(l_f, |m_1 \times \cdots \times m_k|)\, k)$ time if $[m_1, ..., m_k]$ belongs to* **domain** *θ_f and $O(k)$ time otherwise., Our algorithm improves this bound to $O(k + |\theta_f(m_1, ..., m_k)|)$ time if $[m_1, ..., m_k]$ belongs to* **domain** *θ_f and $O(k)$ time otherwise. By coarser analysis the total preprocessing time contributed by θ is $O(\min(l\, k_{max} 2^{l\, k_{max}},\ l\, k_{max}\, |\Gamma|^{k_{max}}))$ for Chase and $O(\min((l + k_{max}) 2^{l\, k_{max}},\ (k_{max}|F| + l)|\Gamma|^{k_{max}}))$ for us.*

3. We use $O(wn(\Gamma))$ auxiliary space to represent the set Γ, whereas Chase uses $\Omega(wp(\Gamma))$ space. When we include the threaded lists used in the sparse implementation for θ, our total *auxiliary space to store* θ *during preprocessing is roughly* $O((k_{max} + 2^{k_{max}})2^{l\,k_{max}} + l\,2^l)$. *Chase's space is comparable. The factor of* $2^{k_{max}}$ *is due to overallocating dynamic arrays, and can be shed during matching.*

4. To represent μ_f^j *we use* $O(|\Gamma| + wn(\mathbf{range}\ \mu_f^j))$ *auxiliary space, whereas Chase uses* $\Omega(|\Gamma| + wp(\mathbf{range}\ \mu_f^j))$ *space. By a coarse analysis for total preprocessing space contributed by* μ *we get a bound of* $O(l\,k_{max}\,|\Gamma|)$ *for both Chase and us.*

Proof

1. For each $m \in \Gamma$ Chase's algorithm computes $\mu_f^j(m) = m \cap \Pi_f^j$ by actually intersecting m and Π_f^j, which takes $\Omega(\min(|m|, |\Pi_f^j|))$ time. We avoid computing the intersection, and spend only $O(|\mu_f^j(m)|)$ time to establish the value of $\mu_f^j(m)$ for each $m \in$ **domain** μ_f^j. The time needed to construct these conversion maps is charged to *modify domain* operations, *modify range* operations, and overhead for dynamic arrays that store the domains of these maps. By preceding discussion of Case 2 and of Case 3, Step 6 in our algorithm, we know that the cumulative expense of executing *modify domain* operations on each conversion map μ_f^j is $O(|\mathbf{domain}\ \mu_f^j|)$, which includes the cost of maintaining index μ_thread. By preceding discussion of Case 3, Step 2 in our algorithm, a coarse upper bound on the total cost of executing *modify range* operations is $O(\sum_{m \in \mathbf{domain}\ \mu_f^j} |\mu_f^j(m)|)$ for each conversion map μ_f^j. Since the domains of these conversion maps all use dynamic 1-dimensional arrays with index values ranging from 1 to $|\Gamma|$, the overhead per array is $O(|\Gamma|)$ by Lemma 3. Combining these costs yields the first part. To obtain a coarse upper bound on the time to construct all of the conversion maps, we use the following inequalities: $|\mu_g^j(m)| \le l_g$, $|\Pi_g^j(m)| \le l_g$, $A(g) \le k_{max}$, and $|\mathbf{domain}\ \mu_g^j| \le |\Gamma|$ for $g \in F$ and $j = 1,\ldots,A(g)$. Consequently, we obtain a rough upper bound $O(k_{max}\, l_g\, |\Gamma|)$ on the cumulative charges to construct all conversion maps for each function $g \in F$. The result follows.

2. For each $[m_1, \ldots, m_k] \in \times_{i=1}^{k} \mathbf{range}\ \mu_f^i$ Chase's algorithm computes $\theta_f(m_1, \ldots, m_k)$ by evaluating the set $\{f(c_1, \ldots, c_k) \in PF \mid [c_1, \ldots, c_k] \in m_1 \times \cdots \times m_k\}$ naively, which takes $\Omega(\min(l_f, |m_1 \times \cdots \times m_k|)\,k)$ time. Roughly speaking, our algorithm assumes that the initial value of $\theta_f(m_1, \ldots, m_k)$ is $\{v\}$ by default. Then it gets new values in the *modify domain* operation of Case 3, Step 3 by copying. Each copy takes $O(k)$ time in order to maintain k threaded indexes. The value of $\theta_f(m_1, \ldots, m_k)$ increases one element at a time in the *modify range* operation of Case 3, Step 4, where an $O(1)$ time per element is a coarse upper bound. Thus, we spend $O(|\theta_f(m_1, \ldots, m_k)|)$ time from Step 4 and another $O(k)$ time from Step 3 (for maintaining sparse threaded indexes) to establish the value of $\theta_f(m_1, \ldots, m_k)$. By Lemma 3 the overhead to maintain the dynamic k-dimensional array storing θ_f is bounded by $O(k\,|\times_{j=1}^{k} \mathbf{range}\ \mu_f^j|)$, which also means that k is charged to every unit of space in the array implementing θ_f. This proves the first part. Our improvement over Chase is revealed by the following calculation:

$$|\theta_f(m_1, \ldots, m_k)| = |\{[q_1,\ldots,q_k] : f(q_1,\ldots,q_k) \in PF\} \cap \times_{i=1}^{k} m_i| \le \min(l_f, |\times_{i=1}^{k} m_i|)$$

To prove a coarse upper bound on the total time needed to construct all of the transition maps, we first prove a time bound for a single map θ_f, where f has arity k. Since $|$**range** $\mu_f^j| <= 2^{l_f}$, we can bound the overhead costs at $O(k\ 2^{k\ l_f})$. Since $|$**domain** $\theta_f| \le 2^{k\ l_f}$ the total cost in constructing θ_f is $O(|$**domain** $\theta_f|\ (l_f + k + 1) + k\ 2^{k\ l_f}) = O(2^{k\ l_f}(l_f + k))$. Alternatively, since we also know that $|$**range** $\mu_f^j| < |\Gamma|$, then another bound on overhead costs is $O(k\,|\Gamma|^k)$. Since $|$**domain** $\theta_f| \le |\Gamma|^k$, another bound on the total cost in constructing θ_f is $O(|\Gamma|^k\ (l_f + k + 1) + k\ |\Gamma|^k) = O(|\Gamma|^k(l_f + k))$. Summing over all function symbols g with arity greater than 0, we obtain the bound $O(\min(|\Gamma|^{k_{max}}(l + k_{max}\,|F\,|), (l + k_{max})2^{l\ k_{max}}))$ on the total cost of constructing all transition maps. Analysis for Chase's algorithm follows similar logic.

3 and 4. Follows from previous analysis. ■

The fine analysis in parts 1 and 2 of the preceding theorem reveal our asymptotic advantage over Chase's algorithm. The following simple calculation illustrates our potential space advantage hinted at in parts 3 and 4. When the *SE*_tree implementing Γ is a full binary tree with weight w at each node, then $wn = |\Gamma|$ and $wp = |\Gamma|\ \log|\Gamma|$.

4. Elimination of gaps

A gap in the *SE*_tree represents a set of patterns which is not a match set. In the extreme case, all the internal nodes except the root could be gaps. Thus it is useful to consider how to eliminate gaps in order to save space.

Consider the *SE*_tree implementing *SE*_structure $(PF,\ \Gamma,\ .,\ v)$. For convenience, we say a pattern q *labels* a tree node x if $x \in \Gamma$-$index(q)$. Thus, if Z is the set of patterns represented by a node z in the *SE*_tree, then $Z = \{q \in PF \mid q$ labels an ancestor of $z\}$.

We say a gap in the *SE*_tree is *maximal* if its parent is not a gap. The set of maximal gaps can be computed efficiently if we add a parent pointer to each node in the *SE*_tree. We say an *SE*_tree is *compact* if it has no gaps. If M is a finite set of patterns, we use $glb(M)$ to represent the most general pattern that is more specific than any pattern in M. Lemma 13 in the Appendix gives a necessary and sufficient condition for the existence of $glb(M)$.

Let T be an *SE*_tree implementing *SE*_structure $(PF,\ \Gamma,\ .,\ v)$, and let T' be the new *SE*_tree that results from T due to the insertion of a new pattern p into PF using the on-line preprocessing algorithm given in section 3.3. Assuming T is compact, we consider how to make T' compact also. We prove the following lemma:

LEMMA 5. *If x is a gap in T', then every descendant of x is either a gap or a leaf labeled by p.*

Proof Let X be the set of patterns represented by x. According to Lemma 15 of the Appendix, X is the match set of $glb(X)$ before p is added to PF. After p is added, x becomes a gap, and X is no longer the match set of $glb(X)$. Thus $X \cup \{p\}$ must be the match set of $glb(X)$, and $glb(X) < p$. This implies that any match set containing X must also contain p. Now consider a descendant y

of x in T' that is not labeled by p. Let Y be the set of patterns represented by y. Then $X \subseteq Y$. Since p is a new pattern, it only labels leaves. Thus Y does not contain p. Therefore Y is not a match set with respect to $PF \cup \{p\}$, and y is a gap in T'. ■

If we label the maximal gaps by p, then p is automatically added to all the sets represented by the gaps. As a result, each node whose parent is a gap should be deleted from Γ-*index*(p). If this node is a new leaf, then it is not in any Γ-*index* after it is deleted from Γ-*index*(p) and must be deleted from T' also. Once this is done, every node in T' represents some match set with respect to $PF \cup \{P\}$, and there are no gaps. Obviously, the deletion of leaves can be totally avoided if we do not add them to T' and Γ-*index*(p) in the first place.

In Section 3.2 recall the two cases for implementing the operation *modify-range*(Δ, z). (1) For each range element $y \in f[\Delta]$ whose preimage is entirely contained in Δ, we execute a *replace* operation y **with**:= z on Γ. (2) For each element $y \in f[\Delta]$ not handled in case (1), we execute an *add* operation Γ **with**:= y **with** z.

We call this implementation from Section 3.2 the *basic implementation*. To avoid introducing any gap into the *SE*_tree, we should handle Case (1) differently: for each range element $y \in f[\Delta]$ whose preimage is entirely contained in Δ, we mark y as a gap; for each maximal gap g, we execute a *destructive replace* operation g **with**:= z on Γ. Case (2) is handled as before. This new implementation of *modify range* is called the *compact implementation*.

5. Adaptation to Simple Patterns

Hoffmann and O'Donnell [20] presented an algorithm tailored to the *Simple* subclass of patterns for which the preprocessing time and space costs for bottom-up multi-pattern matching are greatly reduced.

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

For Simple Patterns P Hoffmann and O'Donnell observed that the transitive reduction of the partial ordering $(PF, <)$ forms a directed tree (which they called the *subsumption* tree) with v at the root (assuming that v occurs in *PF*). 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 θ_f uses $O(l^k)$ space, a great improvement over the general case but still expensive. Hoffmann and O'Donnell have also argued 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's special purpose algorithm for Simple Patterns runs in preprocessing time $O(k_{max}\, l^2 + |F|\; h\; l^{k_{max}})$ and space $O(l^2 + |F|\; l^{k_{max}})$, where h is the height of the subsumption tree. They also presented a test deciding whether a given set of patterns is Simple with time $O(k_{max}\, l^2)$ and space $O(l^2)$.

Our algorithm, presented in the preceding section, adapts favorably to problem instances in the class of Simple Patterns. For Simple Patterns our incremental algorithm has better asymptotic performance than Hoffmann and O'Donnell's nonincremental special purpose algorithm.

COROLLARY 6. *For Simple Patterns the preprocessing costs of our algorithm are* $O(k_{max}\, l^2 + (h + k_{max})\, l^{k_{max}})$ *time and* $O(l\; k_{max}\; (|F| + h) + (k_{max} + 2^{k_{max}})\; l^{k_{max}})$ *space. The space bound can be improved to* $O(l\; k_{max}\; (|F| + h) + k_{max}\; l^{k_{max}})$ *during matching.*

Proof Since $|\Gamma| = l$ for Simple Patterns Theorem 4 (1) says that the time contributed by all conversion maps μ is $O(k_{max}\, l^2)$.

Next we determine the time contribution of the transition maps θ. When PF is Simple, each match set, and so each Chase match set, is linear ordered in the subsumption tree. Thus, each Chase match set can be represented by its minimal element, and there can be no more than $|\Pi_f^j| \leq l_f$ such minimal elements for each $f \in F$, and each $j = 1, ..., A(f)$. Since PF is Simple, for any match set m, $|m| < h$. Then by Theorem 4 (2.), the total time bound contributed by all transition maps θ_f over all function symbols $f \in F$ is $O(\sum_{f \in F} (h + k_{max}) l_f^{k_{max}}) = O((h + k_{max})(\sum_{f \in F} l_f)^{k_{max}}) = O((h + k_{max}) l^{k_{max}})$.

By Theorem 4 (3.), the auxiliary space needed to store Γ is $O(wn(\Gamma)) = O(l\; h)$. Since, by preceding analysis, the size of each dimension of the array storing θ_f is bounded by l_f, then the space used to store all of the transition maps θ together with the threaded lists is roughly $O((k_{max} + 2^{k_{max}})\; l^{k_{max}})$. Space $2^{k_{max}}\; l^{k_{max}}$ accounts for overallocating dynamic arrays, and can be removed for matching. Since the space needed to store each conversion map μ_f^j is $O(l + wn(\mathbf{range}\; \mu_f^j)) = O(l + l_f\; h)$, then the total space utilization for all conversion maps is roughly $O(l\; k_{max}\; (|F| + h))$. ■

A slight modification to our algorithm further reduces the space needed to store Γ and each conversion map to $O(l)$ without sacrificing our time/space bounds for the general problem.

Let T be a compact *SE*_tree implementing the *SE*_structure $(PF, \Gamma, ., v)$, and let T' be the new *SE*_tree that results from T due to the insertion of p into PF using the on-line preprocessing algorithm described in section 3.3. Assume that PF is Simple. Then there are l nodes in T and l Γ-*index* sets.

We say a node x in T (therefore also in T') is *affected* if it represents a set X of patterns such that $X \cup \{p\}$ is a new match set w.r.t. $PF \cup \{p\}$. Note that if x is affected, then either x has a child labeled by p in T', or x itself is labeled by p in T'. An affected node x is maximum if all affected nodes of T are descendants of x.

We say an compact *SE*_tree is *reduced* if each of the tree nodes belongs to exactly one Γ-*index*. Thus, if T is reduced, then each Γ-*index* contains exactly one node in T, and the total space needed for the tree nodes and Γ-*index* sets is $O(l)$. We assume that T is *reduced*, and consider how to make T' reduced in case $PF \cup \{p\}$ is also Simple.

LEMMA 7. *If T is reduced, then the following properties hold.*

1. If $n_1 \in \Gamma\text{-}index(p_1)$ *and* $n_2 \in \Gamma\text{-}index(p_2)$ *are two nodes in T such that node* n_1 *is the parent of node* n_2*, then* $p_2 < p_1$.

2. T forms the subsumption tree of PF before p is added.

3. There exists a maximum affected node in T.

4. The maximum affected node is not a gap and not labeled by p in T′.

5. $PF \cup \{p\}$ *is Simple iff all the affected nodes in T except the maximum one are either gaps or leaves labeled by p in T′.*

Proof

1. Since n_1 is the parent of n_2, then there is a match set containing both p_1 and p_2. Therefore either $p_1 < p_2$ or $p_2 < p_1$. Since n_1 represents a match set containing p_1 but not p_2, then $p_2 < p_1$.

2. This follows immediately from Property 1.

3. Let n_1 and n_2 be two different affected nodes representing the two match sets N_1 and N_2 respectively before the insertion of p. Then the nearest common ancestor x of n_1 and n_2 represents the match set $X = N_1 \cap N_2$. Since n_1 and n_2 are affected, then after the insertion, there is a match set $M_1 = N_1 \cup \{p\}$ and another match set $M_2 = N_2 \cup \{p\}$. Then $M_1 \cap M_2 = X \cup \{p\}$ is also a match set (see Lemma 16, Appendix). Thus x is affected. This means that the nearest common ancestor of any two affected nodes is also affected, and there must be a unique maximum affected node.

4. Let x be a node in T. Then x has a label $q \neq p$. We need to show that if x is a gap or is labeled by p in T', then x cannot be the maximum affected node. Let X be the set of patterns in PF represented by x before adding p. Before adding p to PF, X is a match set of q. After adding p, X is no longer a match set. This means that $X \cup \{p\}$ is a match set of q. Therefore $q < p$, and there must be some match set M that contains p but not q. Let m be the node in T' representing M. Then either m or its parent is affected. Since neither m nor its parent can be a descendant of x, then x is not maximum.

5. $\Rightarrow$ Suppose $PF \cup \{p\}$ is Simple. Let $x \in \Gamma\text{-}index(q)$ be an affected node that is not a gap and not labeled by p in T'. Then x has a child m labeled by p. According to the proof of property 1, we have $q > p$. This means every match set containing p also contains q. Thus x is the maximum affected node.

$\Leftarrow$ Suppose all the affected nodes except the maximum one m are either gaps or labeled by p. Let $x \in \Gamma\text{-}index(q)$ be a node in T' such that $q \neq p$. Then x is not a new leaf. We need to show that either (1) $q > p$, (2) $q < p$, or (3) p and q cannot be in the same match set. Consider the following cases. If x is a proper descendant of m, then x is either a gap or a leaf on Γ-*index*(p). The proof of property 4 shows that $q < p$ in this case. If x is an ancestor of m, then any match set containing p also contains q, and there is at least one match set (for example, the match set represented by x) that contains q but not p. Thus $q > p$. Otherwise, x is neither an ancestor nor a descendant of m. In this case, neither descendants nor ancestors of x are labeled with p. Therefore p and q cannot be contained in the same match set. ■

The proof of property 5 also tells us the position of p in the subsumption tree of $PF \cup \{p\}$ if it is Simple: p must be a child of the pattern labeling the maximal affected node, and an ancestor of patters labeling other affected nodes. The preceding discussion justifies the following new implementation of *modify range*(Δ, z), which we call the *reduced implementation*:

> If PF is simple and there is only one element $m \in f[\Delta]$ whose preimage is not entirely contained in Δ, we execute an *add* operation Γ **with**:= m **with** z and make all the affected children of m the children of the newly created node. Otherwise, $PF \cup \{p\}$ is not Simple, and we execute the *compact implementation*.

THEOREM 2. *Whenever PF is simple, and the reduced implementation of modify range is used, then the on-line preprocessing algorithm given in Section 3.3 maintains the invariant that the SE_tree is reduced, and is consequently the subsumption tree.*

Proof Follows immediately from Lemma 7. ■

6. Pattern Deletion

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 it is not the argument of any pattern in PF. The deletion algorithm follows the same logic as the addition algorithm but in a backwards order to undo the effect of addition.

To delete a pattern p from PF, we also need to modify the *SE*_tree for *SE*_structure (PF, Γ, P, v), the range of the transition map θ_f, and the domains and ranges of all the conversion maps μ^j_f. If p has the form $f(t_1, ..., t_k)$, we have to consider whether each t_i, $i = 1, ..., k$, should also be deleted. If p is the only pattern in PF with function symbol f whose ith child is t_i, then we have to delete t_i from Π^i_f, and then modify the *SE*_tree for the range of μ^i_f and the domain of θ_f. If t_i is not in P and is not a child of any pattern in PF, then we should also delete t_i from PF recursively.

First we show how to modify *SE*_trees. Since all the *SE*_trees can be handled the same way, we will consider the *SE*_tree for *SE*_structure (PF, Γ, P, v) only. Let x be a node in the *SE*_tree representing a match set X that contains p. After p is deleted from PF, x represents the match set $X' = X - \{p\}$. The question is whether there is another node y in the *SE*_tree representing the same set X', and if so, how should we merge x and y.

To answer this question, we need two additional fields for each node x in the *SE*_tree - (1) a parent pointer $parent(x)$ pointing to the parent of x, and (2) a label list field $label\text{-}list(x)$ storing a list of patterns in *PF* that label x. The label lists are initially empty. Each time a node x is added to $\Gamma\text{-}index(p)$, pattern p is added to the right end of $label\text{-}list(x)$, and each time a node x is deleted from $\Gamma\text{-}index(p)$, p is deleted from $label\text{-}list(x)$. The leftmost element of a list is called the *head* of the list.

For convenience, we also use the following notations. We assign an integer $age(q)$ to each pattern q in *PF* so that if q is added to *PF* by the ith insertion and has not been deleted, then $age(q) = i$. Thus, for any tree node x, the patterns in $label\text{-}list(x)$ are in decreasing order of their ages from left to right. We then define the age of a tree node x to be the age of $head(label\text{-}list(x))$. Thus it makes sense to say that one node or pattern is younger or older than another. We say a node x is *normal* if it is older than all its proper descendants and has a different age than any of its siblings. It is not difficult to see that if all nodes in the *SE*_tree are normal, then different tree nodes represent different sets of patterns. Thus, our main concern is how to keep every node in the *SE*_tree normal after each deletion. The solution depends on the way that patterns are inserted. We assume that the *SE*_tree is maintained by the *basic implementation* of *modify range*. In this case, the youngest tree nodes are always the new leaves, and each internal node can get at most one new child (which is a new leaf) for each new pattern added. Therefore the *SE*_tree resulting from pure insertions has the following properties:

(1) all the tree nodes are normal;

(2) patterns labeling a parent are older then patterns labeling its children.

These two properties lead to the deletion algorithm described below.

Let p be the pattern just deleted from *PF*. Then we also delete p from the label list of each node $x \in \Gamma\text{-}index(p)$. If p is the head of $label\text{-}list(x)$, then x becomes younger and may no longer be normal. For each such possible non-normal node x with parent y, we store a pair $[x, y]$ into the set *affected* and temporarily detach x from y, leaving an *SE*_tree with only normal nodes. Then we add the detached nodes back to the *SE*_tree one by one, making sure that no non-normal node results from this addition:

```
    procedure add_back();
        for [x, y] ∈ affected loop
1           if label-list(x) = [] then
                for c ∈ children(x) loop
                    make_child(c, y);
                end loop;
            else    make_child(x, y);
            end if;
```

```
        end loop;
    end add_back;
```

On line 1, we find that *label-list* (x) is empty, which implies that x and its parent y represent the same set of patterns. Consequently, we do not add x back to the *SE*_tree, but let y adopt all the children of x. In this case, we say that x is *merged into* y. The procedure *make_child* (x, y) adds x into *children* (y), and checks whether y has another child c having the same age as x. If there is such a node c, x and c are combined. Care is taken to ensure that Property (1) and (2) are maintained for each tree node. Details are given below.

```
    procedure make_child (x, y);
2       if ∃ c ∈ children (y) | age (c) = age (x) then
            prefix := the longest common prefix of label-list (x) and label-list (c);
3           if label-list (x) = label-list (c) then
                for z ∈ children (x) loop
                    make_child(z, c);
                end loop;
            elseif  prefix = label-list (x) then
                label-list (c) −:= prefix;
                children (y) less:= c;
                make_child(c, x);
            elseif  prefix = label-list (c) then
                label-list (x) −:= prefix;
                make_child(x, c);
            else    t := newnode ();
                label-list (t) := prefix;
                make_child(t, y);
                label-list (x) −:= prefix;
                make_child(x, t);
                label-list (c) −:= prefix;
                children (y) less:= c;
                make_child(c, t);
            end if;
        else    children (y) with:= x;
            parent (x) := y;
        end if;
    end;
```

It should be clear that *make_child*(x, y) does not change the set of patterns represented by either x or y, except in line 3, where we find that x and c represent the same set of patterns and therefore merge x into c. Efficiency can be improved here if we merge the node having fewer children into the other.

Modifying the conversion maps and transition maps with respect to pattern deletion is much easier than it is with respect to pattern addition. As in pattern addition, the task of modifying a map consists of modifying the domain and range. To modify the domain of a map M, we simply delete those merged nodes or tuples containing merged nodes from **domain** M. The space released by this deletion can be put in a free list and reused later (when new nodes are added to the *SE*_tree as a result of pattern insertion). To modify the range of a map M, we simply replace each merged node x in **range** M by the node into which x is merged.

Analysis of procedure *make_child* is straightforward. The test on line 2 can be done in $O(1)$ expected time if $children(y)$ are hashed by the head of their label lists. (Maintaining the hash tables increases insertion costs by $O(1)$ space per tree node and $O(1)$ time per *add* operation.) If this test succeeds, it takes another $O(|prefix|)$ time to find the longest common prefix *prefix*. For this cost, we reduce the total size of label lists and, therefore, the total size of Γ-indices by $|prefix|$. The other costs are $O(1)$ per invocation of *make_child*, where the total number of invocations is bounded by the number of descendants of the nodes in $\Gamma\text{-}index(p)$. Thus, we pay $O(1)$ time for each match set from which p is deleted plus $O(1)$ time for each deletion of nodes from Γ-indices.

We have assumed that the *basic implementation* of *modify range* is used for pattern insertion. If we want to use the *compact implementation*, then it may happen that an ancestor has a label younger than some of its descendants' labels. We can modify the procedure *make_child* to accommodate this situation, but we do not know how to bound the time complexity. Since in general, it is not easy to check whether *PF* is Simple after each deletion, the *reduced implementation* can only be used in a very limited way: once *PF* is no longer Simple, it will not be considered Simple again until *PF* contains only one pattern v.

Finally, we want to make some comments on the effect of pattern deletions on the amortized overhead of maintaining a dynamic array. Successive deletions of elements from the domain of an array can make the array sparse. To improve the space utilization, we can halve the range of a dimension whenever the load factor of that dimension is below one fourth. Using an argument similar to the proof of Lemma 3, we can show that the amortized overhead due to an arbitrary sequence of doublings and halvings of a k-dimensional array is still $O(k)$ for each entry added to the array starting from the unit array.

7. Space/Time tradeoff

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

-- --

For each $f \in PF$, let PF_f be the set of subpatterns in PF of the form $f(x_1,...,x_k)$. Let PF_f be partitioned into q disjoint equal size sets $PF_{f,1},...,PF_{f,q}$, and consider equations,

$$\Pi^i_{f,j} = \{c_i : f(c_1, \cdots, c_k) \in PF_{f,j}\}$$
$$\mu^i_{f,j} = \{[m, m \cap \Pi^i_{f,j}] : m \in \Gamma\}$$
$$\theta_{f,j} = \{[[m_1, ..., m_k], m] : m_1 \in \textbf{range}\, \mu^1_{f,j}, \cdots, m_k \in \textbf{range}\, \mu^k_{f,j}\}$$
$$\textit{where } m = \{f(c_1, \cdots, c_k) \in PF_{f,j} \mid c_i \in m_i, i=1,...,k\} \cup \{v\}$$

We modify the Bottom-Up Step as follows. Let $t = f(t_1, ..., t_k)$ be a subject tree. Instead of computing one Chase match set $ms(t_i)$ for each child t_i of t and one Hoffmann and O'Donnell match set (H-O match set) $MS(t)$ for t, we compute q small Chase match set $ms_1(t_i)$, ..., $ms_q(t_i)$ for each child t_i of t and q small H-O match set $MS_1(t)$, ..., $MS_q(t)$ for t as follows:

$$ms_j(t_i) = \mu^i_{f,j}(MS(t_i))$$

$$MS_j(t) = \theta_{f,j}(ms_j(t_1), ..., ms_j(t_k))$$

Then we compute the H-O match set $MS(t) = MS_1(t) \cup \cdots \cup MS_q(t)$. This disjoint union can be computed in $O(q)$ time either by hashing or by table-looking. If table-lookup is used, we need a union table T_f that maps the tuple $[MS_1(t), ..., MS_q(t)]$ to the union $MS_1(t) \cup \cdots \cup MS_q(t)$. Since $MS_j(t) \subseteq PF_{f,j}$, and $| PF_{f,j} | \le l_f/q$, then the size of the T_f is $O((2^{l_f/q})^q) = O(2^{l_f})$.

Consider the space required by θ_f tables. If $r^i_{f,j} = |\textbf{range}\, \mu^i_{f,j}|$, then $r^i_{f,j} = O(2^{|\Pi^i_{f,j}|}) = O(2^{|PF_{f,j}|}) = O(2^{l_f/q})$, and $|\theta_{f,j}| = O(r^1_{f,j} \times ... \times r^k_{f,j}) = O(2^{l_f k/q})$. Thus, the total space storing the q θ_f tables is $O(q 2^{l_f k/q})$, which for $q > 1$ is asymptotically better than Chase's algorithm in the worst case.

Space for other data objects are as follows.

1. *SE*_tree for the ranges of θ tables. Since each set x in **range** $\theta_{f,j}$ is a subset of $PF_{f,j}$, then $|\textbf{range}\, \theta_{f,j}| = O(2^{|PF_{f,j}|}) = O(2^{l_f/q})$. Thus the space of the *SE*_tree encoding **range** $\theta_{f,j}$ is $O(2^{l_f/q}\, l_f/q)$. Since there are q such *SE*_trees for f, then the space for all these *SE*_trees is $O(l_f\, 2^{l_f/q})$. If the partition method is not used, we have one *SE*_tree encoding **range** θ_f which takes $O(2^{l_f}\, l_f)$ space.

2. Similarly, the *SE*_tree for **range** $\mu^i_{f,j}$ takes up $O(2^{l_f/q}\, l_f/q)$ space. There are $k\, q$ such trees for f with $O(l_f k\, 2^{l_f/q})$ cumulative space bound.

3. The space for $\mu^i_{f,j}$ is $O(\Gamma) = O(2^l)$. There are $q\, k$ such maps for f, occupying $O(qk\, 2^l)$ space altogether..

In summary, the total space for each function symbol f is $O(q\, k\, 2^l + l_f\, k\, 2^{l_f/q} + q 2^{l_f k/q})$. When $q = \dfrac{l_f\, k}{\log k + l}$, we obtain the approximate minimum $O(l_f k^2\, 2^l / l)$. Summing over all function symbols, we get the overall space bound $O(k_{max}^2\ 2^l)$.

This upper bound can be further improved by reducing the size of μ maps and the union tables. Let PF_f, $PF_{f,i}$, $\Pi^i_{f,j}$, and $\theta_{f,j}$ be defined as before. We split each map $\mu^i_{f,j}$ into smaller maps $\mu^{i,\alpha}_{f,j}$, with **domain** $\mu^{i,\alpha}_{f,j}$ = **range** α, where $\alpha = \theta_{g,s}$, $g \in F$, $s = 1, ..., q$. The size of $\mu^{i,\alpha}_{f,j}$ is $O(|$**range** $\alpha|) = O(2^{l_g/q})$. Summing over all $g \in F$, $s = 1,...,q$, $j = 1,...,q$, $i = 1,...,k$, and $f \in F$, we get an upper bound $O(k_{max}\ q^2\ |F|\ 2^{l/q})$ for the total space needed for the μ tables.

Because of this splitting, the Bottom-Up Step should be modified accordingly. Let $f(t_1, ..., t_k)$ be a subject tree. Assume that $t_i = g_i(...)$. As before, we split the H-O match set $MS(t)$ into q small H-O match sets $MS_1(t), ..., MS_q(t)$, and split each Chase match set $ms(t_i)$ into q small Chase match sets $ms_1(t_i), ..., ms_q(t_i)$. The small H-O match sets are computed as before:

$$MS_j(t) = \theta_{f,j}(ms_j(t_1), ..., ms_j(t_k))$$

but the small Chase match sets are computed differently:

$$ms_j(t_i) = ms_{j,1}(t_i) \cup \cdots \cup ms_{j,q}(t_i)$$

where $ms_{j,s}(t_i) = \mu^{i,\beta}_{f,j}(MS_s(t_i))$, $\beta = \theta_{g_i,s}$. Again, the disjoint union $ms_{j,1}(t_i) \cup \cdots \cup ms_{j,q}(t_i)$ can be computed in $O(q)$ time either by hashing or table-lookup. This increases the time per step to q^2. If table-lookup is used for the disjoint union, then we need a union table $T^{i,g_i}_{f,j}$ to map the tuple $[ms_{j,1}(t_i), ..., ms_{j,q}(t_i)]$ into the union $ms_{j,1}(t_i) \cup \cdots \cup ms_{j,q}(t_i)$. Let $\gamma_{g_i,s} = PF_{g_i,s} \cap \Pi^i_{f,j}$, and let $\gamma_{g_i} = PF_{g_i} \cap \Pi^i_{f,j}$. Since $ms_{j,s}(t_i) = \mu^{i,\beta}_{f,j}(MS_s(t_i)) \subseteq PF_{g_i,s} \cap \Pi^i_{f,j} = \gamma_{g_i,s}$, then the size of $T^{i,g_i}_{f,j}$ is $O(2^{|\gamma_{g_i,1}|} \times ... \times 2^{|\gamma_{g_i,q}|} = O(2^{|\gamma_{g_i,1} \cup \cdots \cup \gamma_{g_i,q}|} = O(2^{|\gamma_{g_i}|})$. Summing over all the function symbols g_i, we obtain the upper bound $O(2^{|\Pi^i_{f,j}|}) = O(2^{l_f/q})$ for the total space for the union tables of the form $T^{i,*}_{f,j}$. Summing this space further for $j = 1,...,q$, $i = 1,...,k$, and $f \in F$, we get upper bound $O(k_{max}\ q\ 2^{l/q})$ for the total space for all the union tables, which is less than the space for the μ tables.

The space for the *SE*_trees and θ tables are roughly as before. Thus the overall space is $O(k_{max}\ q^2\ l\ 2^{l/q} + q 2^{l\ k_{max}/q})$.

Since this approach is meaningful only for step time complexities better than $O(l)$, *i.e.*, $q = O(\sqrt{l})$, the best upper bound we can get in this case is roughly $O(\sqrt{l}\ 2^{c\ k\ \sqrt{l}})$ for some constant c. This result also indicates that this approach is useful only when $|\Gamma| \gg 2^{\sqrt{l}}$.

In a practical implementation it is not necessary for PF_f 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 maps $\mu^{i,\alpha}_{f,j}$ can be omitted for $\alpha = \theta_{g,s}$, where $s > j+1$. 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 maps $\mu^{i,\alpha}_{f,j}$ can be omitted for all $\alpha = \theta_{g,s}$, where $s \neq j+1$. It is an interesting question how to find a partition of PF that minimizes the map size for a fixed per step time bound.

8. Match set elimination

Aiming for a bottom-up pattern matching method that utilizes space efficiently by avoiding conversion and transition maps, Hoffmann and O'Donnell[20] investigated the subclass of binary Simple Patterns; i.e., Simple Patterns in which the maximum arity of any function symbol 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. For binary Simple Patterns they gave an algorithm requiring no transition maps, but uses $O(l^2)$ space for both preprocessing and computing $MPTM_P$, $O(l\ h^2)$ preprocessing time (recall that h is the longest path in the subsumption tree), and $O(h^2)$ time instead of $O(1)$ time for the Bottom-Up Step (1).

Hoffmann and O'Donnell also considered reducing pattern forests to equivalent binary form. For each function symbol $f \in F$ where $A(f) > 2$, introduce a new function symbol two_f. Transformation $T1$ replaces each f-pattern $f(x_1,...,x_k)$ in PF where $k>2$ by $f(two_f(x_1,x_2),x_3,...,x_k)$. Transformation $T2$ applies $T1$ repeatedly until it can no longer be applied.

The following lemma states without proof that transformation $T1$ and, consequently, $T2$ is correct.

LEMMA 8. *Let patterns p' and q' be formed from patterns $p,q \in PF$ by transformation $T1$. Then $p \le q$ if and only if $p' \le q'$.*

Although it is correct, transformation $T2$ may not always be usefully applied. Hoffmann and O'Donnell showed that $T2$ sometimes, but not always, preserves the Simple Pattern property. For a counterexample, consider two patterns $f(x_1,x_2,x_3)$ and $f(y_1,y_2,y_3)$ in a Simple pattern forest PF. If $x_1 > y_1$, $x_2 < y_2$, and x_3 is incomparable with y_3, then the new pattern forest that results from transformation $T1$ would not be Simple, because of $two_f(x_1,x_2)$ and $two_f(y_1,y_2)$.

However, we can give an interesting class of pattern forests that remains Simple under transformation $T2$. A Simple pattern forest PF is *Very Simple* if for each k-ary function symbol $f \in F$ with $k>2$ and every two distinct f-patterns $f(x_1,...,x_k)$ and $f(y_1,...,y_k)$, we know that $\forall\ i=1,...,k\text{-}1\ ((\forall\ j=1,..,i \mid x_j \ge y_j) \text{ and } (\exists\ j=1,..,i \mid x_j > y_j)) \rightarrow x_{i+1} \not< y_{i+1})$.

LEMMA 9. *Pattern forest PF is Very Simple if and only if the pattern forest PF' that results from transformation $T1$ is Very Simple.*

LEMMA 10. *If a Binary pattern forest is Simple, then it is also Very Simple.*

Proof If $f(x_1, y_1)$ and $f(x_2, y_2)$ are any two f-patterns in PF and $x_1 < x_2$, then $y_2 \not< y_1$. Otherwise, PF would not be Simple; that is, we would have $f(x_1, y_2) < f(x_1, y_1)$ and $f(x_1, y_2) < f(x_2, y_2)$. ■

The preceding lemmas show that

THEOREM 11. *The class of Very Simple Patterns is the largest subclass of Simple Patterns for which transformation $T2$ preserves Pattern Forests that are Simple.*

We will give a bottom-up algorithm for binary Simple Patterns with $O(l)$ space to compute $MPTM_P$ and $O(\log l)$ time to compute the Bottom-Up Step. Our preprocessing time and space are the same as that of Hoffmann and O'Donnell. The algorithm makes use of persistant search trees [33], and we expect it to be fast in practice.

Let PF be the pattern forest for a set P of Simple 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):

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

We call pattern $f(p_1, \cdots, p_k)$ the search argument for Step (4).

Consider any binary function f appearing in PF, and let $f(p_1, p_2)$ be the search argument for Step (4). (We will not discuss unary patterns and constants, which are simpler subcases.) We want to analyze (i) the worst case cost of performing Step (4); and (ii) the auxiliary space while executing Step (4).

An important observation is that, unlike patterns p_1 and p_2, search argument $f(p_1, p_2)$ might not belong to the subsumption tree T! Consequently, if we let $\mathbf{1} > v$ denote a new 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 (4) for search argument $f(p_1, p_2)$ more conveniently by,

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

If $[\mathbf{1},\mathbf{1}]$ is the answer to query (5), then v is the answer to query (4); otherwise, if $[w, z]$ is the answer to (5) for $w,z \neq \mathbf{1}$, then $f(w, z)$ is the answer to query (4).

Expression (5) can be computed by locating the pair belonging to R of nearest ancestors of nodes p_1 and p_2 with respect to subsumption tree T. This characterization is meaningful because of Lemma 10.

In order to compute (5) 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 $\Pi^1_f = \{x : [x,y] \in R\}$. For each $x \in$ **domain** R, define set $S(x) = \cup_{y \geq x} R\{y\}$; for each $z \in S(x)$ define witness

$w(x, z) = \mathbf{min}/\{y \in R^{-1}\{z\} \mid y \geq x\}$

Then we can compute (5) by performing these three queries:

(6) i. $q_1 = \mathbf{min}\ /\{x \in \mathbf{domain}\ R \mid x \geq p_1\}$
ii. $q_2 = \mathbf{min}\ /\{y \in S(q_1) \mid y \geq p_2\})$
iii. $q_3 = w(q_1, q_{2)}$

If either q_1 or q_2 equals **1**, then v is the answer to query (4); otherwise, the answer is $f(q_3, q_2)$.

The three queries (6) reduce computation (5) 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 descendant 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 than k to j 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

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

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 $find(pre(i)) = i$. Also, for all $i \in Q$, whenever there is no $j \in Q$ such that $pre(j) = pre(i) + des(i)$, then we define $find(pre(i) + des(i))$ to be the nearest ancestor k of i belonging to Q; i.e., the node $k \in Q$ such that $pre(k)$ is the maximum for which $pre(k) \leq pre(i) + des(i) < pre(k) + des(k)$. Hence, (7) can be solved by computing $find(z)$, where z is the greatest element in **domain** *find* such that $z \leq pre(p)$.

We can store **domain** *find* as either a red/black tree [16, 36] or Willard's variant of the Van Emde Boas priority queue[37, 38] and obtain the following time/space bounds. Both data structures use space $O(|Q|)$. Computing query (7) costs $O(\log |Q|)$ time with red/black trees, and $O(\log\log l)$ time with priority queues (where l is the number of nodes in T).

Based on the preceding analysis, we can perform query (6), (i.) with $O(l)$ cumulative space if we store all of the domains of *find* maps for each binary function $f \in F$ either as red/black trees or Van Emde Boas priority queues. Query time is $O(\log l_f)$ using red/black trees, $O(\log\log l)$ with priority queues.

To facilitate query (6), (ii.) and (iii.) we can combine witnesses and *find* maps as follows. Let $find_x$ be the *find* map for $S(x)$. Then define

$findw_x(z) = [w(x, find_x(z)),\ find_x(z)]$

We can store all these $findw_x$ maps for each $x \in \Pi^1_f$ using a minor variant of the persistent search tree of Sarnak and Tarjan [33] (see also[11]). Recall that a persistent search tree can store a sequence $T_0, T_1, \cdots, T_r$ of sets, where T_0 is empty in space $O(s)$ in which $s = \sum_{i=0}^{r-1} |T_i \Delta T_{i+1}|$ and Δ represents symmetric difference. It can also support the nearest neighbor operation

$pred(i, x) = \mathbf{max} / \{y \in T_i \mid y \le x\}$ in $O(\log s)$ worst case time.

Consider the sequence $[findw_x\colon x \in \Pi_f^1]$ of maps ordered according to a preordering of Π_f^1 relative to the subsumption tree (where the empty set is implicitly the first member of the sequence). Let us store this sequence in a persistent search tree using domain values of the *findw* maps as keys. Since the sum of the sizes of the symmetric differences of successive maps in the sequence is bounded by $O(\sum_{x \in \Pi_f^1} |R\{x\}|) = O(l_f)$, then query (6) (ii.) and (iii.) can be solved in time $O(\log l_f)$. If q_1 is the answer to query (6) (i.), then the pair $[q_3, q_2] = findw_{q_1}(z)$ solves queries (6) (ii.) and (iii.), where z is the greatest element in **domain** $findw_{q_1}$ such that $z \le pre(p_2)$. The cumulative space for storing *findw* maps in persistent search trees for all the binary functions $f \in F$ is just $O(l)$.

Preprocessing for solving (6) involves constructing the subsumption tree T and computing preorder and descendant numbers (*pre* and *des*) for each of its nodes. Hoffmann and O'Donnell's Algorithm **A**[20] decides whether *PF* is Simple and computes the transitive closure of T in time $O(l^2 k_{max})$ and space $O(l^2)$. It is straightforward to modify their algorithm to decide whether *PF* is Very Simple and to produce T without changing the theoretical complexity. Once T is available, *pre* and *des* can be computed in $O(l)$ steps (since T has l nodes).

Preprocessing for (6) (i.) involves computing *find* maps over set Π_f^1 for each function symbol $f \in F$. If Π_f^1 is preordered with respect to T, we can compute the *find* map for f as follows. Pass through Π_f^1 in linear time, defining $find(pre(x))$ to be x for each $x \in \Pi_f^1$ encountered. Recall that we also need to compute the nearest ancestor of x in Π_f^1 to be assigned to $find(pre(x)+des(x))$ whenever $pre(x)+des(x)$ is not the preorder number of some node $y \in \Pi_f^1$. These ancestors can be computed by stacking the anticipated number $pre(x)+des(x)$ together with the ancestor of x while searching through Π_f^1. It may be helpful to think of the algorithm as processing numbers $pre(x)$ as left parentheses (which are all distinct) and $pre(x)+des(x)$ as balancing right parentheses (which need not be distinct for different values of x). Details are given below.

--Initialize *ancestor* to be the artificial top element of all nodes in T

--whose preorder number is less than $old\_num = l+1$; its ancestor

--*old_ancestor* is undefined

ancestor := **1**

--Handle left boundary of T using 0 as an artificial preorder number

$find(0) := ancestor$

$old\_ancestor := undefined$

$old\_num := l+1$

$stack := [old\_ancestor, old\_num]$

for $x \in \Pi_f^1$ **loop**

  (**while** $old\_num < pre(x)$)

```
--old_num is the pre(y)+des(y) for some node y whose nearest ancestor
--is old_ancestor
     find(old_num) := old_ancestor
     pop stack
     ancestor := old_ancestor
     [old_num,old_ancestor] := top stack
   end
   if old_num=pre(x) then
     pop stack
     ancestor := old_ancestor
     [old_num,old_ancestor] := top stack
   end
   find(pre(x)):=x
   if old_num≠pre(x)+des(x) then
--This test guarantees that old_num values in successive stack entries must
--be distinct
     old_num := pre(x)+des(x)
     old_ancestor := ancestor
     push [old_num,old_ancestor] onto stack
   end
   ancestor:=x
end
(while old_ancestor≠undefined)
-- Process remaining right boundaries
  find(old_num):=old_ancestor
  pop stack
  [old_num,old_ancestor] := top stack
end
```

Algorithm **Compute_find**

Algorithm **Compute_find** runs in $O(l_f)$ steps. If we fold in the code to store **domain** *find* in a red/black tree, the preprocessing time is $O(l_f \log l_f)$. In a single preorder traversal of T, we can preorder the elements of Π_f^1 for all functions $f \in F$ in $O(l)$ time. The total preprocessing time to compute red/black trees storing *find* maps for all of the function symbols together is then $O(l \log l)$. Using Willard's data structure instead takes expected time $O(l \log l)$ or worst case time $O(l^2 \log l)$, because it depends on perfect hashing[13].

Preprocessing for (6) (ii.) and (iii.) involves computing $findw_x$ maps over sets $S(x)$ for each $x \in \Pi_f^1$. We compute these maps according to a preorder search through Π_f^1. Suppose that y comes immediately after x in the preordering of Π_f^1. Suppose also that $findw_x$ is computed for set $S(x)$. Our goal is to compute $findw_y$ for set $S(y)$ by performing modifications to $findw_x$. It suffices to consider two cases: (1) where y is a proper descendant of x in T, and (2) otherwise.

If y is a proper descendant of x, then $S(y) = S(x) \cup R\{y\}$. In this case we can compute $findw_y$ by first computing the *find* map *local_find* for $R\{y\}$ using Algorithm **Compute_find**. By Lemma 10 we know that no element in $R\{y\}$ is a proper ancestor of any element in $S(x)$. Hence, for each $z \in$ **domain** *local_find*, if $local\_find(z) \neq \mathbf{1}$, we perform the update $findw_x(z) := [y,\ local\_find(z)]$, where y will always be a new witness; otherwise if $local\_find(z) = \mathbf{1}$, we perform a nearest neighbor query $a = \mathbf{max}/\{u \in$ **domain** $findw_x \mid u \leq z\}$, and assign $findw_x(a)$ to $findw_x(z)$. The map that results from these operations is $findw_y$.

If we assume that dummy value 0 is the first element in Π_f^1 in which $S(0)$ and $findw_0$ are both empty, then the preceding approach for case (1) can be used to compute the first *findw* map in our sequence. To handle case (2) in which y is not a proper descendant of x, we first find the closest proper ancestor u of y in Π_f^1, where dummy value 0 is regarded as a proper ancestor of every other node. Next, we update $findw_x$ to form a copy of $findw_u$. Finally, we update the copy of $findw_u$ to obtain $findw_y$ using the method for case (1).

More specifically, let Δ be the union of the sets $(\{pre(i) : i \in R\{y\}\} \cup \{pre(i) + des(i) : i \in R\{y\}\})$ for all y coming after u among the preordered elements of Π_f^1 such that y is an ancestor of x. Then for each $z \in \Delta$, if it belongs to the domain of $findw_u$, assign $findw_u(z)$ to $findw_x(w)$; otherwise, remove z from **domain** $findw_x$. This step turns $findw_x$ into a copy of $findw_u$. Map $findw_y$ is obtained by further modifying $findw_x$ according to the method for case (1).

If we use a persistent red/black tree, the total preprocessing costs to compute and store maps *findw* for function f are $O(l_f)$ space and $O(l_f \log l_f)$ time. The cumulative preprocessing costs to compute these maps for all functions $f \in F$ is thus $O(l)$ space and $O(l \log l)$ time.

Summing up the preceding discussion, we have

THEOREM 12. *Bottom-Up Step (4) can be computed for binary Simple Patterns in $O(\log l)$ time and $O(l)$ auxiliary space. Total preprocessing costs are $O(l^2)$ time and space.*

The reduction of Very Simple pattern forests *PF* to binary form introduces $O(|F|)$ new function symbols and $O(k_{max}\, l)$ new subpatterns. The cost of the Bottom-Up Step is approximately doubled, while the theoretical complexity for preprocessing remains unchanged.

The time bound for Theorem 12 can be improved to $O((\log\log l)^2)$ by using a persistent form of the Van Emde Boas queues to answer queries of type (6) (ii.) and (iii.). These queues can be made persistent by applying the results of Dietz[8]. Dietz's result gives as an immediate corollary that the Van Emde Boas structure can be made persistent at a time cost of a factor of loglog l per operation. The time for lookups is worst-case; the preprocessing time (to build the data structure) is expected, because it depends hashing[9, 13] to keep the space down. The space bound remains $O(l)$. The expected preprocessing time is $O(l(\log\log l)^2)$.

9. Conclusion

We believe that a deeper analysis and exploitation of the structure of pattern matching can lead to further algorithmic improvements. It might also be worthwhile to consider hybrid pattern matching methods that combine our different algorithms. The main open problem in the method of match set elimination is to compute the subsumption tree *T* in better time and space than Hoffmann and O'Donnell's Algorithm **A**. Of course, this method would also benefit from improvements in construction time for persistent Van Emde Boas priority queues. In a subsequent paper we will report how to extend our algorithms to a more complex pattern language, which is used to perform semantic analysis within RAPTS.

Acknowledgements We are grateful for helpful discussions with David Chase, Mike Fredman, Fritz Henglein, Chris Hoffmann, Ken Perry, and Dan Willard. We also thank the CAAP referees.

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. Cai, J., Paige, R., and Tarjan, R., ''More Efficient Bottom Up Tree Pattern Matching,'' in *Proc. CAAP 90,* ed. A. Arnold, Lecture Notes in Computer Science, vol. 431, pp. 72-86, Springer-Verlag, 1990.
6. Cai, J., Facon, P., Henglein, F., Paige, R., and Schonberg, E., ''Type Transformation and Data Structure Choice,'' in *Proc.TC2 Working Conf. on Constructing Programs from Specifications,* May, 1991.
7. 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.
8. Dietz, P., ''Fully Persistent Arrays,'' *J. Algorithms,* submitted 1990.
9. Dietzfelbinger, M., et. al., ''Dynamic Perfect Hashing: Upper and Lower Bounds,'' in *Proc. IEEE 29th FOCS,* pp. 524-531, Oct., 1988.
10. 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.
11. Driscoll, J., Sarnak, N., Sleator, D., and Tarjan, R., ''Making Data Structures Persistent,'' in *Proc. 8th ACM STOC,* pp. 109 - 121, ACM, May, 1986.
12. Dubiner, M., Galil, Z., and Magen, E., ''Faster Tree Pattern Matching,'' in *Proc. 31st IEEE FOCS '90,* Oct., 1990.
13. Fredman, M., Komlos, J., and Szemeredi, E., ''Storing a Sparse Table with O(1) Worst Case Access Time,'' *JACM,* vol. 31, no. 3, pp. 538-544, July, 1984.
14. Giegerich, R. and Schmal, K., ''Code Selection Techniques: Pattern Matching, Tree Parsing, and Inversion of Derivors,'' in *Proc. ESOP '88,* Lecture Notes in Computer Science, vol. 300, pp. 245-268, Springer-Verlag, 1988..
15. Givler, J. and Kieburtz, R., ''Schema Recognition for Program Transformations,'' in *ACM Symposium on LISP and Functional Programming,* pp. 74-85, Aug, 1984.
16. Guibas, L. and Sedgewick, R., ''A dichromatic framework for balanced trees,'' in *Proc. 19th IEEE FOCS,* pp. 157-184, 1978.
17. 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.

18. Heckmann, R., ''A Functional Language for the Specification of Complex Tree Transformations,'' in *Proc. ESOP '88*, Lecture Notes in Computer Science, vol. 300, pp. 175-190, Springer-Verlag, 1988.

19. Henry, R., *Encoding Optimal Pattern Selection in a Table-Driven Bottom-Up Tree-Pattern Matcher,* Computer Science Dept., U. of Washington, 1989. Technical Report

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

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

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

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

24. 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.

25. Kosaraju, S., ''Efficient Tree Pattern Matching,'' in *Proc. 30th IEEE FOCS '89*, Oct., 1989.

26. Lipps, P., Moncke, U., and Wilhelm,R., ''OPTRAN - A Language/System for the Specification of Program Transformations: System Overview and Experiences,'' in *Proc. 2nd CCHSC Workshop*, ed. D. Hammer, Lecture Notes in Computer Science, vol. 371, pp. 52-65, Springer-Verlag, 1988.

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

28. Mehlhorn, K., *Sorting and Searching,* Data Structures and Algorithms, 1, Springer-Verlag, 1984.

29. 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.

30. 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.

31. 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.

32. Reps, T. and Teitelbaum, T., *The Synthesizer Generator,* Springer-Verlag, 1988.

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

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

35. 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.

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

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

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

Appendix: Pattern Algebra

Let U be the set of all possible patterns. Let $\geq$ be the *more general than* relation between patterns. The relation $\geq$ is reflexive, transitive and antisymmetric. Thus $(U, \geq)$ is a partial order. It is easy to see that any subset S of U has a least upper bound $lub(S)$ in U. Thus, U is a join lattice with v being the maximum element. But it is not a lattice.

Two patterns in U are said to be *compatible* if they have a lower bound in U. We can show that

LEMMA 13. *A finite set of patterns P has a greatest lower bound $glb(P)$ in U iff these patterns are mutually compatible.*

Proof The *only* if part is trivial. We need only to prove the if part.

Basis: P contains at least one leaf x. If $x = v$, then $glb(P) = glb(\{x, glb(P - \{x\})\}) = glb(P - \{x\})$. If x is a constant, then $glb(P) = x$.

Induction: Suppose that all patterns in P have the same function symbol f with arity $k > 0$. Then $glb(P) = f(glb(\{x_1\colon f(x_1, ..., x_k) \in \mathrm{P}\}), ..., glb(\{x_k\colon f(x_1, ..., x_k) \in P\}))$. ■

Let PF be any finite subset of U. A subset M of PF is called a match set (wrt PF) if there is a pattern t in U such that $M = \{ x \in PF \mid x \geq t \}$. By the definition of match set and compatibility we have,

LEMMA 14. *If M is a match set, then the patterns in M are mutually compatible.* ■

LEMMA 15. *M is a match set wrt PF iff $M = \{ x \in PF \mid x \geq glb(M)\}$, i.e, iff M is a match set of $glb(M)$ wrt PF.*

Proof The if part is trivial. Consider the *only* if part. Since M is a match set wrt PF, then there is some $t \in U$ such that $M = \{ x \in PF \mid x \geq t \}$. Since $glb(M) \geq t$, then $M = \{ x \in M \mid x \geq glb(M)\} \subseteq \{ x \in PF \mid x \geq glb(M)\} \subseteq \{ x \in PF \mid x \geq t\} = M$. ■

LEMMA 16. *If M_1 and M_2 are two match sets wrt PF, then $M_1 \cap M_2$ is also a match set wrt PF.*

Proof Since M_1 and M_2 are two match sets wrt PF, then $M_1 = \{ x \in PF \mid x \geq glb(M_1)\}$ and $M_2 = \{ x \in PF \mid x \geq glb(M_2)\}$. Therefore

$$M_1 \cap M_2$$

$$=\{x \in PF \mid x \geq glb(M_1) \text{ and } x \geq glb(M_2)\}$$

$$=\{x \in PF \mid x \geq lub(\{glb(M_1), glb(M_2)\})\},$$

which is a match set. ■