# Parallel Algorithms for Free and Associative-Commutative Unification

Gaétan Hains
Wolfson College

A thesis submitted for the degree of Doctor of Philosophy

Oxford University Computing Laboratory
Programming Research Group


1 January 1989

## Abstract

Gaétan Hains, Wolfson College.

Thesis submitted for the degree of Doctor of Philosophy.
Michaelmas term 1988.

### Parallel algorithms for free and associative-commutative unification

A survey of algorithms for free unification is given, followed by an overview of the computability and complexity of unification problems. Second-order unification is known to be undecidable, and a proof is given that the first-order problem is also undecidable under an arbitrary set of axioms.

A new systolic algorithm is introduced for term minimisation or *term compaction*. This is a general-purpose tool for systems using structure sharing. Apart from time and space savings, its use allows subterms to be tested for equality in constant time.

The use of compact terms greatly simplifies free term matching and gives rise to a linear-time algorithm with lower processing overheads than the Paterson-Wegman unification algorithm. A sublinear-time solution to the same problem is also given, assuming preloaded data. No existing algorithm for free unification has a sublinear-time implementation and this is related to the notion of a sparse P-complete problem.

The complexity of restricted associative-commutative term matching is analysed. Contrary to an earlier conjecture the problem is NP-complete if variables occur at most twice but their number is unrestricted. Parallel methods are suggested as efficient solutions for the — tractable — linear and 1-variable versions of the problem.

Results presented here should be useful in the implementation of fast symbolic manipulation systems

# Contents

# Chapter 1

# Introduction

This thesis is written as a guide to the design of fast algorithms for free term matching, free unification and associative-commutative term matching. The goal of this research was to identify tractable unification problems, to evaluate existing methods for their solution and design new ones where appropriate. The emphasis is on speed, as measured by the worst-case complexity of algorithms. Only methods with direct relevance to practical use were chosen, and are presented in the wider context of unification theory [52] [54]. Few assumptions are made about the environment in which unification is to be used. Terms are represented as graphs, the most common data structure in use now. Results presented here should be useful in the implementation of fast symbolic manipulation systems [7][6][29] [44].

Earlier work of particular relevance is that of Boyer and Moore on structure sharing [8], Robinson on unification and general proof algorithms [48][49], Paterson and Wegman on linear-time free unification [43][12], and Benanav, Kapur and Narendran on the tractability of equational term matching [4][32].

We advocate a very restricted use of parallelism with three goals in mind: a realistic complexity measure including communication delays, a reasonable use of resources and the production of simple concurrent programs. Accordingly, the usual models of parallel complexity [10][42] [19] are only used as reference. Algorithms are described as mesh-connected networks of communicating processes without any use of shared memory.

Chapter 2 introduces concepts and definitions used in unification theory, and discusses the complexity of unification problems in general. A survey of algorithms for free unification is given, followed by an analysis of unification problems in the associative-commutative theory.

Our restricted view of parallel algorithms is presented in Chapter 3, followed by a description of data structures for terms and a new systolic algorithm for their minimisation or *compaction* (i.e. the elimination of common subexpressions). This algorithm is a general-purpose tool for system using structure sharing. Apart from time and space savings, its use simplifies manipulations by allowing subterms to be tested for equality in constant time.

We then turn to free unification in Chapter 4. The use of compact terms reduces free term matching to little more than graph traversal. This gives rise to a simple and fast term matching algorithm, which is linear in the worst case and found to have lower processing overheads than the Paterson-Wegman unification algorithm. We also present a sublinear-time algorithm for the same problem, assuming preloaded data. No existing algorithm for free unification allows a sublinear-time implementation and this is related to the theoretical notion of a sparse P-complete problem.

In Chapter 5 we continue the complexity analysis of associative-commutative term matching begun in Chapter 2. This is a direct continuation of the work by Benanav, Kapur and Narendran. Contrary to a previous conjecture, the problem remains NP-complete if variables are restricted to occur twice but their number is unrestricted. We then outline parallel methods for the solution of tractable cases of the problem.

I would like to thank my supervisor Bill McColl for initiating and supporting this research. Thanks also to Tony Hoare for suggesting the study of associative-commutative unification, to Richard Miller, Quentin Miller, Bryan Todd and Wayne Luk for interesting technical discussions, and to Bill McColl and my wife Linda Bilodeau for proofreading the thesis. David Brown has been very helpful with literature searches, in particular for revealing the existence of the Downey-Sethi-Tarjan algorithm for congruence closure.

The Association of Commonwealth Universities, le Fonds FCAR (Gouvernement du Québec), the Programming Research Group and Wolfson College have contributed financially to this research. Wolfson College has provided excellent accommodation throughout my stay at Oxford. *Remerciements à mon épouse Linda Bilodeau pour son appui durant mes études.*

# Chapter 2

# Unification theory

This chapter introduces concepts and definitions concerning unification problems. Some definitions and theorems are more general than strictly necessary in further chapters. This puts the present work in context and gives the unacquainted reader access to the vast literature on unification theory. Basic results are presented, followed by more specific theorems about the complexity of free and equational unification problems.

## 2.1 Introduction

Unification is the solution of equations in logic. It is both an elementary operation in *computational logic* (or *symbolic computing*) as well as a challenging computational problem in itself. Depending on the language/calculus involved, variants of the problem can range in complexity from those solvable in logarithmic time, to NP-complete and undecidable ones. In this thesis, we will deal exclusively with first-order languages i.e. languages where variables may only take individual elements as values (and not functions or higher-order functions). A survey of first-order unification is given in [52] where Siekmann gives the following abstract definition:

> "Suppose two terms s and t are given, which by some convention denote a particular structure and let s and t contain some free variables. We say s and t are unifiable iff there are substitutions (i.e. terms replacing the free variables of s and t) such that both terms become equal in a well defined sense."

Unification appears as an elementary step in many algorithms of computational logic:

- Resolution theorem proving
- Logic programming interpreters (using resolution)

- The Knuth-Bendix completion algorithm
- Polymorphic type checking (e.g. in functional languages)
- Rewriting systems

Any gain in speed will therefore accelerate these time-consuming applications.

Unification has varied applications because of its general nature. First-order logical equations can represent a large number of combinatorial problems and their solution is related to problems such as depth-first search, manipulation of equivalence relations and directed hypergraph reachability. For the same reason, unification algorithms can be (and have been) based on corresponding combinatorial algorithms.

The unification operation is used as heavily in theorem proving applications as arithmetic operations are used in numerical computing. However, the need for fast implementations conflicts with the possible variations in size and structure of the input. When implementing arithmetic operations, one can make use of the fact that a fixed-length calculation approximates the exact solution to a known fixed degree. There is no such notion of continuity when unifying two logical expressions: an arbitrarily small change in one of the expressions can change them from 'not unifiable' to 'unifiable' or vice-versa. Unification is a pure decision problem, which allows no approximation.

In the next chapters we will study efficient algorithms and implementations as suggested by theoretical results about the complexity of unification. The central theme is the restricted use of parallelism and specific methods for important special cases of the problem. Efficiency is measured by the worst-case time complexity.

The practical use of parallelism requires consideration of communication complexity. This aspect is treated in Chapter 3.

We will assume a working knowledge of complexity theory [10] [18][42] and algorithm design [1], and deal with 'practical' issues of algorithm design such as communication costs and data structures, which are often overlooked in theoretical models of parallelism. The aim is to provide useful methods to implementers of automatic deduction systems. The next sections give definitions which are used in the rest of the thesis, basic theorems and more advanced results concerning the complexity of unification problems.

## 2.2 Definitions

### 2.2.1 Terms and substitutions

Given a countable set of variables $X$, and a countable set of function symbols (or simply 'functions') $F$, we call *terms* the finite elements of the free algebra $T(F, X)$ generated by $F$ over $X$. The set of variables occurring in a term t is

denoted by Var(t). The function symbols are partitioned as $F = \bigcup_{i=0}^{\infty} F_i$ where $F_i$ contains only functions applicable to $i$ arguments. Such functions are said to have *arity* $i$ . The elements of $F_0$ are called constants.

Variables are usually denoted by the letters u,v,w,x,y,z; constants by the letters a,b,c,d and other functions by the letters f,g,h.

Terms can be viewed as partial functions from sequences of positive integers to $F \cup X$. The domain of term t, O(t) is then called the set of *occurrences* of t [1]. For example, if t is $f(a, g(x, b))$ then $t([]) = f$, $t([2, 1]) = x$ and $\mathrm{O}(t) = \{[], [1], [2], [2, 1], [2, 2]\}$.

A *substitution* $\sigma$ is a function $X \to T(F, X)$ which is constant almost everywhere; it is usually identified with its non constant part

$$\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}.$$

The pairs $x_i \mapsto t_i$ are called *bindings*. A substitution extends uniquely to an endomorphism on terms and we will ignore the distinction between this endomorphism and the substitution itself. Thus, for a term t, $\sigma(t) = t[x_1/t_1; \ldots; x_n/t_n]$ where the replacements are simultaneous. Substitutions may be composed like any other endofunctions i.e. $\sigma\tau(x) = \sigma(\tau(x))$. Given substitutions $\sigma_1$, $\sigma_2$, we define $\sigma_1 \leq \sigma_2$, or $\sigma_1$ *generalises* $\sigma_2$, iff

$$\exists \tau \ \sigma_2 = \tau\sigma_1.$$

A substitution $\sigma$ is *idempotent* if $\sigma^2 = \sigma$.

**Proposition 1** *A substitution* $\sigma = \{x_i \mapsto t_i\}_{i=1}^{n}$ *is idempotent iff its bindings are independent:*

$$\forall (i, j)\, x_i \notin t_j(O(t_j))$$

*i.e. if none of the variables in its domain occur in the terms in its range.*

**Proof**

If all bindings are independent, then the terms $t_1, \ldots, t_n$ don't contain any occurrence of the variables $x_1, \ldots, x_n$ and so, for any term t, $\sigma(t)$ contains no occurrence of the $x_i$. Therefore $\sigma(\sigma(t)) = \sigma(t)$ and $\sigma^2 = \sigma$.

Conversely, suppose there are indices i,j such that $x_i$ occurs in $t_j$. Consider then the value of $\sigma$ on the term $x_j$:

$$\sigma(x_j) = t_j$$

$$\sigma^2(x_j) = \sigma(t_j) = t_j[x_i/t_i] \neq t_j$$

$$\sigma^2 \neq \sigma$$

[ ]

[1] The word *occurrence* is also used to denote an instance of a symbol (usu. a variable) in a given term. The exact meaning intended should be clear from the context

**Definition 1** *A substitution $\sigma$ is called* cyclic *if $\forall n$, $\sigma^n(domain(\sigma))$ contains terms in which variables of domain($\sigma$) occur. In other words, $\sigma$ is cyclic unless successive application removes the variables in its domain. The substitution is otherwise called* acyclic. [ ]

For example, $\{x \mapsto f(x,a)\}$ and $\{x \mapsto y, y \mapsto x\}$ are cyclic while $\{x \mapsto h(y), y \mapsto z\}$ is not.

Given $\sigma = \{x_i \mapsto t_i\}_{i=1}^n$, consider the directed graph $G(\sigma)$ whose nodes are the bindings of $\sigma$, with an edge from $(x_i \mapsto t_i)$ to $(x_j \mapsto t_j)$ exactly when $x_i$ occurs in $t_j$. In other words, $G(\sigma)$ contains an edge from binding $b_1$ to binding $b_2$ if, to avoid 'variable capture', $b_1$ should not be applied after $b_2$. Then $\sigma$ is cyclic iff $G(\sigma)$ contains a cycle. Also, $\sigma$ is idempotent iff the graph contains no edges.

If $\sigma$ is acyclic then $\exists(k > 0)\, \sigma^{k+1} = \sigma^k$ and then $\sigma^k$ (the transitive closure $\sigma^*$ of $\sigma$) is an idempotent substitution and will be called in this context the *idempotent closure* of $\sigma$.

Stickel [54] suggests the following method for obtaining an idempotent substitution from an acyclic substitution $\sigma = \{x_i \mapsto t_i\}_{i=1}^n$. **From now on we will require all substitutions to be acyclic.**

**Algorithm 1** *Idempotent closure.*

1. *Topologically sort $G(\sigma)$ in increasing order of bindings.*
2. *Let $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ be the resulting list after reindexing.*
3. *Let $\sigma_1 = \{x_1 \mapsto t_1\}$*
4. *For $k = 2, \ldots, n$. let $\sigma_k = \sigma_{k-1} \cup \{x_k \mapsto \sigma_{k-1}(t_k)\}$*
5. *Output $\sigma_n$.*

[ ]

We will need some auxiliary notation to prove the correctness of Stickel's algorithm. Let $\sigma_k$ be as defined in steps 3 and 4. Let $\sigma^{(k)}$ be the restriction of $\sigma$ to the first k variables in its domain: $\sigma^{(k)} = \sigma_{|x_1,\ldots,x_k}$. To prove the equivalence of two substitutions, it is sufficient to show they have equal values on their domain variables.

Let $t[x_1, \ldots, x_r]$ be a term t such that Var(t) $\cap \{x_{r+1}, \ldots, x_n\} = \emptyset$. Consider then the results of steps 1 and 2 in the algorithm:

$$\sigma = \{x_1 \mapsto t_1[\,], x_2 \mapsto t_2[x_1], \ldots, x_n \mapsto t_n[x_1, \ldots, x_{n-1}]\}$$

**Lemma 1** $\forall(i \leq k)\sigma^{(k)*}(x_i) = \sigma^*(x_i)$
**Proof** Let $i \leq k$,

$\sigma^{(k)*}(x_i) =$
$\sigma^{(k)*}\sigma^{(k)}(x_i) =$
$\sigma^{(k)*}(t_i[x_1, \ldots, x_{i-1}]) =$
$\sigma^{(i)*}(t_i[x_1, \ldots, x_{i-1}]) =$
$(\sigma_{|x_1,\ldots,x_{i-1}})^*(t_i[x_1, \ldots, x_{i-1}]) =$
$(\sigma^*)_{|x_1,\ldots,x_{i-1}}(t_i[x_1, \ldots, x_{i-1}]) =$
$\sigma^*(t_i[x_1, \ldots, x_{i-1}]) =$
$\sigma^*(\sigma(x_i)) =$
$\sigma^*(x_i).\ [\,]$

**Lemma 2** $\forall(k \leq n)\sigma_k = \sigma^{(k)*}$
**Proof** By induction on k.
*Base* $\sigma_1 = \{x_1 \mapsto t_1[\,]\} = \{x_1 \mapsto t_1[\,]\}^* = \sigma^{(1)*}$.
*Hypothesis* $\sigma_{k-1} = \sigma^{(k-1)*}$.
*Step*
$\sigma_k = \sigma_{k-1} \cup \{x_k \mapsto \sigma_{k-1}(t_k[x_1, \ldots, x_{k-1}])\} =$
$\sigma^{(k-1)*} \cup \{x_k \mapsto \sigma^{(k-1)*}(t_k[x_1, \ldots, x_{k-1}])\} =$
$\sigma^{(k-1)*} \cup \{x_k \mapsto \sigma^{(k-1)*}\sigma^{(k)}(x_k)\} =$
$\sigma^{(k-1)*} \cup \{x_k \mapsto \sigma^{(k)*}(x_k)\} =$
$\sigma^{(k)*}.\ [\,]$

**Proposition 2** *Given a substitution $\sigma$ (acyclic), Stickel's algorithm always terminates with $\sigma_n = \sigma^*$.*
**Proof**
Termination follows from the fact that $\sigma$ is acyclic and so $G(\sigma)$ can be topologically sorted.

Partial correctness follows from the last lemma:

$$\sigma_n = \sigma^{(n)*} = \sigma^*.[\,]$$

In some situations, it is simpler to leave a substitution in its non-idempotent form. It should then be understood that the idempotent closure could be computed using the above algorithm. The time required for this transformation depends on the data structures used to represent substitutions, but can be made linear in the size of a representation for $\sigma$.

### 2.2.2 Equational algebras, unifiers

An *axiom* is an unordered pair of terms written $t = t'$. Given a finite set of axioms $E$, the *equational theory* presented by $E$ is the set of equations formed by the finest congruence on $T(F, X)$ closed under substitutions and containing $E$. This congruence is also called $E$-equality, or $E$-equivalence, and is written $=_E$. Thus, $t_1 =_E t_2$ iff one is derivable from the other by a finite proof using the axioms of $E$. For example, if $E$ contains the axiom $f(x, f(y, z)) = f(f(x, y), z)$ then we have

$$f(a, f(b, f(c, d))) =_E f(f(f(a, b), c), d).$$

In the case of an empty theory, $=_\emptyset$ is just the (syntactic) equality on terms.

**Definition 2** *For substitutions* $\sigma_1, \sigma_2$*,*

- $\sigma_1 =_E \sigma_2$ *iff* $\forall(x : X)\, \sigma_1(x) =_E \sigma_2(x)$
- $\sigma_1 \leq_E \sigma_2$ *iff* $(\exists\tau)\, \tau\sigma_1 =_E \sigma_2$.

A *E-unifier* (or simply *unifier*) for a set of terms $S = \{t_1, \ldots, t_n\}$ is a substitution $\sigma$ such that $\sigma(t_1) =_E \sigma(t_2) =_E \ldots =_E \sigma(t_n)$. The set $S$ is said to be *E-unifiable* (or simply *unifiable*) if there exists such a substitution. It is clear that S is unifiable exactly when $\{f(t_1, \ldots, t_n),\ f(x, \ldots, x)\}$ — for $x$ a fresh variable — is unifiable. For this reason, unification problems can always be formulated as pairs of terms.

Let now $U_E(S)$ be the set of all unifiers for S. A subset of substitutions $\Sigma \subseteq U_E(S)$ is a set of *most general unifiers* for S if:

1. $\forall(\sigma : U_E(S))\, \exists(\sigma' : \Sigma)\, \sigma' \leq_E \sigma$ (completeness)
2. $\forall(\sigma, \sigma' : \Sigma)\, \neg(\sigma \leq_E \sigma')$ (minimality, i.e. generality)
3. $\forall(\sigma : \Sigma)\, \mathrm{Dom}(\sigma) \subseteq \mathrm{Var}(S)$ (simplicity)

We then write $\Sigma = \mu U_E(S)$ and call the elements of $\Sigma$ *most general unifiers* of $S$. The first two axioms make $\Sigma$ a minimal basis of unifiers for S. The last axiom is not necessary in the theories studied in this thesis ($\emptyset$ and AC). The usual application of *term matching* is term rewriting, where only one *idempotent* unifier is needed; an idempotent unifier is not necessarily minimal (most general). The most general unifiers are used as output from (free or associative-commutative) unification as part of an exhaustive search process.

**Definition 3** *A theory E is called:*

- unitary (of type 1) *if* $\forall S\, |\mu U_E(S)| \leq 1$*;*
- finitary (of type $\omega$) *if whenever* $U_E(S) \neq \emptyset$*,* $\mu U_E(S)$ *is finite;*
- infinitary (of type $\infty$) *if whenever* $U_E(S) \neq \emptyset$*,* $\mu U_E(S)$ *always exists and there exists a set of terms S for which* $\mu U_E(S)$ *is infinite;*
- nullary (of type 0) *if there exists a set of terms S for which* $U_E(S) \neq \emptyset$ *but* $\mu U_E(S)$ *does not exist.*

For example, the empty theory is unitary [43], the associative-commutative theory is finitary, the associative theory is infinitary and the associative-idempotent theory is nullary [3], [50].

We now formally define unification problems as decision problems.

**Definition 4** *UNIF-E, unification in the theory E.*
**Input** *Two terms $s, t$.*
**Output** *"YES" if $\exists\sigma\ \sigma(s) =_E \sigma(t)$, "NO" otherwise.*

**Definition 5** *TMATCH-E, term matching in the theory E.*
**Input** *Two terms $s, t$.*
**Output** *"YES" if $\exists\sigma\ s =_E \sigma(t)$, "NO" otherwise.*

**Definition 6** *EQUAL-E, equality in the theory E.*
**Input** *Two terms $s, t$.*
**Output** *"YES" if $s =_E t$, "NO" otherwise.*

For example:

**Definition 7** *UNIF-A, associative unification.*
**Input** *Two terms $s, t$ containing some associative functions $F_A$.*
**Output** *"YES" if $\{s, t\}$ is A-unifiable, "NO" otherwise:*

$$A = \{f(x, f(y, z)) = f(f(x, y), z) | f \in F_A\}$$

Similarly one defines UNIF-C, UNIF-I and UNIF-D where the axioms are respectively commutativity, idempotence and distributivity only. Problems UNIF-$E$ where the theory $E$ is non-empty are referred to as *equational unification* as opposed to free unification which is simply written UNIF. One important equational unification problem is UNIF-AC where some functions are both associative and commutative. It occurs frequently in proof systems which manipulate arithmetic expressions. Free unification is used by Prolog interpreters and type checkers for polymorphic type structures. Term matching is used in term rewriting, to select applicable rewrite rules.

We now consider graphs as an abstract data structure for terms and substitutions.

### 2.2.3 Graph representations and complexity

It has long been acknowledged that manipulating syntactic objects is best done using some pointer representation of their structure. For instance if terms were to be encoded as strings of symbols, the operations of insertion/deletion of subterms would require traversal of the complete string. If however they are represented as trees ('syntax trees'), then insertion/deletion can be done dynamically in time independent of the size of the terms.

One problem with tree representations is that they cannot take advantage of regularities in the structure of terms. For example, if an algorithm is searching the term $f(g(a, h(c), y), g(a, h(c), y))$ for an occurrence of the variable $z$, it will have to traverse the whole tree before completion. If on the other hand, the two arguments of $f$ were merged into a single structure, the algorithm could terminate

twice as fast. This simple observation is the basis for the powerful method of *structure sharing* which was proposed for use in resolution proof systems by Boyer and Moore [8]. Structure sharing was introduced into unification methods in [43] where Paterson and Wegman also gave the first linear-time algorithm for UNIF. The following definitions follow those of Paterson and Wegman; but are not completely standardised in the literature.

A term is represented by a rooted, connected, ordered, labeled, directed acyclic graph (henceforth: *dag*) as follows. Each node corresponds to (possibly multiple occurrences of) a symbol in the term and is labeled with this symbol. Constants and variables are represented by nodes of outdegree 0. Terms whose root symbol is a function of arity n are represented by one root node for the function with outgoing edges pointing to subgraphs for the respective subterms. Outgoing edges from a node are labeled with the integers $1, \ldots, k$, where $k$ is the arity of that node's function symbol, and are ordered from left to right in graphical representations of the dag.

Repeated subterms can be represented as shared subgraphs and for this reason a given term may have multiple dag representations. A dag with no shared subgraphs is simply an oriented version of a syntax tree and will be called a *tree*. If the only shared subgraphs are the different occurrences of variables, then the dag is called a *simple dag*. For example, the term

$$f(g(a, h(c), y), g(a, h(c), y))$$

can be represented as a simple dag on 10 nodes, a tree on 11 nodes, or as a *reentrant* dag on 6 nodes where the node for g has indegree 2 (see Figure 2.1).

**Definition 8** *A term dag is* compact *if for all pairs of distinct nodes in it, the subterms rooted at the nodes are different.*

In the above example, the reentrant dag on 6 nodes is compact.

**Definition 9** *A term is* linear *if no variable occurs twice in it i.e. no variable node is shared or repeated.*

The tree representation of a linear term is therefore the same as its representation as a simple dag.

Compactness can make an exponential difference in both size of unifiers and time of unification as in the following example:

$$\text{UNIF}\{g(f(x_1, x_1), \ldots, f(x_{n-1}, x_{n-1})); \; g(x_2, \ldots, x_n)\}$$

An idempotent unifier for these two terms assigns to $x_n$ a term which, if represented as a tree has size $2^n$, but which can be represented by a compact dag of size linear in n.

The input to a unification problem is usually presented as a single dag (possibly not connected) with two nodes of indegree 0 where the terms to be unified

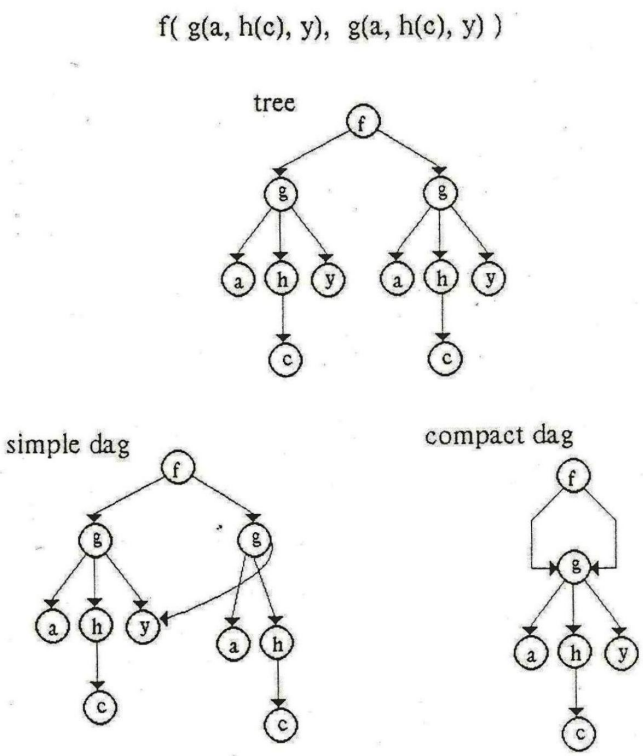


Figure 2.1: graph representations

are rooted. *The time complexity of a problem is given as a function of the size of the dags it is given as input* which can vary by an exponential factor. Important parameters (other than size) in measuring the efficiency of unification algorithms are the dag's depth, its compactness, the number of subterms which share variables and the number of variables.

We will return to the issue of structure sharing in Chapter 3.

## 2.3 Basic theorems

This section discusses unification decision problems and the computation of unifiers.

### 2.3.1 Undecidability

In their most general instances, unification problems are undecidable: no algorithm can solve them effectively for every input. This was first recognised in the case of higher-order languages. In the first-order language $T(F, X)$ defined earlier, terms represent individual elements of a model $\mathcal{M}$ for the language. In second-order languages, function symbols and function variables representing functions $\mathcal{M} \to \mathcal{M}$ are allowed. In third-order logic, there are terms representing functions of type $\mathcal{M} \to (\mathcal{M} \to \mathcal{M})$ or $(\mathcal{M} \to \mathcal{M}) \to \mathcal{M}$, and so on.

Huet [26] cites an early paper by W.E.Gould (1966) where it is shown that higher-order languages can be nullary, certain pairs of terms having an infinity of unifiers, none of them most general. He then proves a stronger result.

**Theorem 1** *(Huet 1973)*
*The unification problem for third-order logic is undecidable.*

Second-order unification is just as difficult in the general case.

**Theorem 2** *(Goldfarb 1981 [20])*
*The unification problem for second-order logic is undecidable.*
**Outline of proof**
The proof is an effective reduction of a known undecidable problem to second-order unification.

Given an instance of Hilbert's tenth problem — a Diophantine equation $H$ to be solved in the positive integers — a pair of terms can be constructed which are unifiable iff $H$ is solvable. The terms encode addition and multiplication of integers into the indices of function variables.[ ]

We will now show that even with first-order unification, if the equational theory is unrestricted, the problem becomes undecidable. An algorithm for the general problem may not terminate on certain inputs.

**Theorem 3** *General first-order equational unification is undecidable.*
**Proof**
There exists a theory $U$ such that the Halting problem effectively reduces to UNIF-$U$. We will construct an equational theory $U$ in which equality simulates the possible transitions of a universal Turing machine. Given an instance of the halting problem (a Turing machine with an input on its tape), there is a corresponding instance of UNIF-$U$ which is true iff the Turing machine halts.

The Turing machine execution is expressed as a sequence of *instantaneous descriptions* (IDs) which will be encoded as *ID-terms*. An ID-term is a triple of the form ID(state, tape, transitions). To simulate Turing machine tapes and finite-state transitions, we will use sequences and set structures respectively.

**Axioms for sequences:** Let the following symbols be defined

$$\text{seq} \in F_2; \;\; \text{single} \in F_1; \;\; \text{empty} \in F_0.$$

Sequences are simulated by the following axioms.

- $\text{seq}(x, \text{empty}) =_U \text{single}(x)$
- $\text{seq}(\text{empty}, x) =_U \text{single}(x)$
- $\text{seq}(x, \text{seq}(y, z)) =_U \text{seq}(\text{seq}(x, y), z)$
- $\text{seq}(x, \text{single}(y)) =_U \text{seq}(x, y)$
- $\text{seq}(\text{single}(x), y) =_U \text{seq}(x, y)$

**Axioms for sets:** Let $\text{set} \in F_2$. Sets are simulated by the following axioms.

- $\text{set}(x, \text{set}(y, z)) =_U \text{set}(\text{set}(x, y), z)$
- $\text{set}(x, y) =_U \text{set}(y, x)$
- $\text{set}(x, x, y) =_U \text{set}(x, y)$

Without loss of generality, the sets we consider are always of size at least 2. There is no need for axioms representing singletons or empty sets: Turing machines with fewer than two transitions can be ignored.

**Syntax for ID terms:** An ID-term is of the form ID($state, tape, transitions$) with the following syntax.

- $state ::= S_{START} | S_2 | \ldots | S_{n-1} | S_{STOP}$
- $tape ::= \text{tape}(leftseq, presentsymbol, rightseq)$
- $leftseq, rightseq ::= \text{sequence}[bit]$
- $presentsymbol ::= bit$

- $sequence[T] ::= \text{empty}|\text{single}(T)|\text{seq}(T, sequence[T])$
- $bit ::= 0|1$
- $transitions ::= \text{set}(trans, trans)|\text{set}(trans, transitions)$
- $trans ::= \text{trans}(oldstate, oldbit, move, newbit, newstate)$
- $oldstate, newstate ::= state$
- $oldbit, newbit ::= bit$
- $move ::= L|R$

If for example the Turing machine has transitions $t_1, t_2, t_3, t_4$ and its input is $[1, 0, 0, 1, 0]$, the initial ID will be encoded as the following ID-term:

$$\text{ID}(S_{START},\ \text{tape}(\text{empty}, 1, \text{seq}(0, \text{seq}(0, \text{seq}(1, 0)))),\ \text{set}(\widehat{t_1}, \text{set}(\widehat{t_2}, \text{set}(\widehat{t_3}, \widehat{t_4})))),$$

where $\widehat{t_i}$ is the syntactic representation of $t_i$.

The simulation is nondeterministic and can even backtrack to previous IDs: any matching axiom can be applied to an ID-term and the axiom can be applied in either direction. However, from the point of view of computability, determinism is irrelevant. The nondeterministic-backtracking machine has the same set of reachable IDs as deterministic restrictions of it.

**Axioms for transitions.**

The following axioms define the action of transitions on the machine ID.

- Move left when only one symbol is to the left of the head.
  $\text{ID}(x_S; \text{tape}(\text{single}(x^L_{bit}), x_{bit}, y^R); \text{set}(\text{trans}(x_S, x_{bit}, L, x'_{bit}, x'_S), z)) =_U$
  $\text{ID}(x'_S; \text{tape}(\text{empty}, x^L_{bit}, \text{seq}(x'_{bit}, y^R)); \text{set}(\text{trans}(x_S, x_{bit}, L, x'_{bit}, x'_S), z))$
- Move left with two or more symbols to the left of the head.
  $\text{ID}(x_S; \text{tape}(\text{seq}(x^L_{bit}, y^L), x_{bit}, y^R); \text{set}(\text{trans}(x_S, x_{bit}, L, x'_{bit}, x'_S), z)) =_U$
  $\text{ID}(x'_S; \text{tape}(y^L, x^L_{bit}, \text{seq}(x'_{bit}, y^R)); \text{set}(\text{trans}(x_S, x_{bit}, L, x'_{bit}, x'_S), z))$
- Move right when only one symbol is to the right of the head.
  $\text{ID}(x_S; \text{tape}(y^L, x_{bit}, \text{single}(x^R_{bit})); \text{set}(\text{trans}(x_S, x_{bit}, R, x'_{bit}, x'_S), z)) =_U$
  $\text{ID}(x'_S; \text{tape}(\text{seq}(x'_{bit}, y^L), x^R_{bit}, \text{empty}); \text{set}(\text{trans}(x_S, x_{bit}, R, x'_{bit}, x'_S), z))$
- Move right with two or more symbols to the right of the head.
  $\text{ID}(x_S; \text{tape}(y^L, x_{bit}, \text{seq}(x^R_{bit}, y^R)); \text{set}(\text{trans}(x_S, x_{bit}, R, x'_{bit}, x'_S), z)) =_U$
  $\text{ID}(x'_S; \text{tape}(\text{seq}(x'_{bit}, y^L), x^R_{bit}, y^R); \text{set}(\text{trans}(x_S, x_{bit}, R, x'_{bit}, x'_S), z))$

For an instance of the halting problem:

- Turing machine with transitions $t_1, \ldots, t_n$,
- Input sequence $[b_1, \ldots, b_k] \in \{0,1\}^*$,

the solution of UNIF-$U$ for the following pair of ID-terms is "YES" if and only if the Turing machine can halt on the given input:

ID($S_{START}$; tape(empty, $b_1$, seq($b_2$, seq($b_3 \ldots b_k$))); set($\widehat{t_1}$, set($\widehat{t_2} \ldots \widehat{t_n}$)))

ID($S_{STOP}$; $u$; $v$)

[]

### 2.3.2 NP-hardness, NP-completeness

The undecidability of general equational unification has a simple intuitive cause: axioms contain variables. To see how this is a decisive factor, let $U$ be the equational theory defined in the proof of Theorem 3 and consider an algorithm $\mathcal{A}$ which would correctly decide UNIF-$U$ whenever it would terminate. Then $\mathcal{A}$ must transform terms in ways consistent with $=_U$ and stop whenever it obtains a given target term. Each transformation involves some axiom, but because axioms contain variables they can match with an infinity of different terms; the finite set of axioms $U$ can in general create infinite sequences of distinct $U$-equivalent terms.

One obvious solution to this problem is to allow only 'constant' axioms (called *defining relations* in universal algebra). The result is called a *finitely presented algebra* or FPA. FPAs have less expressive power than general equational algebras. For example properties such as idempotence, commutativity and associativity cannot be finitely presented because they would require one defining relation for every term (or pair or triple of terms respectively) to which they apply. However, unification in an arbitrary FPA is decidable though probably not in polynomial time.

**Theorem 4** *(Kozen 1977 [35])*
*Unification in FPAs is NP-complete.*

To avoid undecidability and still retain the simplicity of equational algebras, one must restrict the axioms allowed. Nevertheless interesting cases are of high complexity.

**Theorem 5** *[18] UNIF-C is NP-complete.*

It is also known that TMATCH-AC is NP-complete and so UNIF-AC is NP-hard. But the question (UNIF-AC $\in$ NP ?) remains unsolved.

The addition or removal of axioms may or may not change the complexity of a problem.

**Theorem 6** *(Kapur and Narendran 1986[32])*
*TMATCH-AI, TMATCH-CI, TMATCH-ACI and the corresponding unification problems are all NP-hard.*

All the above mentioned problems thus seem to require specialised methods to cope with combinatorial explosion, if they are to be solved efficiently.

### 2.3.3 P-completeness

The suspected intractability of equational unification problems has a simple intuitive cause: the axioms. To see why this is so, consider the general case where terms $t_1, t_2$ are given to unify and $t_1$ contains the function occurrences $\{f_1, \ldots, f_n\}$ (with possible repetitions). For any subset $\{f_{i_1}, \ldots f_{i_k}\}$ a different term can be obtained from $t_1$ by applying the relevant axiom(s) to each $f_{i_j}$. Any one of these $2^n$ terms could be equal to $t_2$ and so any unification algorithm could be forced to search a number of substitutions exponential in the size of $t_1$.

Again, there is an obvious 'solution' to this: remove the axioms and deal only with UNIF, free unification. The result of such a drastic simplification is of course much less expressive than equational unification but still surprisingly powerful as shown by the following propositions.

Given a graph (or Boolean matrix), the associated reachability problem (or transitive closure calculation) consists of deciding, for any pair of vertices whether they are connected by a path in the graph.

**Proposition 3** *The reachability problem in an undirected graph reduces to UNIF.*
**Proof**
Consider a graph with vertices $\{x_1, \ldots, x_n\}$ and edges $\{(x_{i_1}, x_{j_1}), \ldots, (x_{i_k}, x_{j_k})\}$ and identify the vertices with variables X. Construct then the following terms of $T(F_{k+2} \cup F_0, X)$:

$$t_1 = f(x_{i_1}, \ldots, x_{i_k}, a, x_b)$$
$$t_2 = f(x_{j_1}, \ldots, x_{j_k}, x_a, b).$$

Then $t_1, t_2$ are unifiable iff vertices $x_a$ and $x_b$ are in different connected components of the graph.[]

Notice that the use of a function from $F_{k+2}$ in Proposition 3 is not essential. The same reduction can be done using a tree of binary functions so that the arity of $f$ does not grow with the number of variables. The size of the terms is linear in the size of the input graph.

The main computational advantage of free unification is its tractability. Every step in the execution of a Prolog interpreter solves an instance of it. To unify two terms given in dag form it is sufficient to verify that their structures are compatible, subject to the constraint that two subterms unified with the same variable must be unified themselves. In fact it is straightforward to design an algorithm for doing this in quadratic time or less. A worst-case linear-time

algorithm appeared in [43] which closed the issue of sequential complexity of UNIF.

In the following years there was great interest in the possibility of parallel methods for Prolog requiring parallel implementations of UNIF. The question then arose of whether there existed very fast parallel algorithms for it.

The standard theoretical framework in which such a question can be treated is that of polylogarithmic-time parallel algorithms on abstract shared-memory machines [19]. The class of problems included in P (solvable in polynomial time) and solvable by polylog-time algorithms is called NC (or $NC^+$ depending on the exact model). NC is related to the class P in a way analogous to the P=$^?$NP question. There is a class of problems in P called *logtime-complete for P* or simply *P-complete*, which are not solvable in NC unless NC=P. In the present state of research, NC=P seems very unlikely [37] [42] and so P-completeness is strong evidence that a problem is not solvable by a shared-memory computer in time $O(\log n)^k$ for any positive k. Such problems are probably 'inherently sequential'; they are not amenable to parallel decomposition in a way which allows exponential gains in speed.

A known P-complete problem is the monotone circuit value problem (MCV) [42]. Its definition is as follows. Given a combinational (acyclic) circuit built from two-input AND/OR gates, and an assignment of bits to its inputs, compute its output.

**Proposition 4** *(Dwork et al. 1984 [14])*
*MCV is reducible to UNIF.*

P-completeness of a problem is usually shown by giving a parallel log-time reduction from a known P-complete problem. The reduction in the proof of Proposition 4 was computable in parallel log-time and therefore,

**Theorem 7** *(Dwork et al. [14])*
*UNIF is P-complete.*

This was also proved independently by Yasuura [62][63] using the reachability problem in directed hypergraphs.

Unlike general UNIF, TMATCH is solvable in NC, [15][57][58]. The existence or absence of polylog-time algorithms shows the feasibility (or difficulty) of large-scale parallel decomposition. However, the shared-memory computer model on which they are based makes their stated complexity unrealistic because of hidden communication costs. Practical algorithms for TMATCH can take advantage of the problem's natural decomposition but must also avoid communication problems when implemented on a distributed memory architecture. We will return to this issue in Chapter 3.

### 2.3.4 Algorithms for UNIF.

This subsection describes the most important published algorithms for free unification. All of them take advantage of the uniqueness of a most general unifier in UNIF.

**Proposition 5** *[48] The empty theory is unitary.*

The earliest definition of a unification algorithm came with Robinson's introduction of resolution theorem proving, using unification as its basic step [48]. His algorithm is still used as a standard in most Prolog interpreters but without the costly test for acyclicity (or *occurs check*) at step 5. One notable exception is Colmerauer's Prolog-II where free unification is generalised to cyclic terms [7].

In the statement of Robinson's algorithm, the following definition is used. The *disagreement set* of a set of terms $A$ contains the subterm(s) rooted at the leftmost symbol occurrence where terms in $A$ differ. For example the disagreement set of

$$\{g(h(x),x),\ g(h(x),g(a,b)),\ g(h(x),b)\}$$

is $\{x,\ g(a,b),\ b\}$. In the version given below, sets of terms are always of size two or less, and disagreement sets are always of size two.

**Algorithm 2** *(Robinson)*
**Input** *Terms* $t_1, t_2$.
**Output** $\sigma_A$*, the most general unifier of* $t_1, t_2$ *if it exists. 'NO' otherwise.*
**Method**

1. $\sigma_0 := \emptyset;\ k := 0;\ A := \{t_1, t_2\}$.
2. *If* $|\sigma_k(A)| = 1$ *then*
   $\sigma_A := \sigma_k;$ *Output*$(\sigma_A);$ *Stop*
3. $B_k :=$ *disagreement set of* $\sigma_k(A)$.
4. $U_k, V_k :=$ *the two distinct terms in* $B_k$ *where* $V_k \in X$ *if either one is a variable.*
5. *If* $V_k \in X$ *and does not occur in* $U_k$ *then*
   $\sigma_{k+1} := \{V_k \mapsto U_k\} \circ \sigma_k;\ k := k+1;$ *Goto*2
   *Else Output('NO'); Stop* [ ]

Robinson's algorithm applies a divide-and-conquer strategy to the propagation of unification constraints. Because it assumes a simple dag, it will perform the same operations multiple times for a shared subterm. For a highly reentrant term dag its performance degrades to $\Omega(2^n)$ time, where n is the size of the graph.

Linear time is attained by a non-redundant search of the dag. The Paterson-Wegman algorithm is based on a representation of subterms by their root vertices and unification constraints by equivalence classes of vertices.

**Definition 10** *An equivalence relation $\mathcal{E}$ on the vertices of a term dag $G$ is called* valid *if:*

1. $\forall n.\forall(f : F_n).\ f(t_1, \ldots, t_n) \equiv_{\mathcal{E}} f(s_1, \ldots, s_n) \ \Rightarrow\ \bigwedge_{i=1}^{n}(t_i \equiv_{\mathcal{E}} s_i)$
2. $\forall(f_1, f_2 : F)\ f_1 \equiv_{\mathcal{E}} f_2 \ \Rightarrow\ f_1 = f_2.$
3. *$G/\mathcal{E}$ is acyclic.*

The following lemma is the basis of Paterson and Wegman's algorithm and is implicit in many others [28][48][49] [57][64].

**Lemma 3** *(Paterson and Wegman 1978 [43])*
*Given a dag with roots $u, v$ then terms rooted at $u$ and $v$ are unifiable if and only if there exists a valid equivalence $\mathcal{E}$ such that $u \equiv_{\mathcal{E}} v$. In the affirmative, there is a unique finest $\mathcal{E}$ which represents the most general unifier of $u, v$.* [ ]

The resulting algorithm is described in Chapter 4.

A (directed) hypergraph of order two consists of a set of nodes $V$, a set of directed edges from $V \times V$ and a set of *hyperedges*. A hyperedge consists of a pair of source nodes and a target node : pairs$(V) \times V$. The usual definition of reachability (or closure) is generalised as follows in hypergraphs of order two. A node $v$ is said to be reachable from a set $S \subseteq V$ either if there is an edge $(s, v)$ such that $s \in S$ or if there is an hyperedge $(\{s, t\}, v)$ such that $s, t \in S$.

Yasuura [63] gave a method where unification constraints are represented as nodes in a hypergraph of order two. The initial constraint $u \equiv v$ is at the root of the hypergraph, from which a reachability calculation determines the unification bindings. Although conceptually equivalent to Paterson and Wegman's, this method allows parallel decomposition of the problem. Just like graph reachability is solvable in NC, Yasuura defines a shared-memory parallel algorithm for the hypergraph problem. In the worst case the algorithm is no better than sequential execution because of the P-completeness of UNIF (and hence, of hypergraph reachability). However, the sequential behaviour is only caused by the handling of constraints on variables. In the event of terms containing relatively few variables, parallelism will be useful. This is expressed by the algorithm's worst-case complexity $O((\log n)^2 + n' \log n')$; where $n'$ is the number of distinct variables occurring in the terms.

Another useful parameter is the number of *repeated* variables i.e. variables which occur in both terms or twice in one of them. If the number of repeated variables is constant (in particular, if the terms are linear) then UNIF has been shown to be solvable in $O(\log n)^2$ time [31].

Another algorithm for UNIF which allows a parallel implementation is that of Martelli and Montanari [39]. It represents unification as equation solving by transformation of a system of so-called multiequations. The method is first

presented as a nondeterministic rewriting algorithm which could be given a distributed implementation: rewrite rules can be applied in parallel. The algorithm is then refined into an efficient sequential program of worst-case complexity $O(n' \log n')$, where again $n'$ is the number of variables involved.

As we will see later, Yasuura's algorithm is limited by its communication requirements which makes it useless as a practical use of parallelism. For the same reason, we will not consider a parallel implementation of Martelli and Montanari's algorithm.

## 2.4 More about complexity.

The previous section has given us reason to believe that many unification problems are intractable in their most general form. It does not follow that all interesting *instances* of these problems are computationally hard. As a trivial example, the following is an easily decidable instance of TMATCH-AC:

$$f[x_1, x_2, x_3] \text{ vs } f[a, b, c]$$

but the general problem is NP-complete. Unfortunately, the mere realisation of this fact is of little help: an algorithm must deal with a whole class of problems, some of which may be arbitrarily easier than others. Moreover the class of inputs for which an algorithm is efficient must either occur frequently in a given application or be relatively easy to detect mechanically.

When concerned with worst-case performance of algorithms, it is therefore useful to isolate *special cases* of the problem which are both easy to detect and of lower complexity than the general case. This section studies the complexity of special cases of some important unification problems and in particular the problem TMATCH-AC. As is often the case, there are as many negative results as positive ones. The negative results are very useful though, and one of them was unexpected (Theorem 10). They prevent us from investing design energy into useless heuristics and guide the search for more appropriate restrictions of the problem.

Axioms which occur often in theorem-proving applications — and are therefore useful to incorporate into unification algorithms — are associativity (A), commutativity (C) and idempotence (I). Problems suffixed with a combination of these are to be understood as defined over the intersection (*not* union) of the corresponding theories. For example TMATCH-AC refers to the theory where some functions are both associative and commutative, other functions being free.

Some common combinations of { A,C,I } are:

- A without C or I (as for list construction operators or function composition)
- AC without I (as for arithmetic operators)
- ACI (as for Boolean conjunction and disjunction or set operators)

and we will now restrict attention to those theories.

To simplify the presentation, throughout this section terms will be treated as simple dags. Clearly this does not affect the validity of lower bounds on complexity: if a problem is suspected of requiring time $T(n)$ in the size of simple dags, then it would require at least as much in the size of compact dags. In later chapters we will also consider methods for dealing with compact dags without loss of efficiency. Most authors in the area of unification theory make the tacit assumption that algorithms for simple dags can always be adapted in this way.

Figure 2.4.1.

| Theory | EQUAL | TMATCH | UNIF |
|---|---|---|---|
| A | in P | NP-complete | NP-hard |
| AC | in P | NP-complete | NP-hard |
| ACI | in P | NP-hard | NP-hard |

### 2.4.1 Equality, term matching and unification

Two natural restrictions of unification are term matching and equality testing. Any instance of unification where one term is free of variables may be solved as a term matching problem:

$$(t(O(t)) \cap X = \emptyset) \Rightarrow [(\exists\sigma\ \sigma(s) =_E \sigma(t)) \Leftrightarrow (\exists\sigma\ \sigma(s) =_E t)]$$

Similarly, unification without variables is equivalent to equality testing.

We will now consider the tractability of EQUAL-$E$, TMATCH-$E$ and UNIF-$E$ when $E$ ranges over { A, AC, ACI}. The results presented or cited in [4][32] concern the complexity of all nine problems except EQUAL-A. The exact status (membership in NP or not) of the three UNIF-$E$ problems is unknown. The next proposition completes Figure 2.4.1.

**Proposition 6** *EQUAL-A is computable in linear time.*
**Proof**
Let the input terms be $s, t$ and $F_A \subseteq F_2$ the set of associative functions. Without loss of generality we will 'flatten' nested occurrences of any given $f \in F_A$ and write it as an operator on lists. For example write $f[a, g(a), x, b]$ for $f(f(a, g(a)), f(x, b))$. This transformation requires only linear time.

Suppose now that

$$s = g[s_1, \ldots, s_m];\ s_i[] \neq g$$

and

$$t = g[t_1, \ldots, t_n];\ t_i[] \neq g$$

where $g \in F_A$.

Then for $s =_A t$ to hold it is sufficient that

$$m = n \wedge \forall(i \leq n)\ s_i =_A t_i$$

This condition is also necessary because equality, unlike matching, may only occur through applications of the associative axioms, which don't affect the (flattened) list structure of $s$ or $t$.

Testing equality therefore consists in two passes through the terms. One to recursively flatten every associative function and another to test for syntactic equality, the whole requiring linear time [ ]

AC-equality is also solvable in linear time.

**Proposition 7** *EQUAL-AC is computable in linear time by a systolic algorithm.*
**Proof**
Given two dags for AC-terms $s, t$, choose a function symbol $f$ different from the root symbols of $s$ and $t$, construct the dag $f(s,t)$ and apply dag compaction (DC: Chapter 3) to it. Then $s$ and $t$ are equal if and only if the output has merged the corresponding subgraphs in $f(s,t)$. []

Equality is also tractable for I, C, AI and ACI but we will not expand on this here.

The results in Figure 2.4.1 suggest, in all three theories, the implementation of a special equality testing procedure and the identification of tractable special cases of term matching or unification. TMATCH-AC is treated in the next subsection as well as Chapter 5.

We will now deal exclusively with the empty and AC theories.

### 2.4.2 Restrictions of UNIF-AC

The P-completeness of free unification has prompted research in the determination of special cases of UNIF solvable in NC, such as free term matching, monadic unification (unification over $T(F_1, X)$) and unification of linear terms [2] [14] [15] [30] [41] [57] [62] [63] [31].

Similarly there are simple syntactic properties which make AC- unification or term matching tractable. One such property is linearity.

**Theorem 8** *(Benanav, Kapur and Narendran 1985[4])*
*TMATCH-AC (decision problem) is computable in time* $O(|s|\,|t|^3)$ *for linear terms of sizes* $|s|$ *and* $|t|$.

This result is based on the following simplification: when subterms have disjoint sets of variables, the union of matching substitutions for the subterms is a matching substitution for the complete term. Distinct subterms of linear terms always have disjoint sets of variables.

The proof of Benanav, Kapur and Narendran can be extended to apply to unification, under the condition that variables are not shared by the terms being unified. That part of the proof ($m' = 0$) which corresponds to theirs is only sketched here. Refer to [4](theorem 2) for full details.

In what follows, an associative function applied to a list of the form $f[t_i, \ldots, t_{i+j}]$, $j = 0$, will be understood as a convenient notation for $t_i$ itself.

**Theorem 9** *The decision problem UNIF-AC is computable in polynomial time for linear terms* $s, t$ *with disjoint sets of variables; i.e. when h(s,t) is linear.*
**Outline of proof**
Let $t = f[t_1, \ldots, t_n, x_1, \ldots, x_{n'}]$ and $s = f[s_1, \ldots, s_m, y_1, \ldots, y_{m'}]$, where $f \in F_{AC}$, $x_i, y_j \in X$ and $t_i, s_j$ are terms with root symbols in $F - \{f\}$. Then by symmetry, it is sufficient to consider the following cases.

- $n' > 0$ and $m' > 0$
  - $0 < m' \leq n$ and $0 < n' \leq m$
    In this case, the terms are unifiable by

    $$\{y_1 \mapsto t_1, \ldots, y_{m'-1} \mapsto t_{m'-1}, y_{m'} \mapsto f[t_{m'}, \ldots, t_n]\} \cup$$

    $$\{x_1 \mapsto s_1, \ldots, x_{n'-1} \mapsto s_{n'-1}, x_{n'} \mapsto f[s_{n'}, \ldots, s_m]\}$$

  - $0 < n < m'$ and $0 < n' \leq m$
    Here $s, t$ are unifiable as follows.

    $$\{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n\} \cup$$

    $$\{x_1 \mapsto s_1, \ldots, x_{n'-1} \mapsto s_{n'-1}, x_{n'} \mapsto f[s_{n'}, \ldots, s_m, y_{n+1}, \ldots, y_{m'}]\}$$

  - $0 \leq n < m'$, $0 < m < n'$ and $m' - n \leq n' - m$
    Again $s, t$ are unifiable:
    $\{y_1 \mapsto t_1, \ldots, y_n \mapsto t_n, x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\} \cup$
    $\{y_{n+1} \mapsto x_{m+1}, \ldots, y_{m'-1} \mapsto x_{m'+m-(n+1)}\} \cup$
    $\{y_{m'} \mapsto f[x_{m'+m-n}, \ldots, x_{n'}]\}$

- $m' = 0$
  Here, $t = f[t_1, \ldots, t_n, x_1, \ldots, x_{n'}]$ and $s = f[s_1, \ldots, s_m]$ are not unifiable if $m < n + n'$ so assume $m \geq n + n'$. The terms are unifiable if and only if, for i:(1..n) there exists distinct $j_i$ and $\sigma_i$ such that $\sigma_i(t_i) = \sigma_i(s_{j_i})$. If this is the case, the unifier is

  $$\bigcup_{i=1}^{n} \sigma_i \cup \{\text{any injective substitution of the remaining } s_j \text{ for the } x_i\}$$

  To verify the existence of such $\sigma_i$ in the first place, recursively test every pair $(t_i, s_j)$ for unifiability and construct the bipartite graph $G$ on $(1..n) \times (1..m)$ where an edge indicates a positive result. Then check the existence of a (maximal) matching of size n in G. This can be done in time $O(nm^2)$ and the whole algorithm can be shown by induction to take time $O(|t|\ |s|^3)$, where |term| denotes the size of a term (its number of symbol occurrences). [ ]

The *enumeration* problem UNIF-AC with single variable occurrences has been studied by Lincoln and Christian [36] using a constrained exhaustive search algorithm. Their implementation has been found to perform faster than others on sample problems of size $\leq 7$ and of depth 1, but no analysis of the algorithm's asymptotic behaviour was made.

Having proved the tractability of TMATCH-AC for linear terms, Benanav, Kapur and Narendran conjecture that

> "the associative-commutative matching problem remains in polynomial time as long as it is possible to put a bound on the number of times every variable can occur in a term being matched. Since in practice, one hardly sees a variable occurring more than 2 or 3 times, it may in fact be possible to develop an efficient associative-commutative matching algorithm for such practical cases."

They also report implementing this strategy in the RRL rewriting system [4]. Unfortunately, such an approach will not be generally efficient as suggested by the next theorem.

We first need the following definition.

**Definition 11** *The (restricted) satisfiability decision problem (SAT).*
*INSTANCE: A set* $X = \{x_1, \ldots, x_m\}$ *of Boolean variables, a collection* $C = \{c_1, \ldots, c_n\}$ *of* clauses *over* $X$ *(sets of possibly negated variables, or* literals*) such that*

1. $\forall(j : (1..n))\ |c_j| \leq 3$
2. $\forall(i : (1..m))\ \#\{c_j | x_i \in c_j \text{ or } \neg x_i \in c_j\} \leq 3$

*i.e. clauses are of size 3 or less and each variable is contained in at most 3 clauses.*
*QUESTION: Is there an assignment* $X \to \{0, 1\}$ *which satisfies all clauses in* $C$ *(a clause being the disjunction of its literals) ?* [ ]

The above version of SAT is known to be NP-complete [18].

**Theorem 10** *TMATCH-AC remains NP-complete (resp. UNIF-AC remains NP-hard) even if each variable occurs at most three times in the terms to be matched (resp. unified).*
**Proof**
Inclusion in NP follows from that of the general problem. NP-hardness of the special case is shown by giving a polynomial-time reduction from SAT. Given an instance of SAT, a pair of AC-terms is constructed which is unifiable if and only if the clauses are satisfiable. The Boolean variables are represented by syntactic variables and the Boolean constants by syntactic constants $0, 1$.

Define the following symbols:

$$f \in F_{AC},\ \ 0, 1 \in F_0,\ \ g_2 \in F_2 - F_{AC},\ \ g_3 \in F_3,\ \ h \in F_n$$

$$x_1, \ldots, x_m, u_1, \ldots, u_n \in X.$$

Without loss of generality assume that all clauses contain two or three literals. Corresponding to each clause $c_j$, j:(1..n), construct a pair of terms $s_j, t_j$ as follows.

If $|c_j| = 2$ and $x_{i_1}, x_{i_2}$ are the variables occurring (possibly negated) in $c_j$ then let

$$s_j = f(g_2(x_{i_1}, x_{i_2}), u_j).$$

There are 3 pairs from $\{0,1\}^2$ which satisfy $c_j$ when assigned to $(x_{i_1}, x_{i_2})$. Let $p_{j_1}, p_{j_2}, p_{j_3}$ be those three pairs and define

$$t_j = f[g_2(p_{j_1}), g_2(p_{j_2}), g_2(p_{j_3})].$$

For example if $c_j = \{x_4, (\neg x_7)\}$ then

$$s_j = f(g_2(x_4, x_7), u_j) \text{ and}$$

$$t_j = f[g_2(0,0), g_2(1,0), g_2(1,1)].$$

If $|c_j| = 3$ with variables $x_{i_1}, x_{i_2}, x_{i_3}$, then let

$$s_j = f(g_3(x_{i_1}, x_{i_2}, x_{i_3}), u_j) \text{ and}$$

$$t_j = f[g_3(q_{j_1}), \ldots, g_3(q_{j_7})]$$

where $q_{j_1}, \ldots, q_{j_7}$ are the seven triples of $\{0,1\}^3$ which satisfy $c_j$ when assigned to $x_{i_1}, x_{i_2}, x_{i_3}$.

It is clear that the construction is computable in polynomial time from an encoding of the SAT instance.

For any j, $s_j$ and $t_j$ can be matched by any one of either 3 or 7 substitutions, depending on the size of $c_j$. Every such substitution assigns truth values to the variables of $c_j$ so as to satisfy it. If we were to match *simultaneously* $s_j$ with $t_j$ for all j, then there would be an assignment of truth values to $x_1, \ldots, x_m$ which satisfies all clauses. Conversely, $C$ is only satisfiable if $s_j$ and $t_j$ can be matched simultaneously for all j:(1..n). This condition is equivalent to the following instance of TMATCH-AC

$$s = h(s_1, \ldots, s_n) \text{ and } t = h(t_1, \ldots, t_n)$$

being successful.

Also, by hypothesis each Boolean $x_i$ occurs in at most 3 clauses and so each syntactic $x_i$ occurs at most 3 times in $t$. The only other variables are the $u_j$ which occur only once each.

We conclude that any instance of SAT is reducible to an instance of TMATCH-AC with at most 3 occurrences of each variable. [ ]

This result will be strenghtened in Chapter 5 by showing that TMATCH-AC is also NP-complete when the maximal number of occurrences of any variable is 2.

Assume now that terms are represented by simple dags, i.e. constant subterms can be repeated but variables are single nodes. In the proof of Theorem 10, consider the subgraph $G'$ consisting of all occurrences of $g_2, g_3$, the variable nodes

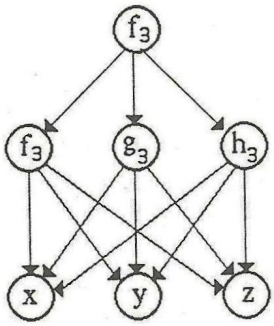


Figure 2.2: A non-planar term dag.

$x_1, \ldots, x_m$ and the edges between them. Because the clauses in SAT have size at most 3, the $g_2, g_3$ nodes in $G'$ have outdegree 3 or less. Similarly, the nodes $x_1, \ldots, x_m$ have indegree 3 or less because variables occur 3 times or fewer in SAT. It follows that $|E(G')| \leq 3\,|V(G')|$ and this also holds for the complete terms because the rest of the input dag is a tree. Therefore Theorem 10 shows that instances of TMATCH-AC can have very sparse graph structures and still form an NP-complete problem. This fact can be strengthened by requiring the planarity of subgraph $G'$.

Let us define a term to be *planar* if it can be represented as a simple dag which is planar [2]. Observe that a sparse term dag is not necessarily planar. For example it may contain the $3 \times 3$ complete bipartite graph (Figure 2.2). It is known that SAT remains NP-complete if the restriction on variables to occur at most three times is replaced by planarity of the following 'occurrence graph' [18]:

$$\{(c, x) \in C \times X \mid x \in c\}.$$

**Corollary 1** *TMATCH-AC remains NP-complete if the input terms are planar.*
**Proof**
Repeat the proof of Theorem 10 for an instance of SAT where clauses are of size at most 3 and the occurrence graph is planar. [ ]

Another parameter of term structure which could conceivably affect complexity is the depth of subterm nesting. In the case of UNIF there are NC algorithms (based on transitive closure) to unify terms of depth one, but the problem is P-complete as soon as terms have depth two [41]. With associative functions, restricting depth is not even a reasonable restriction as it would only allow terms to represent lists of bounded length, defeating the very purpose of associative unification.

[2] Strictly speaking we are discussing the planarity of the underlying undirected graph.

One could imagine restricting the depth to which *different* associative functions are nested, but this would not be strong enough. As shown in the proof of theorem 10, a single level of an associative function $f$, nested among three levels of free functions is sufficient for the NP-completeness of TMATCH-AC.

Clearly, multiple variable occurrences are the cause of complexity in unification and matching problems. A promising approach is therefore to identify properties which are generalisations of linearity but still allow tractable solutions to TMATCH-AC or UNIF-AC. We will return to this topic in Chapter 5.

### 2.4.3 A word of warning

Having proved properties of the decision problems, there is one major difference between free and equational unification which should be appreciated: most general unifiers for the latter are not always unique. For a unification algorithm to be useful in a theorem proving environment, it must not only solve the decision problem but also output a set of most general unifiers when necessary. Consider for example the associative terms $f(a, x)$ and $f(x, a)$. A decision algorithm will accept them as unifiable in little time, but a complete output of their most general unifiers would never terminate:

$$\{x \mapsto a\}, \{x \mapsto f(a, a)\}, \{x \mapsto f[a, a, a]\}, \ldots$$

Similarly, AC- or ACI-terms may have an exponential number of (independent) most general unifiers while being easily decidable. For example AC-unification of $f[x_1, \ldots, x_n]$ and $f[a_1, \ldots, a_n]$ can be decided in one traversal of the terms, but generates $n!$ most general unifiers.

In its usual applications, TMATCH-AC is used as a decision problem or as a single-output procedure but UNIF-AC is used as an enumeration procedure [16]. In general a mechanical proof by exhaustive search could require any of the unifiers in $\mu U_E$ for its correctness, so that a decision algorithm is useless for this application. Nevertheless, to compute sets of most general unifiers it is first necessary to select pairs of unifiable terms, and this is where fast decisions can be useful. Also in rewriting applications, full unification is not needed and term matching is only used as a decision problem (or with a single output).

With the AC theory, our goal is to identify tractable problems (parallelism being of little help otherwise) among special cases of the *decision* problem TMATCH-AC. For this reason, we will not deal with UNIF-AC and concentrate on the efficiency of TMATCH-AC. The next chapter introduces our restricted methodology of parallel algorithm design followed by sections on data structures and term compaction.

# Chapter 3

# Parallelism

This chapter deals with issues of algorithm and data structure design for the parallel processing of unification problems. A method is introduced for the maintenance of structure sharing (*dag compaction*) in a distributed system, based on the minimisation of finite automata.

## 3.1 Design of parallel algorithms

This section discusses constraints on the design of parallel algorithms and methods for their efficient implementation as message-passing systems.

### 3.1.1 The cost of communication

At present there is no universally accepted complexity model of parallel computation. This situation is a result of conflicting views as to how communication is to be represented. In the most successful theoretical models, communication is assumed to have unit cost ('shared-memory machines'), regardless of the number of processes involved [10][42]. Such theories study the inherent limitations of parallelism related to the subdivision and recombination of data; they are also well integrated with the better studied area of sequential complexity and algorithms.

A different view, more common among practitioners, is that communication is costly and should be taken into account ('message-passing machines'). This leads to explicit consideration of communication networks and the design of algorithms for particular network topologies. A good example of this approach is the theory of VLSI complexity [55]. Although its theoretical foundations are less well understood, we will choose the latter approach when dealing with communication. In the present state of technology — electronic processors linked by electronic/electrical signals — the complexity of parallel algorithms is a better estimate of execution time if it includes consideration of communication delays.

To allow simple and efficient implementations, we will put strict constraints on the use of computing resources. These constraints are inspired by the prag-

matics of parallel processing. If parallel unification is to be used within complex symbolic manipulation systems, simpler algorithms can only improve the larger system's modularity and reliability.
Parallel algorithms presented in this thesis are all based on the following assumptions:

1. There is no shared memory or global communication i.e all processing and communication is local.
2. Processes communicate and synchronise via unbuffered channels, a channel being unidirectional and only accessible by a fixed pair of processes.
3. Channels are only created statically.
4. The total number of channels is linear in the number of processes, and linear in the size of the input problem.
5. The communication network is a mesh of dimension 3 or less, or a combination of a fixed number of mesh networks.

The first three assumptions correspond to design decisions for the *occam* [27] programming language and have been discussed in the concurrent programming literature. Together they allow a direct mapping of processes on processors and of channels on wires.

The last two hypotheses are much more restrictive but if respected, allow the possibility of a hardware implementation which is easily expanded. In this context, the dynamic creation of channels would break the regularity of the network. Assumption 5 implies that adding a new processor requires the addition of a fixed number of wires. Assumption 4 prevents the use of, for example, a quadratic number of processes. The total resources needed for an implementation then grow at the same rate (up to a constant factor) as the size of the input, which is a realistic assumption about the use of parallel systems.

Justification for the restriction to mesh networks comes both from practical and theoretical arguments.

- There are many possible communication networks having $p$ processes, $\Theta(p \log p)$ channels and $O(\log p)$ diameter, e.g. [56] but implementing such networks requires significantly more wires than processors. This makes very large implementations impractical because of packing problems.
- It has been proposed that optical computing technology could make highly connected networks practical by using lenses and lasers to replace wires. However, if optical logic gates are also used then 'lens based systems become severely propagation limited, with proposed systems having propagation delays as much as 1000 times larger than the logic gate speed' [61].
- Geometric results concerning sphere packing imply that the maximum or average physical distance between two of $n$ processing elements is $\Omega(\sqrt[3]{n})$.

This holds even if 'wires' have effectively no volume (e.g. lasers). From the assumption of the bounded speed of light, it follows that $\Omega(\sqrt[3]{n})$ is an absolute lower bound on the time required for any computation using $n$ processors [59].

- In our experience [22], algorithms for regular networks lead to simple concurrent programs. Such programs make it easier to reason about the precise tradeoffs between computation and communication and are therefore expandable to larger networks with predictable efficiencies.

We will not consider 3-dimensional (3D) mesh algorithms. Their design would not be fundamentally different from 2-dimensional (2D) algorithms, and the asymptotic decrease in diameter from 2D to 3D is small:

$$\sqrt{n}/\sqrt[3]{n} = \sqrt[6]{n}.$$

The distinction between 2D and 1D (linear) arrays is essential however. There are two differences between one- and two-dimensional meshes. Diameter is the the obvious difference: $\Theta(\sqrt{n})$ for a 2D mesh of $n$ processes, instead of $\Theta(n)$. The other difference is essential: *I/O bandwidth* or the number of possible inputs and outputs per unit of time. The 2D mesh can make $\Theta(\sqrt{n})$ I/O operations in parallel while the linear mesh is limited to a fixed number of I/O per time step. Therefore, the potential $\Theta(\sqrt{n})$ speedup offered by a rectangular architecture may not be available if the 'environment' communicates serially. This problem does not arise with a linear array because its internal bandwidth is the same as its I/O bandwidth. The issue of bandwidth and loading time will be raised in Chapter 4 in relation to the TMATCH problem.

In any case, the reader should note that the times suggested by Yasuura's $O(n' \log n')$ algorithm [62][63] ($n'$= number of variables) are not possible on mesh architectures. If the input has size $n$ and $\Theta(n)$ processes are used, then the network diameter makes any algorithm $\Omega(\sqrt{n})$ whatever the value of $n'$.

### 3.1.2 Parallel task decomposition

The first question one should answer while designing a parallel algorithm is how to divide the task among processes. There are a few decomposition methods which have proven generally applicable [24] namely algorithmic decomposition, domain decomposition and process farms. Their respective merits and demerits are discussed below.

1. *Algorithmic decomposition*
   To express a sequential algorithm one usually decomposes the problem into procedures for unrelated tasks such as formatting, searching and output. The analogous method for parallel processing is called *algorithmic decomposition* (or *dataflow parallelism*). A typical example is to define one process for each of a number of sequential tasks and to combine them into a linear

array. The result is a pipeline for overlapping successive executions of a sequential algorithm.

Algorithmic decomposition is not restricted to linear process arrays and generally involves an arbitrary (problem specific) communication network among a *fixed* number of processes. Such a network does not violate the regularity requirement because its size is constant with respect to the size of the input problem instance. For the same reason, algorithmic decomposition can only provide a fixed amount of parallelism: constant accelerations over sequential execution. It is however very useful in structuring concurrent programs.

There are many equivalent algorithmic decompositions possible for unification. By far the most useful and elegant is that given by C.Kirchner [33][34]. Kirchner identifies three operations from which unification algorithms can be built for a large class of equational theories:

- 'multiequation decomposition' (DEC)
- 'multiequation merging' (MER)
- 'multiequation mutation' (MUT)

It is useful to reason about unification as a parallel or sequential composition of these operations. In the context of solving UNIF and UNIF-AC, the three operations can be described as follows.

DEC is the creation of subterm unification tasks from an initial pair of terms. For $f$ a free function of $F_n$ and terms $t_i, s_i$, $f(t_1, \ldots, t_n)$ will be unified with $f(s_1, \ldots, s_n)$ only if $t_i$ is unified with $s_i$ for $i = 1, \ldots, n$. In this case DEC would create $n$ subproblems from the initial one. If the subproblems are to be processed in parallel, then DEC can be implemented as a broadcasting operation.

MER is the combination of subterm unifiers into a consistent unifier for larger terms, consistency being the uniqueness of values for occurrences of a variable. In the last example, suppose $n = 2$ and the variable $x$ occurs in both $t_1$ and $t_2$. Let $\sigma_1, \sigma_2$ be such that

$$\sigma_1(t_1) = \sigma_1(s_1) \text{ and } \sigma_2(t_2) = \sigma_2(s_2).$$

Then $\sigma_1 \cup \sigma_2$ will not be a unifier for $f(t_1, t_2)$ and $f(s_1, s_2)$ unless $\sigma_1(x) = \sigma_2(x)$. Such constraints on unifiers are maintained by *merging* equivalence classes of subterms. If unifiers for subterms are created in parallel then MER can be implemented as a merging operation, where a process concentrates messages from different sources. A different strategy is to avoid placing terms involving the same variables in different processes [6]. This simplifies MER to be a union operation on unifiers.

Algorithms for free unification require only combinations of DEC and MER. Their classical implementations are divide-and-conquer sequential algorithms, but DEC and MER can also be combined into a nondeterministic

rewriting algorithm such as Martelli and Montanari's [39]. Kirchner [33][34] has generalised Martelli and Montanari's algorithm to equational theories.

In handling equational theories, a third operation MUT is required. MUT (*mutation*) consists in solving a unification equation to which DEC does not apply because its root symbols are not free functions. In general MUT is problem specific as it depends on the axioms involved. For UNIF-AC, it corresponds to solving linear Diophantine equations, as was discovered by Stickel [53].

Our interest in the DEC-MER-MUT decomposition is to use it as an abstract specification for unification methods. We will not consider the associated rewriting algorithms.

2. *Domain decomposition*

   When the problem space (domain) possesses a natural repetitive structure, it is advantageous to create subproblems corresponding to the divisions of this structure. For example if the input is a matrix each process may be assigned to a submatrix. This type of subdivision is called *domain decomposition* (or *geometric parallelism* or *data structure parallelism*).

   Typically the allocation of subproblems to processes is uniform, depending only on the size of the input. Processes are given an equal share of the problem to balance their processing loads and maximise parallel efficiency. Domain decomposition allows as much parallelism as the size of the input.

   Processes tend to be identical and do not require large amounts of local store as their individual processing task is often simple and applied to a small set of data.

   We will use domain decomposition in a systolic algorithm for the compaction of term dags.

3. *Process farms*
   The third decomposition method is to create what is called a *process farm*. The method is called *task farming* or *task contracting*. A process farm consists of a central *master* process which dynamically creates and sends subtasks in the form of 'work packets' to a set of identical *worker* processes. The workers only communicate with the master to receive subtasks and return results. They do not communicate among themselves.

   A process farm is most useful when there is no possibility of a balanced static decomposition. Work packets may require unequal amounts of processing and so the workers execute asynchronously. The master maintains lists of busy and idle workers. When a work packet is created it is passed to an idle worker which is then registered as busy until receipt of the corresponding result by the master. As long as communication to and from the workers is significantly faster than processing, there is no loss of available processing power.

A process farm has the advantage of dynamically balancing processing loads. It also allows unbounded parallelism, for example in a combinatorial search. The number of workers is only limited by the availability of hardware units and the amount of store available for each process. This last constraint can be the main limitation on the size of a process farm. Workers must be independent of each other and can only communicate small packets with the master if the farm is to function effectively. This can create replication of calculations or duplication of data. The storage requirements of workers can therefore be substantial and prevent their implementation as very small processing elements.

### 3.1.3 Process communication

Once a task is decomposed into processes, it remains to define a connection network and an appropriate communication protocol to combine them into an algorithm for the original problem. In general, the process connection network is independent of the type of decomposition used. For example a linear array of processes can be used for either algorithmic or domain decomposition or as a process farm. While task decomposition is problem specific, process communication tends to be implementation specific and possibly independent of the problem itself.

In designing complete parallel systems, it is always useful to use a combination of parallel decomposition methods. When choosing an implementation strategy, it is useful to remember that algorithmic and domain decompositions divide the storage requirements among processes, while a process farm usually multiplies the amount of storage by the number of worker processes.

### 3.1.4 Correctness and complexity

Subject to the design decisions made earlier, discussion of unification algorithms will now be limited to their correctness and worst-case complexity. Other aspects of the algorithms or their implementation will be collectively treated as 'implementation details' as they are mostly system dependent. It is our belief that the restrictions on algorithms are sufficient to ensure that important 'details' such as process allocation and parallel efficiency can be safely overlooked at this level of design. This abstraction has the advantage of making the algorithms independent of any particular software or hardware system.

**Correctness:**
All procedures should terminate after a finite number of steps from any given input. On termination, decision algorithms should correctly indicate the existence or absence of a unifier and output a unifier where appropriate.

Methods which allow more parallelism at the expense of probabilistic outputs are not suitable for general purpose unification algorithms. A proof system using

probabilistic unification would, strictly speaking, produce uncertain proofs. Such methods are best left to the design of specific applications where their impact can be measured in meaningful ways.

**Worst-case complexity:**
An upper bound on the worst-case time complexity of the algorithm should be given. This bound should include consideration of the number of processes and the number of messages passed along channels during execution. For this purpose, processes are understood as sequential Random Access Machines (RAM) [1] with instructions for manipulating and communicating *words* i.e. numbers or characters, as opposed to Boolean values.

### 3.1.5 Sharing vs copying

An important issue in symbolic manipulation is the choice between sharing and copying repeated objects. Regarding unification, pure structure sharing implies manipulating terms as compact dags and pure structure copying maintains them as trees or simple dags. There are advantages and disadvantages to both methods but structure sharing should be preferred in sequential processing because its consistent use avoids the possibility of an exponential growth for objects with repeated components (e.g. a term such as $f(f(f(x,x),f(x,x)),f(f(x,x),f(x,x))))$. Sharing therefore provides worst-case savings in both time and space. However, its implementation may introduce so-called redirection pointers, which increase the average access time to subterms. We will choose to ignore the overheads so as to design general purpose methods, and simply observe that many (sequential) systems successfully use structure sharing: Prolog interpreters [7], Lisp interpreters [17] and all functional language interpreters based on graph reduction [44].

Nevertheless, when introducing parallel term manipulations, structure copying becomes attractive or necessary for one of two reasons.

The first reason arises when designing shared-memory algorithms, as has been suggested for parallel functional programming machines [44]. In this context, operations involving writing to a shared term must be implemented using a restricted access protocol, if not mutual exclusion between processes. Apart from creating logical complications, this reduces the amount of parallelism available to what is permitted by the shared writing protocol. For example if processes $P_1$ and $P_2$ are to respectively update terms $t_1$ and $t_2$, it is possible that $t_1$ and $t_2$ share a subterm $t_3$ but that at most one of the processes ever accesses $t_3$ at the same time. Unless this fact is detectable by the system (which is unlikely in general) $P_1$ and $P_2$ will only have sequential access to $t_1$ and $t_2$, reducing overall speed. This leads to the possibility that separate copies of $t_1$ and $t_2$ — and hence of $t_3$ — may be processed substantially faster than their shared representation. In shared-memory parallelism, copying is probably necessary to obtain substantial speedups.

The second possibility for introducing copying is the use of distributed mem-

ory parallelism, as proposed here. Consider the situation where two *distributed* processes $P_1$ and $P_2$ are to read/update respectively terms $t_1$ and $t_2$ which share subterm $t_3$. Suppose again that at most one of $P_1$ and $P_2$ ever accesses $t_3$ at the same time. If a single copy of $t_3$ exists, then the best possible situation is that it happens to be stored in the local memory of one of the two processes involved, say $P_1$ together with the rest of $t_1$. $P_2$ can then only store $t_2 - t_3$ in local memory and must communicate with $P_1$ for every (read *or* write) access to $t_3$, thus needlessly interrupting execution of $P_1$. This situation could occur with an arbitrary number of processes, and force their sequential execution.

The only possible solution is again to copy the shared objects for independent parallel processing. Apart from the obvious problem of consistency, maintaining different copies of a single 'object' forces compaction when the copies are eventually merged. It is also possible that repetitions are created through the manipulation of different objects, hence the need for structure sharing within individual processes and for a general compaction algorithm to be applied when merging objects from different processes.

## 3.2 Data structures for terms

This section introduces the concrete representation of term dags as *pointer structures* (also called *linked lists*) and discusses their manipulation.

It is common to assume that terms are represented as trees or simple dags of size proportional to the number of symbol occurrences. This seems implicit in most papers concerned with equational unification. In theory this assumption is not harmful because abstract algorithms for simple dags can always be adapted for compact dags so as to avoid duplication of tasks. In designing detailed algorithms however, the issue cannot be overlooked as it could make an exponential difference in time and space. Whenever possible, terms should be represented by compact dags and should remain compact as they are processed.

As large terms are processed in parallel two important operations are those of decomposition and recombination (or merging). Decomposition is the division of a dag into subgraphs for distributed processing; the results can eventually be merged for centralised processing, output or communication to another process. According to the previous remarks, recombination should ensure that the result of merging or any other dag transformation does not contain repeated subgraphs.

The process of eliminating repeated subgraphs will be called *dag compaction*. The reader acquainted with list manipulation systems should notice that 'compaction' is used here in a more general sense than in some contexts where it refers to a renumbering of pointers and addresses to place a structure in contiguous locations. Dag compaction will be used here to mean the minimisation of the number of nodes. [1] A fast algorithm for compaction is given in Section 3.3.

[1] Hansen [23] uses the expression 'compact lists' for lists which are stored sequentially in memory. However, the Oxford Dictionary of Computing defines 'compaction' as 'Any of a number of methods to reduce unused or unusable space in primary, secondary, or other memory',

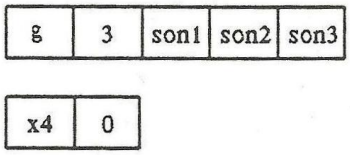


Figure 3.1: node records

### 3.2.1 Pointer structures for free terms

The most flexible and useful representation of graphs is that of pointer structures (linked lists) and we will adopt it for term dags. A dag node will be stored as a record containing one field for its symbol, one for its arity and one for the address of each of its sons, in order of increasing edge label (left to right). It can also include a few 'workspace' fields, not represented here. The structure for a complete term is then a set of node records satisfying the defining properties of a term dag (unique root, no cycles etc). The number of words occupied by the record is twice the number of vertices plus the number of edges, which is directly proportional to the size of the dag.

For implementation purposes it is possible to use only records of size four or less i.e. functions of arity two or less. This is done by converting to and from the usual syntax on input and output. For example a term of the form $f(t_1, t_2, t_3)$ where $f \in F_3$, can be converted to $f_{(1)}(t_1, f_{(2)}(t_2, t_3))$ where $f_{(1)}, f_{(2)} \in F_2$ are new function symbols created while preprocessing. Processing would then involve only $f_{(1)}, f_{(2)}$ and the conversion would be reversed on output.

We will ignore this detail and treat a function occurrence of any arity as a single record. The data structure is then no different from usual representations of graphs, except that nodes have outdegrees bounded by

$$k = \max\{\text{arity}(f) | f \in F\}.$$

With this representation, basic operations on terms are implemented by standard graph algorithms. Searching is done by traversal, in time proportional to the size of a dag. Duplication requires only the addition of a new pointer, in constant time. Acyclicity testing is done by depth-first search (a cycle is detected if and only if a 'back edge' is encountered). Substitution of a term for a variable is done in constant time if the dag is compact (so that the variable is stored as a single record), and the variable record has been located. Application of a multivariable substitution $\sigma$ to a compact dag is done in time $|\text{domain}(\sigma)|$ if the variables have been located, or by a single traversal of the dag otherwise.

---

which clearly includes the present definition as a special case.

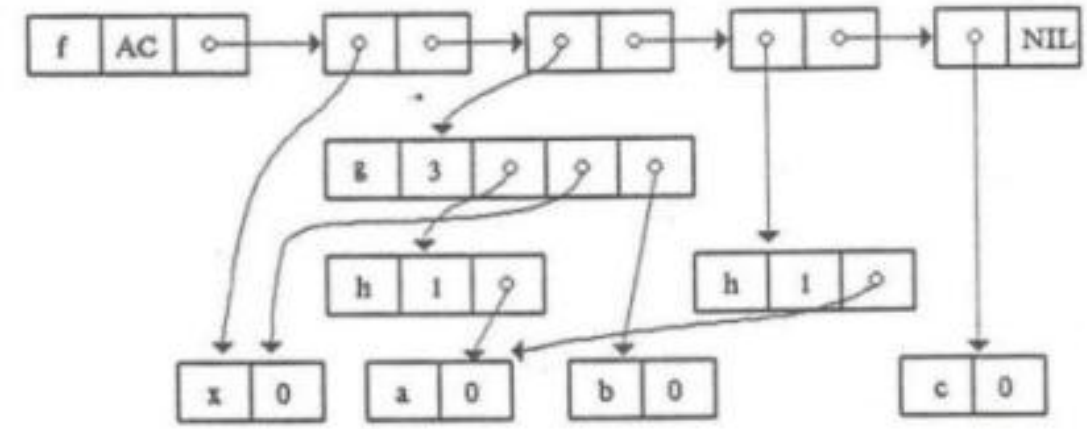


Figure 3.2: $f[x, g(h(a), x, b), h(a), c]$, where $f$ is associative-commutative.

### 3.2.2 Pointer structures for AC-terms

Nodes for associative-commutative functions (in fact for associative functions in general) are stored as lists corresponding to their arguments in flattened form. There is a header record with one field for the function symbol, the special value 'AC' stored in the arity field and a pointer to the beginning of the list of arguments. It is understood that references to the AC-function symbol are to be pointers to this header record, and not to any other part of the AC-lists. The list itself consists of records with two pointer fields. The first one contains the address of the root of an argument term and the second one points to the next list record. The second field of the last list record contains the special value 'NIL'. Arguments themselves (which can be free terms or AC-terms) are stored elsewhere. If a pointer structure rooted at a function $f$ from $F_{AC}$ is not in flattened form, it can be converted to flattened form by depth-first traversal with simple pointer manipulations to insert arguments with $f$ as symbol into the AC-list. One traversal can flatten all occurrences of AC-functions. Searching, duplication, acyclicity testing and substitution are implemented as with free terms.

### 3.2.3 Level sorting

As discussed above, top-down traversal can be used to implement many operations on the pointer data structures. This subsection describes preprocessing for bottom-up traversal to be used in dag compaction. With the given data structures for dags, if a bottom-up traversal algorithm is to be efficient, each one of its 'visits' at a node should take only a bounded number of operations. It is thus not efficient to search for parent nodes between visits. One way to avoid search is to compute a table of parents for each node (via any top-down traversal). An equivalent solution is to partition the nodes into *levels* so that

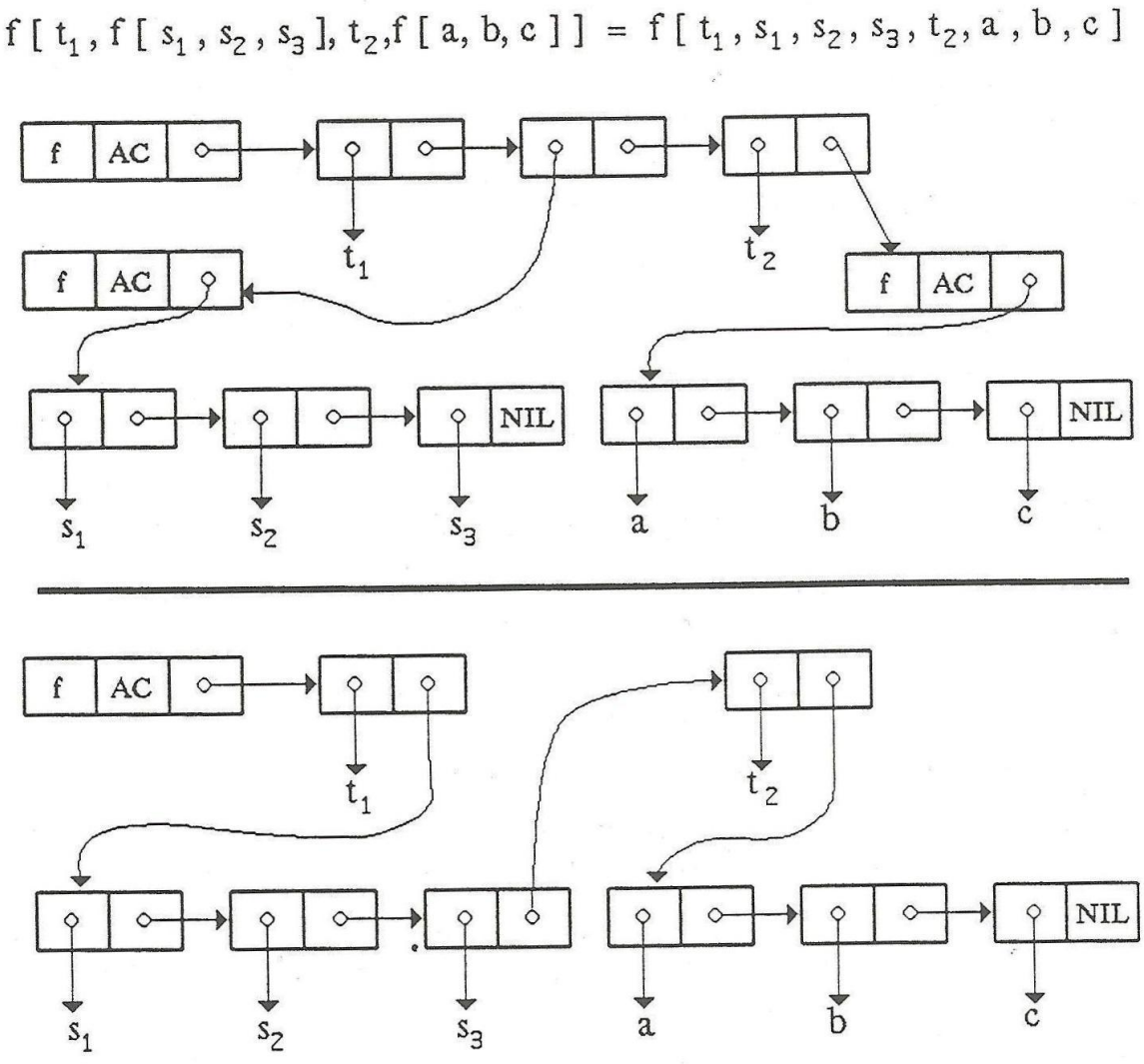


Figure 3.3: Flattening an associative-commutative term.

bottom-up traversal first visits nodes of level 0, then nodes of level 1 and so on. Partitioning of a dag into levels will be called *level sorting*. We will adopt this method of bottom-up traversal: first sort the dag into level lists, then visit the lists in increasing order of level.

Levels are defined here as lengths of maximal paths to the leaves, and not as distances from the root (depths). A consequence of the definition is that a parent of a node at level $i$ is not necessarily at level $i+1$. The reason for this definition will become clear when dag compaction is considered.

**Definition 12** *The* level *of any node $v$ in a free term dag $G$ is the length of the longest path in $G$ starting at $v$ i.e. the depth of the subgraph rooted at $v$. Its value is in the range $(0..depth(G))$. Nodes at level 0 are called* leaves *or* sinks*. There is only one node of level $depth(G)$, namely the root of $G$.*

**Lemma 4** *Let $v$ be an internal node with sons $v_1, v_2, \ldots, v_k$, then*

$$level(v) = 1 + \max\{level(v_r) | r \in (1..k)\}.$$

*The proof is by straightforward induction on the level itself.* [ ]

The details of a sequential algorithm for level sorting are given below. It is included for completeness and the reader will no doubt find variations on this method, perhaps more appropriate for his/her application.

The algorithm is based on the observation that during a depth-first traversal, whenever the process backtracks from a node $v$ the whole subgraph rooted at $v$ has been visited. Assuming that the levels of $v$'s sons have been computed, the lemma allows computation of level($v$) in time proportional to its outdegree (arity).

Let $G$ be the given dag with root $r_G$, vertices $V(G)$ and edges $E(G)$. To implement level sorting, node records have to be augmented with an additional field, used either as a flag to indicate that a node has not been visited, or used to store the level value computed previously. Assume therefore the existence of an array *level* of type

$$\textbf{Array } V(G) \textbf{ of } (\text{Integer} \cup \{\text{not-visited}\})$$

where *not-visited* is a special value distinct from integer values. If the traversal is to take time $O(|E(G)|)$, then nodes should be visited only once and it is necessary to assume that before execution of the algorithm all level fields are set to not-visited. Because $G$ is only given as a pointer to $r_G$ and because its records could be scattered anywhere in memory, it would appear that satisfaction of this precondition requires an initialisation traversal of $G$. Fortunately there is a more elegant solution which is an adaptation of a method for the dynamic initialisation of arrays. This method avoids preprocessing at the expense of one more data structure, and will be explained after the level sorting algorithm itself.

Execution of levelsort($r_G$) follows a depth-first traversal of $G$ while accumulating values in the level array. On its first visit to a node, the level is set to $-1$. If the node has sons, their levels are recursively computed and the father's level is set to their maximum. Regardless of the existence of sons, the level value is then incremented by 1. On subsequent visits to a node, the level is already computed and no steps are executed.

**Algorithm 3** Level sort.
**Precondition**
$\forall(v : V(G))$. level[$v$] = not-visited.
**Main process**
levelsort($r_G$).

**procedure** levelsort (v: V(G))
1. **if** (level[v] = not-visited) **then**
2. level[v]:= -1
3. **for** r=1 **to** arity(v) **do**
4. levelsort(son[v][r])
5. level[v]:= max{level[v], level[son[v][r]]}
6. level[v]:= level[v] + 1
**end procedure**

**Postcondition**
$\forall(v : V(G))$. level[$v$] = the length of the longest path which begins at $v$ [ ]

**Proposition 8** *Execution of levelsort with the given precondition establishes the postcondition in $O(|E(G)|)$ operations.*
**Proof**
We will show by induction on the level of $v_0 \in V(G)$, that execution of levelsort($v_0$) sets level[$v_0$] to the level of $v_0$.
If $v_0$ is a leaf then arity($v_0$) is 0 and the algorithm reduces to:

**if** (level[$v_0$] = not-visited) **then**
level[v]:= -1
level[v]:= level[v] + 1

If levelsort($v_0$) was executed before then we already have level[$v_0$] = $0 \neq$ not-visited. Otherwise the body of the conditional is executed and level[$v_0$] is correctly set to 0.

Suppose now $v_0$ is an internal node with sons $v_1, \ldots, v_{\text{arity}(v_0)}$. If the call levelsort($v_0$) was already executed there is nothing to prove. Otherwise instruction 2–6 are executed and correctness of the final level value follows directly from the lemma and the induction hypothesis applied to $v_1, \ldots, v_{\text{arity}(v_0)}$.

Consider now the number of times each instruction (1–6) is executed, as a function of the size of the graph.

Instruction 1 is executed once for every edge entering a given node.
Instructions 2, 3 and 6 are executed once for every node.
Instructions 4 and 5 are executed once for every outgoing edge from a given node.
The total execution time is therefore proportional to

$$3\,|E(G)| + 3\,|V(G)| \;\; \leq \;\; 6\,|E(G)|$$

as required [ ]

The output of levelsort is more useful if given as a partition of the nodes and the algorithm can be extended to output this partition during traversal. If there is a known bound $d$ on the depth of the dag, then the partition can be represented as an array of $d$ sets and the nodes added to the appropriate set by direct access. In general however, depth is not known in advance and it is necessary to represent the partition as a growing list of sets. Assume then the existence of a variable *levset* of type

$$\textbf{List of}\,(\mathit{Integer} \times \textbf{List of}\,V(G))$$

whose elements are level numbers with corresponding sets of nodes, and are kept in increasing order of level. To maintain direct access to all lists in levset, replace the integer values of the level fields in node records by pointer values, directed at the corresponding list in levset. The lists are labeled by their (integer) level, and this also provides direct access to the value of a node's level. The extended algorithm is as follows.

**Algorithm 4** Level sort with partition output.
**Preconditions**
$\forall(v : V(G))$. level($v$) = not-visited
levset = [ (0, [ ]) ].
**Main process**
levelsort($r_G$).

```
procedure levelsort (v: V(G))
if (level[v] = not-visited) then
  if (arity(v) = 0) then
    {add v to the first list in levset and set level[v] accordingly}
    level[v]:= first(levset)
    second( head( levset)):= second( head(levset)) cons [ v ]
  else
    lev:= -1
    for r=1 to arity(v) do
        levelsort(son[v][r])
        if first(level[son[v][r]]) > lev then
          lev:= first(level[son[v][r]])
          level[v]:= level[son[v][r]]
    { level[v] points at the level list of highest label among its sons.}
```

```
    if (level[v] = last(levset)) then
        { enlarge levset with a new list }
        levset:= levset cons [( lev + 1, [ v ])]
        level[v]:= last(levset)
    else
        level[v]:= successor of level[v] in levset
        second(level[v]):= second(level[v]) cons [ v ]
end procedure
```

**Postconditions**
$\forall(v : V(G))$. first(level[$v$]) = length of the longest path which begins at $v$.
The i-th list of levset contains exactly the nodes of level i in $G$ [ ]

To avoid the initialisation pass of the depth-first search, adapt a technique used in Aho et.al. ([1] p.71, 2.12 or [5]) for the dynamic initialisation of arrays. Replace the level field in each node record by a pointer and maintain the pointer in each visited node to the address of a record on a stack. Records on the stack contain a pointer back to the node which points to them and a field for the level of that node.

On its first visit to a node, the traversal algorithm could find a pointer value directed (by chance) to a record on the stack. However, this record would not point back at the node and the algorithm would then correctly identify the node as not visited.

This method is also useful in implementing searching, flattening or acyclicity testing as described earlier. It applies to the depth-first traversal of any graph given as a pointer structure, possibly scattered in memory.

Another possibility [2] is to use 'visited' flags on the nodes and follow each traversal of a graph by a second traversal where the flags are reset to 'not visited'. To make the second traversal non redundant, simply use the flags set by the first traversal but with reversed meanings. Such a pair of traversals must be an atomic operation if it is to function correctly.

Of course, when the records defining a dag are all localised in a certain region of memory — perhaps in an array — it is simpler to initialise the whole region and apply directly the original depth-first search algorithm. However, substitution operations may break the locality of records and it is generally not safe to assume locality. The tradeoffs involved in maintaining locality of pointer structures have been an early consideration in the design of Lisp interpreters (see for example [23] where the expression 'compact list' refers to a Lisp list stored sequentially in memory rather than chained with pointers; in this context, 'list compaction' is an operation which simply readdresses pointer records, not to be confused with our definition of dag compaction).

In any case, the dynamic initialisation method allows (non redundant) depth-first traversal of graphs in the most general case where their nodes are stored at random locations.

[2] suggested by Quentin Miller

### 3.2.4 Level sorting of AC-terms

Two representations can be defined for a flattened term of the form $f[t_1, \ldots, t_n]$ where $f \in F_{AC}$. The abstract representation is a dag whose root is labeled with $f$ and has outdegree $n$. The concrete representation is a pointer structure with a record for $f$ and a list of pointers directed at structures for the $t_i$.

Suppose the $t_i$ are free terms, all of depth $d$. Then the node for $f$ is at level $d+1$ and should be placed in the $d+1^{st}$ list of nodes during level sorting. However if the level sorting algorithm is applied to the concrete representation of $f[t_1, \ldots, t_n]$ the pointer records in $f$'s list of arguments will create a chain of depth $n$ so that $f$ will be assigned incorrectly to level $d+n+1$. It is therefore necessary to apply minor changes to the level sorting algorithm, to treat the list of arguments to an AC-function as an (unordered) set of dags, and not as a chain of function symbols.

Let $G$ be as before the abstract representation of our term, so that an associative-commutative function symbol is a single node in $V(G)$. Assume also that the dag is in flattened form with respect to its AC-functions.

**Algorithm 5** Level sort for an AC-term (without partition output).
**Preconditions**
$\forall(v : V(G))$. level[$v$] = not-visited
$\forall(v : V(G))$. (symbol($v$) $\in F_{AC}$) $\Rightarrow$ ($\forall(v, w) : E(G)$. $\neg$(symbol($w$) $\in F_{AC}$))
levset = [ [ ] ].
**Main process**
levelsort($r_G$).

**procedure** levelsort (v: V(G))
**if** (level[v] = not-visited) **then**
  level[v]:= -1
  **if** symbol[v] **not in** $F_{AC}$ **then**
    **for** r=1 **to** arity(v) **do**
      levelsort(son[v][r])
      level[v]:= max{level[v], level[son[v][r]]}
  **else**
    **for** $(v, w) \in E(G)$ **do**
      levelsort(w)
      level[v]:= max{level[v], level[w]}
  level[v]:= level[v] + 1
**end procedure**

**Postcondition**
$\forall(v : V(G))$. level[$v$] = the length of the longest path which begins at $v$ [ ]

## 3.3 Compaction and merging

This section deals with the problem of maintaining term dags in compact form.

In a distributed treatment of unification, repetitions of subterms may be created through independent bindings in different processes. When the subterms are merged in a single process, all such repetitions can be combined to obtain a compact dag.

### 3.3.1 Background

As explained earlier, maximal sharing of subterms is preferable in sequential processing but copying may be necessary for efficient parallel term manipulations. When terms are handled by separate processes and then recombined in a single process, it may be necessary to recompact the resulting merged term.

Non-shared subterms may contain variables and substitutions will change their values and so independent application of substitutions may create new repeated subterms, to be reduced to a single shared structure when merging. For this reason it is desirable to apply a general dag compaction algorithm which will reduce any term dag to an equivalent compact representation. By applying it to the result of a substitution or merging of terms, a compaction algorithm ensures efficiency of subsequent sequential processing. It also avoids the possibility of an exponential growth in size through repeated copying and substitutions. A general-purpose compaction algorithm must identify all repeated instances of terms.

Dag compaction can be used as a primitive operation for distributed term manipulations, which justifies finding the fastest possible algorithm for it. As an abstract computational problem, it is related to several previously studied problems and this will provide useful knowledge about its complexity and algorithmic solution.

### 3.3.2 Terms as finite automata

We now prove the uniqueness of the equivalent compact dag and show that algorithms for the minimisation of automata can be used for its computation.

For any given term, there are multiple dag representations and each dag can be implemented as any one of an infinite set of equivalent pointer structures. The equivalence of pointer structures is simply a matter of renumbering addresses and can be safely overlooked (it is a concrete version of graph isomorphism and so preserves all graph properties).

For the purpose of compaction, we are therefore concerned only with the correspondence between a term $t$ and the set of its dag representations. This set is never empty because it contains the unique tree for $t$. Consider a dag representation $G_t$ of $t$ and the operation on $G_t$ which consists in merging pairs of

nodes where equal subterms are rooted. Let such nodes be called *equivalent*. This operation reduces the number of nodes (and therefore edges) in the graph and must therefore terminate. Its result is by definition a compact dag representing $t$.

Thus there always exists a compact representation $G(t)$ for $t$. Its minimal size and uniqueness can be proved by an independent argument but it is easiest to use some elementary automata theory. The correspondence with automata will also clarify the process of algorithm design for dag compaction. The key to this correspondence is the observation that the term represented by a dag $G$ is uniquely defined by the set of paths from the root of $G$, together with the symbols encountered along each path.

Recall now the definition of the set of occurrences $\mathrm{O}(t)$ for $t \in T(F, X)$. This set contains all the sequences of edge labels that occur on paths from the root of a dag representation for $t$. Given $\mathrm{O}(t)$, $t$ can be uniquely defined as a map

$$t : O(t) \to (F \cup X)$$

such that, for $s \in \mathrm{O}(t)$, $t(s)$ is the symbol found at the end of the path defined by $s$.

Let $k$ be the maximum arity of all function symbols appearing in $t$. Then $\mathrm{O}(t)$ can be seen as a simple formal language over the alphabet $\{1, 2, \ldots, k\}$: a star-free regular language. A dag $G$ for $t$ corresponds directly to an automaton for accepting the sequences in $\mathrm{O}(t)$ of maximal length, i.e. from the root to the leaves. The following table makes this correspondence explicit.

| | |
|---|---|
| nodes $V(G)$ | states |
| root node | initial state |
| leaf nodes | accepting (final) states |
| edge labels ($\in \{1, 2, \ldots, k\}$) | inputs |
| node symbols ($\in F \cup X$) | outputs |
| edges $E(G)$ | transitions |
| dag $G$ | acyclic automaton with output |

Dag $G$ then is seen as an acyclic deterministic finite automaton (DFA) with an output function $V(G) \to F \cup X$. A DFA with an output associated with every state is called a *Moore machine* [25], and so

a term dag $G$ corresponds to an acyclic Moore machine $M(G)$.

Notice that some sequences of $\{1, 2, \ldots, k\}^*$ do not occur on downward paths in $G$ and so some transitions are undefined in $M(G)$. However, for any valid input sequence, the behaviour of $M(G)$ is defined as a sequence of output symbols in $(F \cup X)^*$ and a final state.

**Definition 13** *Two Moore machines $M_1, M_2$ with the same input and output alphabets are* equivalent *if they are equivalent as DFAs and their outputs are the*

*same. In other words $M_1 \equiv M_2$ when for every input sequence $s$, the output sequences of $M_1$ and $M_2$ are the same, and the state $M_1(s)$ is final if and only if $M_2(s)$ is final.*

Clearly, for term dags $G_1, G_2$, the associated Moore machines are equivalent if and only if $G_1$ and $G_2$ represent the same term. The problem of finding an equivalent dag of minimal size therefore reduces to that of finding an equivalent minimal Moore machine. This problem in turn, reduces to the well-known problem of DFA minimisation.

**Lemma 5** *(Hopcroft and Ullman [25])*
*For any given regular language, there is an accepting DFA with a minimal number of states. This minimal automaton is unique up to a bijection on the states (isomorphism of its transition graph).*

**Lemma 6** *For any given DFA $A_1$, there is an equivalent DFA $A_0$ with a minimal number of states, which is unique up to isomorphism.*
**Proof**
Apply the above lemma to the language accepted by $A_1$ [ ]

**Proposition 9** *For any acyclic Moore machine $M$ of input alphabet $\{1, 2, \ldots, k\}$ and output alphabet $F \cup X$, there is an equivalent acyclic Moore machine $M'$ with a minimal number of states, which is unique up to isomorphism.*
**Proof**
Let $\phi$ be a transformation from acyclic Moore machines with alphabets as above to acyclic DFAs with alphabet $\{1, 2, \ldots, k\} \cup (F \cup X)$. Define $\phi$ as a local transformation on the transition graph of $M$: it replaces every node (state) by a pair of nodes linked by a single transition edge. The first node of a pair will be called 'black' and be given a single outgoing edge to the second node which will be called 'white' (see figure 3.4).

A black-to-white edge in $\phi(M)$ is labeled with the output symbol of the node it replaces in $M$ and the edges of $M$ become white-to-black edges. An accepting state is thus replaced with a pair of nodes of which only the white is accepting, and the root is replaced by a new initial black state. By construction, $\phi$ is invertible (because $(F \cup X)$ and $\{1, 2, \ldots, k\}$ are disjoint) and $\phi(M)$ accepts the language which consists of maximal length valid inputs to $M$, interleaved with the corresponding outputs. This language is a subset of $(\{1, 2, \ldots, k\}; (F \cup X))^*$ where the semicolon denotes concatenation.

By the last lemma, there must be a unique minimal acyclic DFA $A_0$, equivalent to $\phi(M)$. Because it accepts the same language as $\phi(M)$, $A_0$ must have a black initial state, white final state, etc. It therefore lies in the range of $\phi$ and the acyclic Moore machine $M' = \phi^{-1}(A_0)$ must be equivalent to M.

Let $n'$ be the number of states in $M'$. If there existed another acyclic Moore machine $M''$ equivalent to $M'$ and $M$ with $n'' < n'$ states, then there would be a DFA $\phi(M'')$ with $2n'' < 2n'$ states and equivalent to $\phi(M)$. But $A_0 = \phi(M')$ has

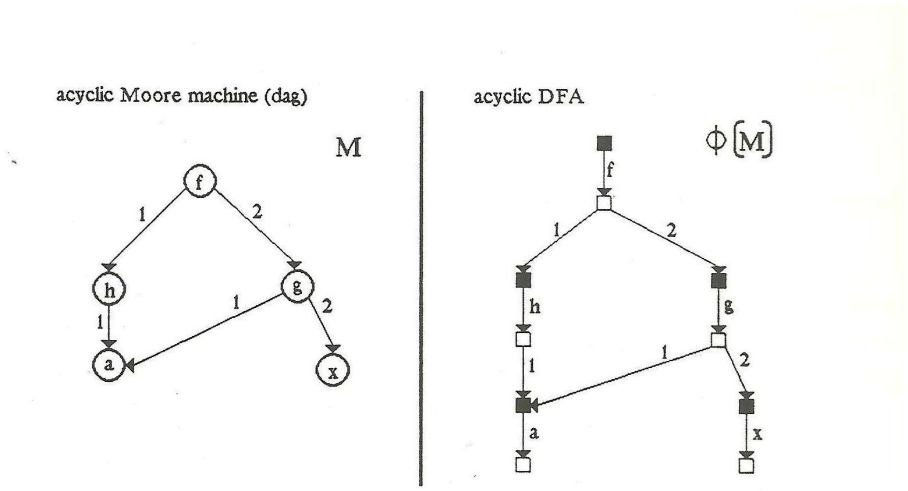


Figure 3.4: The invertible transformation $\phi$ from Moore machine to DFA.

$2n'$ states and this would contradict its minimality. We therefore conclude that $M''$ has as many states as $M'$. In fact, the uniqueness of $A_0 = \phi(M')$ implies that $\phi(M'')$ is isomorphic to it. It follows that $\phi^{-1}\phi(M') = M'$ and $\phi^{-1}\phi(M'') = M''$ are also isomorphic. [ ]

**Corollary 2** *For any term dag G, there is an equivalent term dag of minimal* size*, unique up to isomorphism.*
**Proof**
The proposition applies directly to G if it is interpreted as an acyclic Moore machine. A simple argument concerning the arity of nodes, shows that the equivalent dag with the fewest states is also the one with the fewest edges. This dag is therefore of minimal size. [ ]

**Corollary 3** *For any (free) term $t \in T(F, X)$, there is a compact dag representation $G(t)$ for $t$ of minimal size, unique up to isomorphism.*
**Proof**
A dag of minimal size must be compact. [ ]

Notice that the complete argument given above makes no use of acyclicity. The last corollary would thus apply to cyclic terms (as used in functional programming).

We will now consider the computation of the unique compact dag, through merging of equivalent nodes.

### 3.3.3 Existing algorithms

As was shown above, dag compaction is equivalent to the minimisation of acyclic DFA and so algorithms for this problem apply directly to it. Quadratic-time

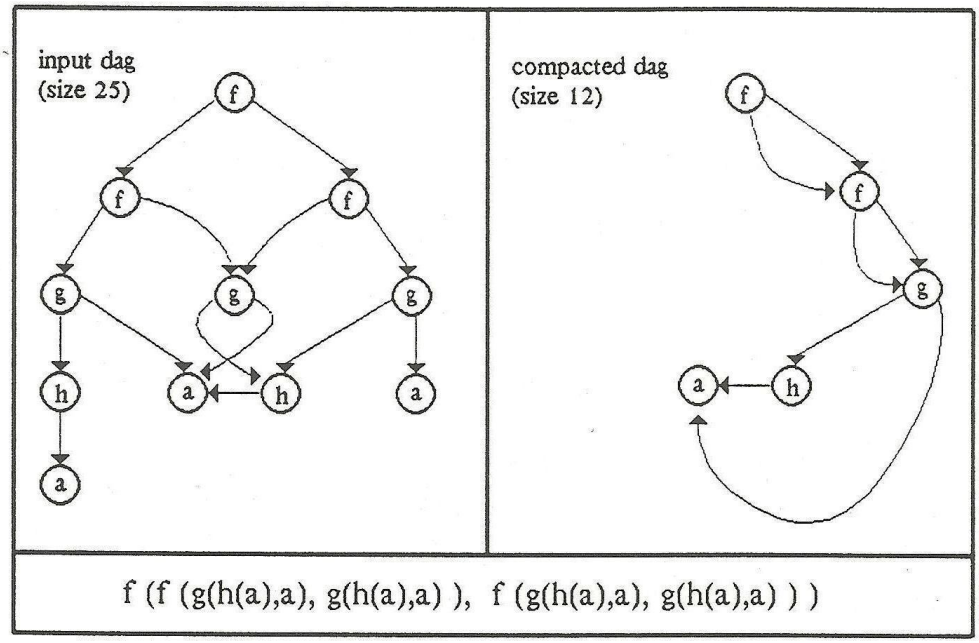


Figure 3.5: A term dag and its compact equivalent.

algorithms to minimise DFA have been given by Hopcroft and Ullman [25], and Brauer [9]. Minimisation of DFA also reduces to the *k-functions coarsest partition problem* and Aho, Hopcroft and Ullman [1] described an algorithm for this problem of complexity $O(n \log n)$, where $n$ is the number of states in the automaton or the number of nodes in the dag. Notice that for our purpose, and unlike the case of general graphs, the number of edges and the number of nodes in a term dag can only be a constant factor apart. The number of nodes is therefore a good measure for the size of the dag.

The dag compaction problem itself is known as the *elimination of common subexpressions* in compiler construction, where it is sometimes solved by methods based on hashing. The resulting algorithms run in expected linear time but are quadratic in the worst case.

Of those mentioned so far, only the Aho-Hopcroft-Ullman algorithm has a worst-case complexity which makes it suitable for very large terms. However, it does not take advantage of acyclicity and so the question arises as to whether there exists a linear-time sequential algorithm for (acyclic) dag compaction. The answer is unknown to us, but there is evidence that it is negative. This evidence follows consideration of yet another related problem, namely computing the congruence closure of an equivalence relation.

Compaction reduces to a special case of congruence closure for which a linear algorithm exists. Unfortunately the reduction involves preprocessing which, at best, requires either sorting or a UNION-FIND calculation. We now expand on this, and argue for the use of a systolic (linear-time) algorithm.

**Definition 14** *Congruence closure*
**Input** *An ordered directed graph $G$ and an equivalence relation $C$ on $V(G)$.*
**Output** *The congruence closure $C^*$ of $C$.*
*$C^*$ is the finest equivalence relation on $V(G)$ which satisfies the following property for any two nodes $v, w \in V(G)$:*
*Let $v$ and $w$ have successors $(v_1, \ldots, v_k)$ and $(w_1, \ldots, w_l)$ respectively. If $k = l$ and $\forall i : (1..k).\ v_i \equiv_{C^*} w_i$, then $v \equiv_{C^*} w$.* [ ]

The congruence closure problem is P-complete and its sequential nature is illustrated by the algorithms for solving it. They maintain a partition data structure, initialised to $C$, and repeatedly merge equivalence classes containing nodes whose corresponding sons lie in the same equivalence classes. Mergings are thus dependent on each other and only proceed bottom-up in the graph.

To apply such algorithms to dag compaction, define two dag nodes as $C^*$-equivalent if they are the roots of equal subterms. The initial equivalence relation $C$ should be set to:

$$v \equiv_C w \text{ iff } v = w \text{ or } v \text{ and } w \text{ are leaves with equal symbols.}$$

A congruence closure algorithm should then be applied with the minor difference that two nodes $v$ and $w$ be set $C^*$-equivalent when their corresponding sons are $C^*$-equivalent **and** symbol$(v)$ = symbol$(w)$. As postprocessing, it only remains to merge nodes equivalent under $C^*$ to obtain the minimal equivalent dag.

Downey, Sethi and Tarjan [13] have given two efficient algorithms for congruence closure. Both are based on the kind of incremental merging of classes described above. The first algorithm requires $O(n \log n)$ time and applies to any graph. The second algorithm applies to graphs $G$ such that $G/C^*$ is acyclic — a stronger restriction than the acyclicity of $G/C$ or $G$ alone— and terminates in linear time.

With a term dag G, nodes can only become $C^*$-equivalent and be merged if they are at the same level and so the compacted graph $G/C^*$ is indeed acyclic. Therefore the linear Downey-Sethi-Tarjan algorithm applies and it would seem that this gives a linear-time algorithm for dag compaction. However, its application requires the equivalence relation $C$ on leaves; i.e. finding which leaves have the same symbols. Clearly, sorting the leaf nodes by their symbol values is sufficient and the question then becomes whether or not this sorting can be done in linear time with respect to $n$, the size of $G$. If not then preprocessing will raise the complexity of the whole algorithm.

First of all, there could be $\Omega(n)$ leaves, and sorting them by the usual exchange-sort algorithms requires $\Omega(n \log n)$ time which is no better than the Aho-Hopcroft-Ullman algorithm for the coarsest-partition problem. Two possible solutions are to use a bucket sort [1] or a systolic sort algorithm [11].

Bucket sorting requires time $\Theta(n + m)$, where $m$ is the number of possible distinct symbols. It uses an array for direct access to $m$ lists, and so is only efficient if a bound on $m$ is known in advance. This is conceivable if for example,

the symbols were strings of length at most $m_0$ on an alphabet of 256 characters. In this case the lexicographic variant of bucket sorting [1] can sort the symbols in $O(256(n+m_0)) = O(256n) = O(n)$ time. However this restriction on the length of symbol strings could be artificial in the context of distributed processes, and we will look for another solution.

Another possibility is that the set of symbols occurring as leaves is already available through a 'symbol table'. Bucket sorting is then possible in time $O(n+m)$ where $m$ is the size of the symbol table. Notice however that the symbol table may be unnecessarily large, since only a fraction of its symbols could occur in a given term. Precomputation of the symbol table could be costly itself and there is no guarantee that the table would be globally available in a distributed environment. Assuming the existence of a symbol table is therefore a debatable choice to make, and we will favour a more general solution.

A simple linear-time solution is to use an array of processes to sort the leaves. This method is completely independent of any other processing but at the expense of $\Theta(n)$ processes. Instead of being used only as preprocessing to the Downey-Sethi-Tarjan algorithm, it can also be extended to an algorithm for the whole dag compaction problem. The next subsection develops this idea into a self-contained systolic algorithm for dag compaction which requires a linear number of processes, runs in linear-time and is simpler than the sequential methods of quasi-linear time complexity described above. Another advantage over the sequential Downey-Sethi-Tarjan algorithm is the use of simpler data structures which together with pipelining could provide significant acceleration when implemented.

### 3.3.4 A systolic algorithm

The systolic algorithm is based on bottom-up comparison of all pairs of nodes. The dag is first sorted into levels and a list of packets created. There is one packet for each node and the list is in non-decreasing order of levels. As a packet flows through the array of processes, it is compared with every other packet. Each packet maintains the address of the node it represents, the address of an equivalent node and addresses of nodes equivalent to its sons in the graph.

Let now $G$ be the input dag to be compacted and
$k = \max\{\text{arity}(v) | v \in V(G)\}$
$n_0 = |V(G)|$
$n = |V(G)| + |E(G)|$.
Notice that $n_0 \leq n \leq (k+1)n_0$ and so $n_0 = \Theta(n)$.

After or during level sorting of $G$, construct for each node a packet for input in the systolic array. The packet is initialised with the information contained in the node's record and has the following form:
**packet**
$A_i, E_i$: Address;
arity$_i$: (1..k);
symbol$_i$: $F \cup X$;
son$_i$: **array** [1..k] **of** Address;

**endpacket**.

Both $A_i$ and $E_i$ are set to the address of this node in memory. The other fields take their values from the corresponding node record fields. The algorithm only updates the values of $E_i$ and $\text{son}_i$.

From now on let $P_i$ be the packet associated with the $i^{th}$ node in level sort order. In other words $P_1, P_2, \ldots, P_{n_0}$ is the concatenation of the lists in *levset* (see earlier section on level sorting), so that whenever $i < j$ the level of node $i$ is no higher than the level of node $j$. Notice that we make no assumptions about the ordering of addresses.

The algorithm folds the list of packets over itself so that every packet meets every other. When $P_i$ and $P_j$ meet in a given process, this process updates $E_i$, $E_j$, $\text{son}_i$ and $\text{son}_j$.

This type of systolic algorithm is often called an 'all pairs' calculation or 'fold over' or 'one side meetings'. Cosnard and Tchuente [11] describe several possible communication designs for it. For our purposes they are all equivalent and the data-flow illustrated in Figure 3.6 (for $n_0 = 5$) is sufficient.

There are $n_0$ processes linked in a linear chain and each process can store two packets. The stream of packets is input at one end by increasing order of index (non-decreasing order of level). After all meetings have occurred the packets are output at the other end of the array. With simple modifications, two different streams of length $n_0$ or less can be processed one behind the other in the array, or one stream can be processed on an array of length $\lceil n_0/2 \rceil$.

To describe and analyse the algorithm it is simpler to abstract from its communication structure. This is easily done by observing that at systolic 'beat' $t$ (where $t \in (2\,..\,2n_0 - 2)$) there are simultaneous meetings of all pairs $(P_i, P_j)$ such that $i + j = t + 1$. The parallel composition of the $n_0$ processes therefore behaves like the following process.

**for** t=2 **to** $2n_0 - 2$ **do**
    **parallel** $(i, j) \in (1..n_0)^2 \mid i + j = t + 1$
        update$(P_i, P_j)$.

The processing step updates $P_i$ so that $E_i$ equals the least possible address of a node equivalent to $A_i$ and for $r \in (1..\text{arity}_i)$, $\text{son}_i[r]$ takes the least possible address of a node equivalent to the $r^{th}$ son of $A_i$. Packet $P_j$ is updated in the same way. Between two addresses for equivalent nodes, the smaller address always overwrites the larger one. In this fashion each equivalence class is eventually compacted to the representative node with least address. The choice of the least addressed node as representative decreases the 'scattering' of node records for the compacted dag. This can be an advantage for auxiliary operations of memory management [23].

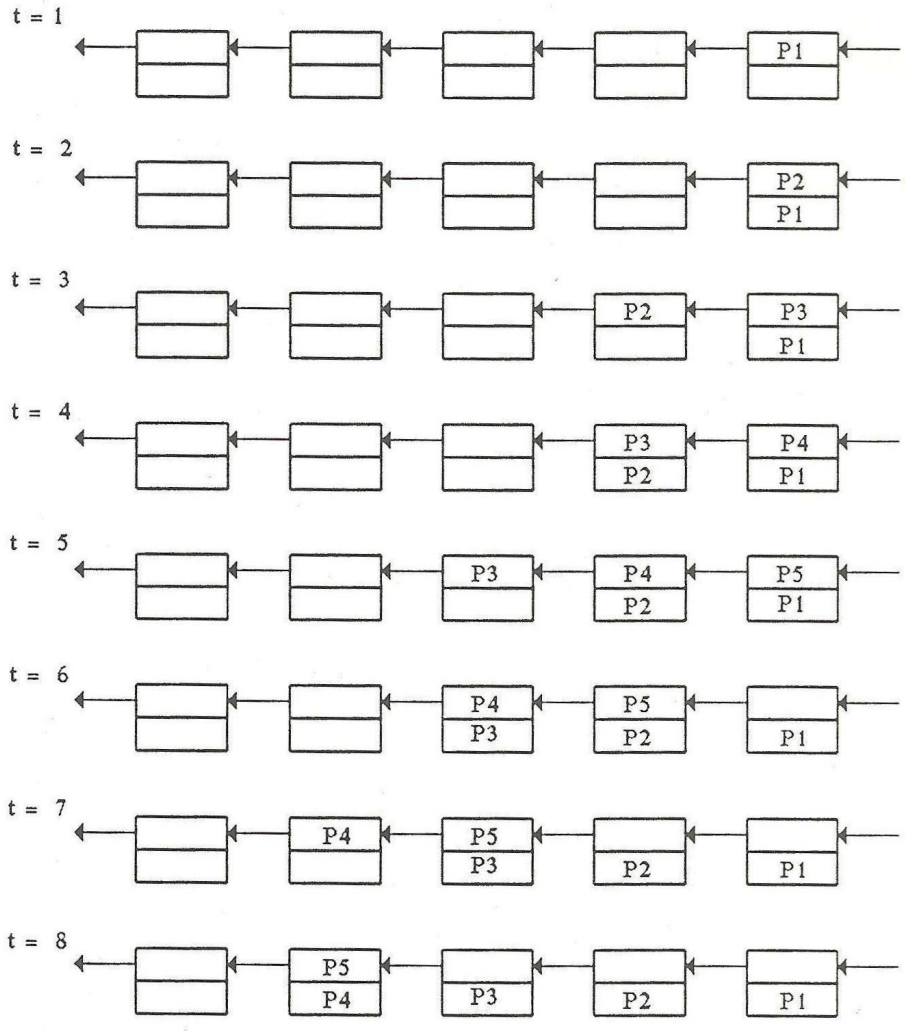


Figure 3.6: Dataflow for DC.

We will now use the notation $A_x \equiv A_y$ to mean that the node at address $A_x$ is equivalent to the node at address $A_y$, i.e. equal subterms are rooted there. Also, let $s_i[r]$ denote the address of the $r^{th}$ son of node $A_i$, i.e. the initial value of $\text{son}_i[r]$.

**Algorithm 6** DC: dag compaction.
**Input** List of packets $P_1, P_2, \ldots, P_{n_0}$ for the nodes of term dag G.

$$P_i = [A_i, E_i, \text{arity}_i, \text{symbol}_i, \text{son}_i]$$

**Output** List of packets $P'_1, P'_2, \ldots, P'_{n_0}$ such that $\{P'_i | i \in (1..n_0), A_i = E_i\}$ is a representation for the compaction of G.

**Procedure**

```
1. for t=2 to 2n_0 − 2 do
2.     parallel (i, j) ∈ (1..n_0)^2 | i + j = t + 1
3.         if arity_i = arity_j
4.         and symbol_i = symbol_j
5.         and ∀r : (1..arity_i). son_i[r] = son_j[r]
6.         then { A_i ≡ A_j }
7.                 E_i:= min(E_i, A_j)
8.                 E_j:= min(E_j, A_i)
9.         else { A_i ≢ A_j, update the son values }
10.                for r=1 to arity_i do
11.                    if son_i[r] = A_j then son_i[r]:=E_j
12.                for r=1 to arity_j do
13.                    if son_j[r] = A_i then son_j[r]:=E_i [ ]
```

Recall that $k$ is the maximum arity of function symbols in the term. Every systolic beat involves the communication of packets of size $k + 4$ and then execution of lines 3–13. The tests in lines 3–5 take $k + 2$ elementary operations, lines 7 and 8 take two operations and lines 10–13 take at most $4k$ operations. Each process can therefore execute synchronously in about $7k$ operations and the whole algorithm requires $(2n_0)(7k)$ or $14kn_0$ operations, disregarding unloading. This is a conservative bound and an implementation could certainly improve on the constant factors

For example, it is possible to overlap lines 3–5 with the packet input itself. This is done in $k + 4$ steps. The most time-consuming branch of the loop is then lines 10–13. It is impossible (because of acyclicity) to find both $\text{son}_i[r_1] = A_j$ *and* $\text{son}_j[r_2] = A_i$ for any $r_1, r_2$. Therefore a worst-case execution will do for $r = 1$ to $r = \text{arity}_i$ two instructions (**if** and :=) but only one for each pass through line 13. Thus lines 10–13 would take $3k$ operations in all. The total loop time is then $4k + 4$ for one loop, or about $8kn_0 +$ unloading for all of DC.

Correctness of DC is shown by preservation of an appropriate invariant on the values of $E_i$ and $\text{son}_i$.

**Lemma 7** *In the execution of DC, when packet $P_i$ meets packet $P_j$, $P_i$ has met with $P_1, \ldots, P_{j-1}$ and $P_j$ has met with $P_1, \ldots, P_{i-1}$.*
**Proof**
$P_i$ meets $P_j$ at time $i + j - 1$ and so meets any $P_{j'}$ with $j' < j$ at time $i + j' - 1$, earlier than $P_j$. The case of $P_j$ is symmetric [ ]

An indirect consequence of Lemma 7 and the key to the algorithm's correctness is that, when two function nodes of the same level are compared, their son arrays contain equal values if and only if the nodes are equivalent. This is where acyclicity is essential.

**Lemma 8** *After every execution of line 13 in DC, the following invariant holds.*

$$\forall (i,j) : (1..n_0)^2.\ i + j = t + 1 \Rightarrow INV(i,j)$$

*where INV(i,j) is the conjunction of the two conditions:*

1. $E_i = \min(A_i, \min\{A_x | x \in (1..j), A_x \equiv A_i\})$
2. $\forall r : (1..arity_i).$
   $son_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & if\ \ s_i[r] \in \{A_1, \ldots, A_j\} \\ s_i[r] & otherwise \end{array} \right]$

**Proof**
See Appendix A. [ ]

**Proposition 10** *After execution of DC, the set of packets $P_i$ for which $E_i = A_i$ is a pointer structure for the unique compaction of $G$.*
**Proof**
Consider the contents of $P_i$ for $i < n_0$ at $t = 2n_0 - 1$ (the case $i = n_0$ is verified in a similar way). Before termination, the last packet to meet $P_i$ was $P_{n_0}$ (by Lemma 7) and so $P_i$ satisfies INV$(i, n_0)$ by Lemma 8. Therefore:

$$\begin{aligned} E_i &= \min(A_i, \min\{A_x | x \in (1..n_0), A_x \equiv A_i\}) \\ &= \min(A_i, \min\{A_x | A_x \equiv A_i\}) \\ &= \min\{A_x | A_x \equiv A_i\} \quad (*) \end{aligned}$$

Also, because trivially $s_i[r] \in \{A_1, \ldots, A_{n_0}\}$ for all $r$, we have:

$$\begin{aligned} \text{son}_i[r] &= \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & \text{if } s_i[r] \in \{A_1, \ldots, A_{n_0}\} \\ s_i[r] & \text{otherwise} \end{array} \right] \\ &= \min\{A_y | A_y \equiv s_i[r]\}\ (**) \end{aligned}$$

It follows from (*) that

$$E_i = A_i \ \Rightarrow \ \ A_i = \min\{A_x | A_x \equiv A_i\}$$

and in this case node $A_i$ is the representative of its equivalence class. By (**), the final value of $son_i[r]$ ensures that this pointer is directed at the representative of the equivalence class containing $s_i[r]$. The resulting graph is therefore equivalent to $G$, with all equivalent nodes (and only those) merged. Therefore it is the unique compact dag equivalent to $G$. [ ]

From the output $P_1, \ldots, P_{n_0}$ of DC, the packets $P_i$ for which $E_i \neq A_i$ are discarded and the others are returned to memory to replace the records with which they were initialised. Records corresponding to discarded packets are no longer referred to by other node records and are then available for garbage collection.

### 3.3.5 Compaction of AC-terms

This subsection outlines changes to the DC algorithm which enable compaction of terms containing associative-commutative function symbols.

As was the case with level sorting, the correct treatment of nodes having function symbols from $F_{AC}$ forces changes to the compaction algorithm for free terms. Simply treating such nodes as functions of variable arity is not sufficient, because of their commutativity.

Conceptually, extending a bottom-up procedure like DC to deal with commutativity is straightforward, and we will illustrate this with an example. Suppose the term $g(f[a,b,c], f[b,c,a])$, where $g \in (F_2 - F_{AC})$, $f \in F_{AC}$ and $a, b, c \in F_0$, is represented as a tree $T$ and given for compaction. Then $T$ could be represented by a list of 9 packets sorted by levels, such as the following:

| **address** | **symbol** | **type** | **sons** |
|---|---|---|---|
| 9 | g | free | (6,4) |
| 6 | f | AC | [1,2,5] |
| 4 | f | AC | [8,3,7] |
| 1 | a | free | |
| 2 | b | free | |
| 5 | c | free | |
| 8 | b | free | |
| 3 | c | free | |
| 7 | a | free | |

If DC was applied directly, it would first compare all the leaves and so compact level 0 to the three packets with addresses 1,2 and 3 respectively. The two packets labeled with $f$ would then be compared with all the leaf packets and have their son's addresses updated accordingly. This transformation is illustrated by parts 1 and 2 of figure 3.7. The three packets for internal nodes would then have values:

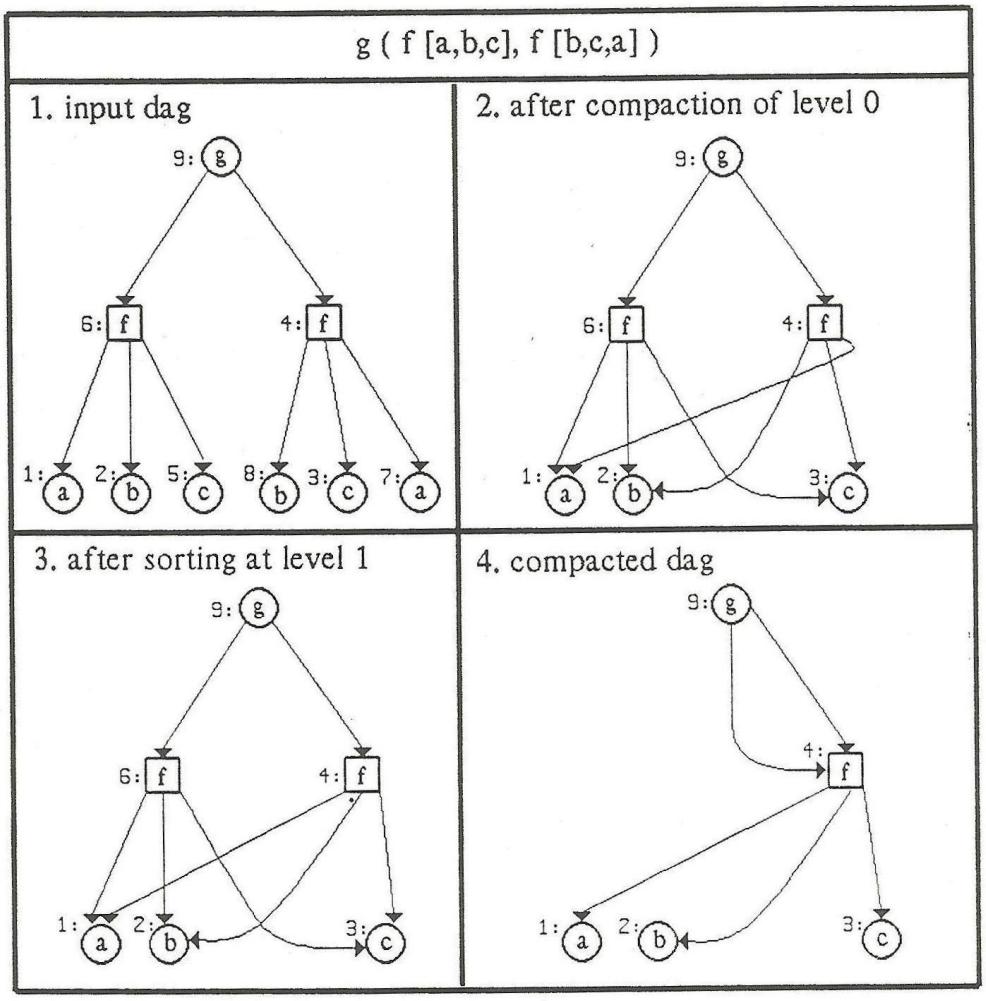


Figure 3.7: Compaction of an AC-term (the AC-functions are in boxes).

| **address** | **symbol** | **type** | **sons** |
|---|---|---|---|
| 9 | g | free | (6,4) |
| 6 | f | AC | [1,2,3] |
| 4 | f | AC | [2,3,1] |

At this point, the two nodes labeled with $f$ are equivalent if and only if they have the same *multisets* of sons addresses. If the two corresponding packets are to be correctly set as equivalent, their lists of sons should first be sorted by addresses — there would be no duplicates — and then tested for equality (see parts 3 and 4 of figure 3.7).

In other words, distinct subterms correspond to distinct addresses after compaction of lower levels and so distinct terms (with roots from $F_{AC}$) at upper levels correspond to distinct multisets of addresses. It is not difficult (but tedious) to verify that this will compact the tree to its unique equivalent compact

dag (and that uniqueness indeed holds for AC-terms modulo permutations of edges leaving AC-nodes). To avoid giving arguments very similar to those of earlier subsections, we will skip the details and hope the reader can reconstruct them.

There are however two problems with implementing this procedure with the same data-flow as DC. First, packets for AC-functions could be arbitrarily large because of their sons lists and this would considerably slow down the flow of packets (proportionally to the size of the largest packet). Also it would force sorting of these arbitrarily large lists within a single process, again an unreasonable change.

The simplest solution is to expand each node for an AC-function into as many packets as it has sons. There would then be three different types of packets:

1. free function nodes
2. AC-function nodes (headers)
3. AC-function sons addresses (pointers).

In our example, the three nodes at levels 1 and 2 could be represented by the following list.

| **packet** | **address** | **symbol** | **type** | **sons** |
|---|---|---|---|---|
| $P_{15}$ | 9 | g | free | (6,4) |
| $P_{14}$ | 6 | f | AC header | |
| $P_{13}$ | 6 | | AC pointer | 1 |
| $P_{12}$ | 6 | | AC pointer | 2 |
| $P_{11}$ | 6 | | AC pointer | 3 |
| $P_{10}$ | 4 | f | AC header | |
| $P_9$ | 4 | | AC pointer | 1 |
| $P_8$ | 4 | | AC pointer | 3 |
| $P_7$ | 4 | | AC pointer | 2 |

Consider then the meeting of packets $P_7$ to $P_{10}$ with packets $P_{11}$ to $P_{14}$, using the same data-flow as before. Packets $P_7$, $P_8$, $P_9$ meet $P_{11}$, $P_{12}$, $P_{13}$ before the headers $P_{10}$ and $P_{14}$ meet.

Without any sorting, it is therefore possible to test the two multisets of packets $[P_7, P_8, P_9]$ and $[P_{11}, P_{12}, P_{13}]$ for equality and inform both header packets of the result before the header packets meet each other. This can be done e.g. by making $P_7$ and $P_{12}$ temporarily 'marked' when they meet, because they point to the same address and neither was already marked before the meeting. Later, $P_7$ will meet the header packet $P_{14}$ which then will check for an 'unmarked' packet. Similarly, $P_{12}$ will be checked by $P_{10}$. When checking a sons address packet, the header packet should reset it to 'unmarked' for future use. If both header packets have only recorded marked packets, then their multisets of sons were equal, and they should be set as equivalent nodes (provided they have the same AC-function symbol) in the same way as in DC.

We conclude that DC can be augmented in a simple way to deal with AC-functions. The resulting algorithm uses at most $2n_0$ packets and as many processes, where $n_0$ is the number of nodes in a dag representation of its input term.

# Chapter 4

# Free unification

This chapter deals with parallel solutions to the free unification problem.

An algorithm called TM is presented for the term matching problem. TM is pipelined and is designed to be repeatedly loaded with pairs of terms encoded as the data structures of Chapter 3. It is argued that TM executes fewer operations on term matching problems than the more general Paterson-Wegman algorithm. Thus TM increases the execution speed of those applications where full unification is not needed. Moreover, there exists a theoretical result to the effect that many logic programs can be executed without the use of full unification, simply with term matching [38].

We then introduce a method for obtaining a sublinear-time algorithm for TMATCH on a mesh of processes, assuming that terms are stored as trees. This method is only of practical value if the environment where TMATCH is required either maintains terms in a distributed form (so that no loading is necessary) or allows parallel input of $\Omega(\sqrt{n}/\log^2 n)$ bits per time step. The last section presents justification for the absence of any sublinear algorithm for UNIF.

The first section considers the communication requirements for general-purpose free unification algorithms, to justify the use of a one-dimensional dataflow in TM.

## 4.1 Input as a bottleneck

Our overall goal is to design algorithms which are as fast and simple as possible, but preferably independent of system details. Because no assumption is made about the environment in which unification is to operate, the physical distribution of input terms is *a priori* unknown. Term dags could be conveniently distributed over many neighbouring processes, or otherwise stored entirely in a single process. In the former case parallel input is possible, but in the latter input speed is limited by sequential communication. It is therefore reasonable to give our unification algorithms the simplest possible interface with their environment: sequential input of terms and sequential output of results. Parallel input would be useless

in the worst case described above, and parallel output could not operate at a faster rate than the corresponding input.

Another important design constraint comes from the observation that unification depends on every node of the input dags. As mentioned earlier, the unification decision problem cannot be approximated or expressed as an optimisation problem. Its answer must be an exact 'yes' or 'no'. Probabilistic outputs, with only a high probability of being correct, are not acceptable for its solution either. Theorem-proving systems and other similar applications need to rely on a deterministic unification algorithm if they are to serve their purpose.

Trivially, a deterministic algorithm for UNIF or TMATCH must solve the equality problem EQUAL as a special case when the terms include no variables. It is clear that equality potentially depends on every node in the input graphs. Therefore a correct unification algorithm must — in the worst case — examine its complete input before giving its decision about unifiability (equality).

The above arguments imply that a unification algorithm which sequentially loads every pair of terms before processing them has complexity $\Omega(n)$, where $n$ is the input size. Given the existence of a linear-time sequential algorithm for UNIF [43][12], input time places a serious limit on the usefulness of parallelism for UNIF and TMATCH. Nevertheless, there are several practical ways to reduce the overheads implicit in the linear complexity, and this is the subject of the following sections. For example detecting non-unifiable inputs often does not require examination of all $n$ nodes. Another possible improvement is to use pipelining to overlap successive executions of an algorithm.

## 4.2 TM: solving TMATCH via dag compaction

This section describes an algorithm called TM for the TMATCH problem. Its overall structure is a double pipeline with the dag compaction algorithm DC as a component. The first subsection gives justification for the design of TM. The next sections describe the algorithm and prove its correctness. The last subsection gives an estimate of its execution time on several extreme instances of TMATCH. This is used in a comparison with the Paterson-Wegman algorithm.

### 4.2.1 Design decisions

By definition the free term matching problem TMATCH consists in deciding, for free terms $s, t$ whether there exists a substitution $\sigma$ such that $\sigma(s) = t$. If $t$ does not contain any variables, as is often the case in applications, then $\sigma(t) = t$ and $\sigma(s) = t$ if and only if $\sigma(s) = \sigma(t)$, so that TMATCH becomes a special case of UNIF.

If $t$ did contain a variable $x_0$, then it could only match occurrences of the same variable in $s$. In that case occurrences of $x_0$ in $s$ could not be bound to any constant subterm of $t$ and would therefore behave like constants. Variables

which occur in both terms are thus irrelevant to the problem of term matching.

From now on we will use the simpler definition of TMATCH: given $s, t$ where $t \in T(F, \emptyset)$, decide whether $s$ and $t$ are unifiable, and output a unifier if they are. For the purpose of this chapter, we will call $s$ the *variable term* and $t$ the *constant term*. The variable term in a term matching problem is often referred to as the *pattern*.

Among special cases of UNIF, TMATCH is most interesting because it often occurs in applications and it is also computationally easier than the full problem. As explained in Chapter 2, UNIF is P-complete and TMATCH is solvable in NC. However, the 'sequential input bottleneck' does not allow the sublinear execution times of an NC algorithm. Nevertheless, the low complexity of TMATCH allows us to design an algorithm which is simpler and faster than a general unification algorithm. The next subsections will expand on this point.

The design of TM is targeted at applications like term rewriting systems and logic programming interpreters where a large number of terms are successively processed for unification. In the case of logic programming, Maluszynski and Komorowski have discovered that a whole class of logic programs may be executed with only a term matching algorithm, avoiding the use of full unification. Moreover, this class of programs can be recognised before execution, as a form of precompilation [38]. In a more general situation, it is always possible to dynamically keep track of which terms contain variables, so as to invoke a term matching algorithm whenever one of two terms to be unified contains none. In both cases described above, a fast TM algorithm would provide accelerations over usual unification algorithms.

It is therefore preferable to concentrate design efforts (for TM) on the reduction of processing overheads and to overlap successive executions by a pipelined architecture. The resulting algorithms will be most useful in applications where many pairs of terms have to be matched/unified in sequence. In addition, TM has a strictly linear-time complexity and so is able to match very large terms, represented with structure sharing. Another advantage of our algorithms is that, unlike some other attempts at practical solutions [40], they do not require terms to be represented as trees or to be linear.

### 4.2.2 Algorithm description

The computation of free unification is best understood as a sequence of DEC and MER operations (see Chapter 3). DEC involves recursive decomposition of the initial unification problem into unification of corresponding subterms. The MER operation consists in unifying subterms which have been bound (unified) to the same variable. Unification fails if DEC encounters two unified subterms with unequal function symbols at their roots (clash), or if a cyclic substitution is created (cycle).

A simple scheme for unification is therefore given along the lines of Martelli and Montanari [39] as follows.

**Algorithm 7** *(DEC;MER)**

**While** *no clash or cycle, and pairs of subterms remain to unify* **do**
  *Apply DEC to all pairs of subterms to be unified.*
  *Apply MER to all variables.*
**EndWhile**
*Test the resulting bindings for acyclicity.*
**If** *acyclic and no clash* **then** *output unifier.* [ ]

In its simplest possible implementation, this algorithm terminates in quadratic time, after unifying every pair of subterms. If however applied to term matching, it only requires a single application of MER, and this provides a good characterisation of what an algorithm for TMATCH is.

Let $(s, t)$ be an instance of TMATCH where $s$ is the variable term, and suppose $s$ and $t$ are unifiable. Then the first application of DEC creates bindings from the variables of $s$ to subterms of $t$. Let $x$ be a variable which occurs more than once in $s$. Then DEC will possibly identify more than one subterm of $t$ as match for $x$. This occurs for example in

$$\text{TMATCH}\{f(x, x); f(g(a), g(a))\}.$$

Application of MER will create new unification tasks: to unify together all subterms matched with $x$. In other words, all subterms unified with $x$ must be unified together, and similarly for every other variable. However, being constant subterms, unifying them only requires testing their equality. It follows that the second application of DEC will only be a test for the equality of specified sets of subterms (see Figure 4.1). There are then no more subterms to unify and Algorithm 7 terminates.

An important observation is that substitutions can never be cyclic, as the bindings all point to constant subterms in $t$. There is therefore no need for an acyclicity test in TM.

In summary, a TMATCH algorithm can be designed as a procedure which:

1. decomposes the initial matching problem into a set of necessary variable bindings, one for each variable occurrence. (DEC)
2. detects any symbol clashes during step 1.
3. ensures that instances of a given variable are bound to equal subterms (MER).
4. outputs the complete unifying substitution.

**Algorithm 8** *Outline of TM.*

*Output* $:= \emptyset$

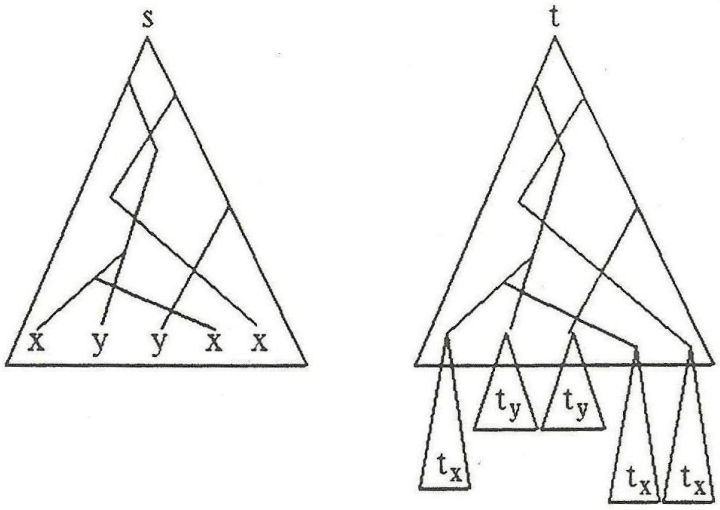


Figure 4.1: Equal subterms must match with occurrences of a same variable.

*Apply DEC to the roots of* $s, t$ *while testing for symbol clashes.*
**If** *no clashes* **then**
  **For** $x \in X$, $x$ *bound to* $\{t_x^{(1)}, \ldots, t_x^{(k)}\}$ *by DEC,* **do**
    **If** $t_x^{(1)} = t_x^{(2)} = \ldots = t_x^{(k)}$ **then**
      *Output:=* $Output \cup \{x \mapsto t_x^{(1)}\}$
    **Else**
      *write('not unifiable'); stop*
*write('unifier = ' Output)* []

Observe that Algorithm 8 is simplest if the terms involved are compact. With compact input terms, a variable occurs as a single node and so needs only be bound once during DEC. Also, the test for equality of constant subterms is trivial: two subterms are equal if and only if they are the same. Accordingly, dag compaction is included as the first phase of TM. In certain environments, this could have the advantage of avoiding a separate application of DC.

Assuming compact input terms, a depth-first traversal of the variable term gives an efficient implementation of DEC if this traversal is used to 'drive' a simultaneous (possibly incomplete) traversal of the constant term. Traversal of the variable term is non-redundant, and the constant term is visited in exactly the same order.

The communication structure of TM is illustrated by Figure 4.2.

The variable term is first input to $DC_1$ from the process which initially contains it. $DC_1$ is a copy of the DC algorithm and reduces the variable term dag to its compact form. Similarly, the constant term in compacted by $DC_2$.

Outputs from $DC_1$ and $DC_2$ are driven into processes $DFS_1$ and $DFS_2$ re-

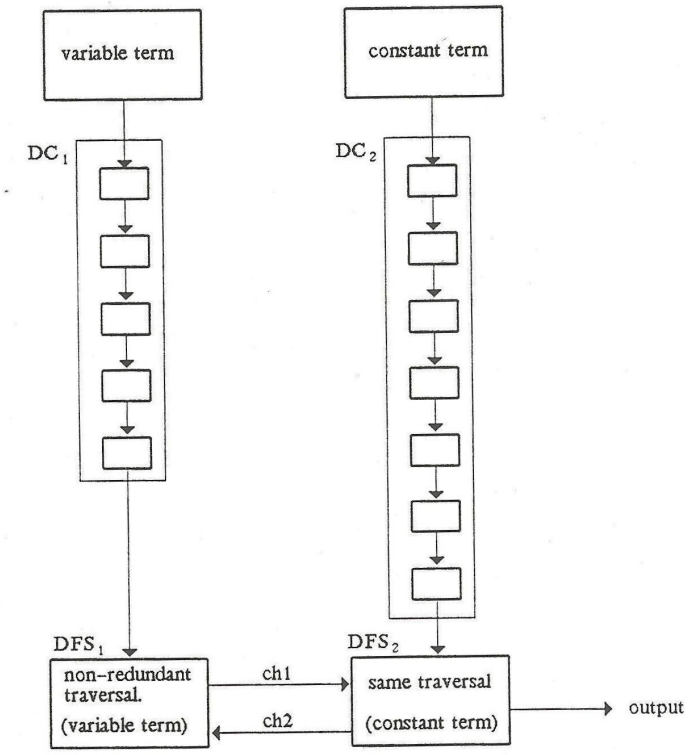


Figure 4.2: Communication structure for algorithm TM

spectively.

$DFS_1$ performs a depth-first search of the compacted variable term. This traversal takes time proportional to the reduced term size. $DFS_2$ is synchronised with $DFS_1$ to perform the same traversal but on the constant term. Both processes communicate and synchronise through channels ch1 and ch2 (Figure 4.2).

As execution traces identical paths in both terms, the necessary bindings can be computed, clashes can be detected and the equality test for subterms is performed on the way.

The details of TM are given below and in Figure 4.3 and Figure 4.4. Explicit communications are expressed in occam-like syntax [27]: *channel ! message* for an output or *channel ? variable* for an input.

**Algorithm 9** *TM*

**Input** *Variable term $s$, constant term $t$. Both free terms.*
**Output** *TMATCH($s, t$). The unifier is idempotent; (not necessarily most general). The output comes as a set of pairs ($x \mapsto v_F$) where $x$ is a variable occurring in $s$ and $v_F$ is the root of the subterm to which $x$ is bound in $t$.*

**Process** *TM*
  **Parallel**
    **Process**
      *Sort $s$ by levels.*
      *Compact $s$ ($DC_1$).*
      *$DFS_1$.*
    **Process**
      *Sort $t$ by levels.*
      *Compact $t$ ($DC_2$).*
      *$DFS_2$.*
**EndProcess**

**Process** *$DFS_1$.*
**Data Structures**
  *Term dag $(V(s), E(s))$ for $s$.*
  *Visited: $V(s) \rightarrow$ Boolean.*
  *Match: $V(s) \rightarrow V(t)$.*
**Procedure**
  *Input node records for $s$, from $DC_1$.*
  *Initialise Visited($v$) to False and Match($v$) to undefined, $\forall v \in V(s)$.*
  *Search$_1$(root($s$)).*
**EndProcess.**

**Process** *$DFS_2$.*
**Data Structures**

**Procedure** Search$_1(v)$
**If** Visited$(v)$ **then** (already occurred, test equality of matching subterms)
  ch1 ! position-please
  ch2 ? position
  **If** position $\neq$ Match$(v)$ *then*
    ch1 ! clash
    exit DFS$_1$
  **Else**
    ch1 ! no-clash
**Else** (first visit to this node)
  Visited$(v)$ := True
  **If** symbol$(v) \in X$ **then**
    ch1 ! symbol$(v)$
    ch2 ? Match$(v)$
  **Else** (function symbol)
    ch1 ! symbol$(v)$
    ch2 ? reply
    **If** reply = clash **then** exit DFS$_1$
    **Else**
      ch2 ? Match$(v)$
      **For** i:=1 **to** arity$(v)$ **do**
        Search$_1$(son$_i(v)$)
**EndProcedure**

Figure 4.3: Procedure Search$_1$: traversal of the variable term in TM.

*Term dag $(V(t), E(t))$ for $t$.*
*Bind: $X \to V(t)$.*
**Procedure**
*Input node records for $t$, from $DC_2$.*
*Initialise Bind$(x)$ to undefined, $\forall x \in X$.*
*Search$_2$(root$(t)$).*
**EndProcess.**[]

### 4.2.3 Correctness

**Proposition 11** *Given free terms $s, t$, TM reports failure if $s$ and $t$ are not unifiable and outputs a unifier for $s$ and $t$ otherwise.*
**Proof** From the correctness of algorithm DC, it follows that processes DFS$_1$ and DFS$_2$ input dag structures $G_s = (V(s), E(s))$ and $G_t = (V(t), E(t))$ which are the compact representations of $s$ and $t$ respectively. Both processes initialise their local tables and then perform the calls Search$_1$(root$(G_s)$) and Search$_2$(root$(G_t)$). We will now show that the joint execution of Search$_1$ and Search$_2$ is an implementation of Algorithm 8 and therefore solves TMATCH.

```
Procedure Search2(v)
ch1 ? message
If message = position-please then
  ch2 ! v
  ch1 ? reply
  If reply = clash then
    write('Not unifiable')
    exit DFS2
Else (message is a function or variable symbol)
  If message ∈ X then
    Bind(message) := v
    write(message ↦ v)
    ch2 ! v
  Else (message ∈ F)
    If message ≠ symbol(v) then
      ch2 ! clash
      write('Not unifiable')
      exit DFS2
    Else (no clash)
      ch2 ! no-clash
      ch2 ! v
      For i:=1 to arity(v) do
      Search2(son_i(v))
EndProcedure
```

Figure 4.4: Procedure $\text{Search}_2$: (incomplete) traversal of the constant term in TM.

Search$_1$ follows a depth-first search of $G_s$ and Search$_2$ traces the same path through $G_t$. The traversals are synchronised as they move down into one more recursive call or exit from one.

Thus, whenever Search$_1$ visits a node $v_s$, Search$_2$ visits a node $v_t$ such that there is a path-sequence (occurrence) $p$ in $O(s)$ (see Chapter 2 for definition) which is also in $O(t)$ and which leads both from the root of $G_s$ to $v_s$ and from the root of $G_t$ to $v_t$. It follows that the subterms rooted at $v_s$ and $v_t$ must be unified. Moreover, it is easily verified that all unification constraints directly derivable from DEC, are thus computed when Search$_1$ first visits a given node $v_s$. On the first call Search$_1(v_s)$, Search$_2$ is passed the value of symbol$(v_s)$ to test for a clash. We conclude that the first step in Algorithm 8 i.e. DEC, is implemented by the first visits made by Search$_1$ to nodes in $V(s)$.

It remains to verify the equality of subterms bound to a given variable $x$. By the compactness hypothesis, $x$ occurs as a single node in $G_s$. On its first visit to $x$, Search$_1$ binds $x$ to the corresponding $v_t \in V(t)$ by the instruction *Match(x):=* $v_t$. Any other occurrence of $x$ leads to a subsequent call Search$_1(x)$. Assume this call to be synchronised with the call Search$_2(v'_t)$. By the compactness of $G_t$, the subterms rooted at $v_t$ and $v'_t$ are equal if and only if $v_t = v'_t$, and this test is performed by Search$_1(x)$ for every new occurrence of $x$:

ch1 ! position-please
ch2 ? $v'_t$
etc.

Variable bindings are output once for each variable by Search$_2$ and so the **If** instruction in Algorithm 8 is correctly implemented as well. This concludes the proof. [ ]

Figure 4.5 illustrates the execution of TM.

### 4.2.4 The Paterson-Wegman algorithm

This subsection is an account of the Paterson-Wegman unification algorithm, to be used for comparing the efficiency of TM. Algorithm PW below is DeChampeaux's version of the Paterson-Wegman algorithm. It is written here in a notation consistent with our data structures for terms.

PW is restricted to inputs where variables are represented by single nodes. The original version of the algorithm required that lists of parent nodes be associated with each input node. This has been included in the present version as a step in the algorithm. This step involves a depth-first traversal of the input dag.

The terms are represented by dag pointer structures as defined in Chapter 3. There are also the following data structures:

Complete : $V \rightarrow$ Boolean

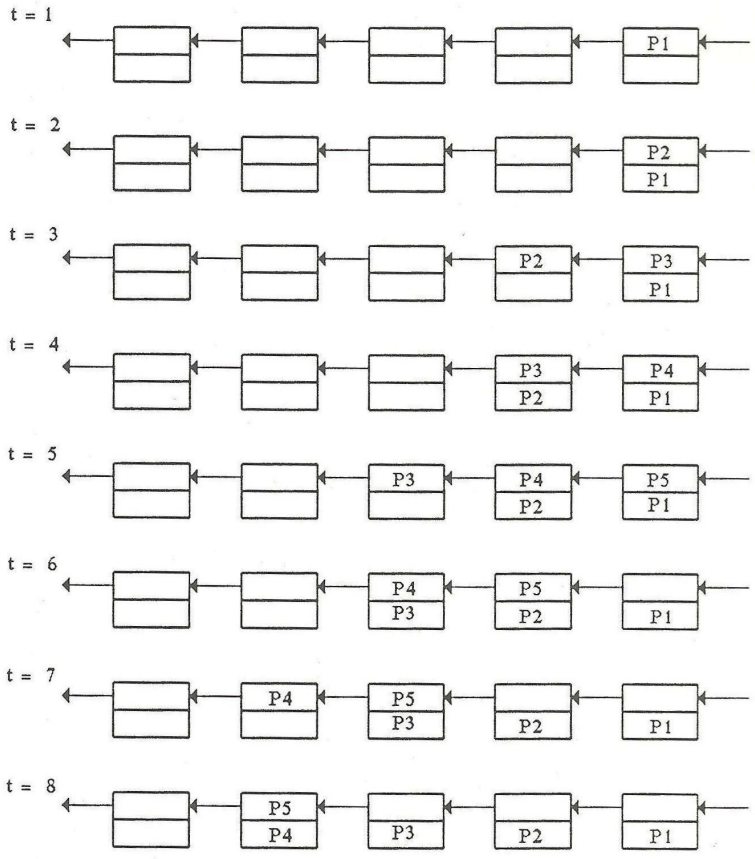


Figure 4.5: Execution of TM

Binding : $V \to V \cup \{\text{NIL}\}$
Parents : $V \to$ List of $V$
Links : Set of Pairs of $V$.

The flags Complete are used to make the algorithm non-redundant in its traversal of the dag. The function Binding defines, when restricted to the set of variable nodes, a unifier. Links is used to represent an equivalence relation which is *valid* in the sense of definition 10 and lemma 3 (Chapter 2). In other words, Links is an undirected graph whose connected components define classes of nodes where unified subterms are rooted.

The main part of the algorithm consists of a non-redundant traversal of the dag via recursive calls to procedure Finish. Whenever a node is visited, all its predecessors in the graph (parents, parents of parents, ...) are first processed by the Finish procedure, if they have not already been visited. Consider now a node $v$ when all its parents have been 'correctly unified' by previous calls to Finish. At this point Links would include all possible unification pairs containing $v$. The stack's contents then direct processing through all nodes $s$ unified with $v$. Each such node is bound to $v$ and the set of bindings is tested for acyclicity. If $s$ is a variable node, the binding $s \mapsto v$ is output as part of the unifier. If it is a function node, new constraints are added to Links accordingly, unless a clash is detected.

**Algorithm 10** *PW.*

**Input** *Two terms given as root nodes $r_1, r_2$ in a dag structure $(V, E)$. No variable must appear as more than one node, but the dags need not be compact.*

**Output** *A failure message if the terms are not unifiable. Otherwise a set of bindings defining a unifier. The unifier is not necessarily idempotent (see Chapter 2).*

**Begin**
Depth-first traversal, for $v \in V$ do:
  Initialise Complete($v$) to False and Binding(v) to NIL.
  Set Parents($v$) to the list of nodes for which $v$ is a son.
Links := { $\{r_1, r_2\}$ }
**While** $\exists v : V$. not Complete($v$) **and** symbol($v$) $\in F$ **do** Finish($v$).
**While** $\exists v : V$. not Complete($v$) **and** symbol($v$) $\in X$ **do** Finish($v$).
write('unified')
**End.**

**Procedure** Finish($v : V$)
**If** Binding($v$) $\neq$ NIL **then** write('failure: cycle'); stop.
Binding($v$) := $v$
Create new stack with operations Push and Pop.

```
Push(v)
While the stack is not empty do
  s := Pop
  If symbol(v), symbol(s) ∈ F and symbol(v) ≠ symbol(s) then
    write('failure: clash'); stop.
  For t ∈ Parents(s) do
    If not Complete(t) then Finish(t).
  For {s, t} ∈ Links do
    If not Complete(t) and t ≠ v then
      If Binding(t) = NIL then
        Binding(t) := v
        Push(t)
      ElseIf Binding(t) ≠ v then
        write('failure: cycle'); stop
  If s ≠ v then
    If symbol(s) ∈ X then write(s ↦ v)
    Else Links := Links ∪ { {son_j(s), son_j(v)} | j : (1..arity(v)) }
  Complete(s) := True
EndWhile
Complete(v) := True
EndProcedure []
```

For more details on PW, the reader should consult [12].

### 4.2.5 Efficiency

Having chosen PW as reference, we will now compare its efficiency with that of TM. Because both algorithms have complexity $O(n)$, it is necessary to consider the 'constant factors' involved in their execution. We will estimate the time taken by TM on the most unfavourable input, and compare this with the time taken by PW under the same assumptions.

**Input** Assume an input where the variable term has $n_1$ nodes and the constant node has $n_2$ nodes. Assume also that the maximal arity of functions is $k = 2$ and so the input size is $n \leq 3(n_1 + n_2)$, allowing for two edges per node.

**Timing of TM** Since $n_2 \geq n_1$, $\mathrm{DC}_2$ will take more time than $\mathrm{DC}_1$ and the execution of $\mathrm{DFS}_1 - \mathrm{DFS}_2$ will begin precisely when the first packet is output from $\mathrm{DC}_2$. Also, it will be sufficient to count the number of operations executed by $\mathrm{DFS}_1$ to complete the estimate of TM's running time.

**DC** Recall from Chapter 3 that DC can be implemented to take time $8kn_0 = 16n_0$ (exclusive of unloading) where $n_0$ is the number of node packets involved. The packets are of size $k + 4 = 6$.

**Level sorting and $\mathrm{DC}_2$** The constant term is first traversed for level sorting and input to TM. Consider any node in the constant term dag with two

outgoing edges. When control moves 'up' from the node, its level has been computed and it is input into $DC_2$. Before this output, control must visit the node and traverse both of its outgoing edges. There are thus $3n_2$ steps interleaved with the input to $DC_2$, which must be added to the overall time for $DC_2$. By the above estimate of DC, level sorting and $DC_2$ together require time

$$3n_2 + 16n_2 = 19n_2$$

at which point $DC_2$ is ready to begin unloading node packets. The packets are output by decreasing order of levels (root first).

**$DFS_1$ and $DFS_2$** Because both processes are synchronised, it is sufficient to estimate the time for $DFS_1$. Output from the DC processes comes by decreasing level order, which does not in general correspond to depth-first order (or even breadth-first order). There are thus time steps where $DFS_1$ is idle waiting to visit a certain node while $DC_1$ presents it with a different node. In other words, input from $DC_1$ is not perfectly overlapped with $Search_1$. In the worst case there is almost no overlap and $DFS_1$ must wait for most of the $n_1$ packets (of size 6) to be read before it begins. This adds time $6n_1$ to TM.

**$DFS_1$** It is also possible that $DC_1$ has been needlessly applied (the variable term being already compact) and so the dag traversed by $Search_1$ includes all $n_1$ nodes. Consider then the execution of $Search_1$ and assume without loss of generality that the dag is a tree (worst-case for TM). Assume also that TM is successful and so the search does not abort. For each of the $n_1$ nodes, the following 9 steps are executed (Figure 4.3) :

**If** Visited($v$), **Else**
Visited($v$) := True
**If** symbol($v$) $\in X$, **Else**
ch1 ! symbol($v$)
ch2 ? reply
**If** reply = clash, **Else**
ch2 ? Match($v$)
**For** i=1 **to** 2 **do**
recursive call.

Hence $9n_1$ steps are executed for $DFS_1$ itself.

**TM total** The overall count is therefore

$$19n_2 + 6n_1 + 9n_1 = 15n_1 + 19n_2$$

instructions in this worst case.

**Input to PW** The same assumptions are made about the input terms for PW. Both terms (of sizes $3n_1, 3n_2$ as above) are loaded in memory when PW begins execution. PW visits every node once either by a call to Finish or by a pass through the **While** loop of Finish($v$), where $v$ is either a descendent of the visited node or an equivalent (unified) node under Link.

**Preprocessing** As in the evaluation of TM, initialisations will be ignored. However, the precomputation of Parents is part of the algorithm and must be accounted. It requires a complete traversal: at least 3 operations per node and each edge is followed 'up' and 'down', i.e. at least

$$3(n_1 + n_2) + 2(2n_1 + 2n_2) = 7(n_1 + n_2).$$

**Tree traversal** It has been assumed that the terms are both trees and in compact form. Therefore no subterm is repeated and so the matching of subterms via Links (there are no variables in this extreme example) is one-to-one between the terms. Inspection of PW (Algorithm 10) then shows that all nodes $v$ in the first term are visited by Finish, pushed onto the corresponding stack, passed through the **While** loop as $s$, linked with a unique $t$ in the other term. Node $t$ is then passed through the **While** loop and execution exits from Finish. There are no calls to Finish for parent nodes because the traversal is assumed to be strictly top-down.

**Instruction count for** $v$ For $v$ visited by Finish, we will consider the **While** loop as associated with the linked node $t$. There are 6 instructions executed specifically for node $v$:

test for **not** Complete($v$), (success)
**If** Binding($v$) $\neq$ NIL, (false)
Binding($v$):= $v$
Create new stack.
Push($v$).
Complete($v$) := True.

**While loop instructions** Node $v$ is then removed from the stack as $s$. It is assumed that Link contains one pair $\{s, t\}$. The following instructions are executed:

0 **While** stack is not empty **do**
1 $s$ := Pop
2 **If** symbol($v$), symbol($s$) $\in F$
3 **and** symbol($v$) $\neq$ symbol($s$) (false)
4 **For** $t \in$ Parents($s$) **do** (nil)
5 **For** $\{s, t\}$ **do**
6 **If** not Complete($t$)
7 **and** $t \neq v$ (true) **then**
8 **If** Binding($t$) = NIL (true) **then**
9 Binding($t$) := $v$
10 Push(t)
11 **If** $s \neq v$ **then**
12 **If** symbol($s$) $\in X$ (false) **Else**
13 Update Links with two new pairs (3 instructions)
14 Complete($s$) := True

For every $v$ node in the variable term ($n_1$ of them), instructions 0–11 and 14 are executed (line 11 is found false). Together with 6 instructions from outside the loop, this makes $19n_1$ instructions. Then the corresponding constant term node $t$ is removed from the stack and processed by one more pass through the **While** loop. If $t$ is found to create a clash with $v$, then only instructions 0–3 are executed with one output message (5 instructions). If $t$ is correctly matched, then instructions 0–6 and 11–14 are executed (6 is found false, Complete($v$) being true). Line 13 accounts for $1+k$ instructions where $k = \text{arity}(t)$. Thus for $t$ which creates no clash: $7 + 4 + k$ or 13 instructions ($k = 2$ here). Calls to Finish thus involve $19n_1 + 13n_2$ instructions.

**PW total** The complete count for PW is therefore

$$7(n_1 + n_2) + 19n_1 + 13n_2 = 26n_1 + 20n_2$$

Even when applying dag compaction needlessly and taking no advantage of shared structure, TM ($15n_1 + 19n_2$) is faster than PW ($26n_1 + 20n_2$). In the worst-case example taken here, if both terms are of comparable size TM is approximately 25% faster. If the constant term is much larger, then both algorithms are equally fast. It is thus possible to include dag compaction within a linear-time term matching algorithm and obtain better worst-case performance than the only other applicable algorithm of equal complexity. In fact, for repeated processing of this worst-case input, TM would effectively proceed twice as fast by compacting one new problem instance while the other is being traversed in DFS. A more significant but more intricate analysis is given in Appendix B, based on a set of test cases.

## 4.3 Sublinear-time tree matching

This section shows that by relaxing the requirement for sequential input we can reduce the complexity of term matching algorithms. Unfortunately, this reduction comes at the expense of communication overheads so that its practical value is uncertain. We will only give an overview of the method, which leaves much for optimisation. It nevertheless gives an asymptotic upper bound for TMATCH. The method described here is a simulation of a shared-memory algorithm, the definition of which is given in terms of trees only. This restriction is not essential however and could be overcome by bottom-up processing of the type used in the DC algorithm.

If parallel input is allowed, or if data is already loaded when term matching is to take place, it is possible to design a mesh algorithm for TMATCH of sublinear complexity. This is done by simulating a logarithmic shared-memory algorithm. The author does not know of any sublinear-time mesh algorithm for P-complete problems such as UNIF while the present type of simulation applies directly to any NC algorithm which uses a linear number of processes and linear space. It is

therefore of interest in itself. A related question of theoretical interest is whether the suspected separation of P and NC is meaningful for algorithms on our model of mesh-connected computers. More directly, which problems admit sublinear-time algorithms on (two- or three- dimensional) rectangular arrays of processes. These questions are beyond the scope of this thesis and we will not consider them any further.

The simulated algorithm (to be called VKR here) is that of Verma, Krishnaprasad and Ramakrishnan [57]. The VKR algorithm solves the TMATCH problem for *trees* of size $n$, in $O(\log^2 n)$ synchronous steps, using $n$ processes. Its model of computation is the Concurrent Read, Exclusive Write Parallel Random Access Machine (CREW PRAM), described in [19]. The reader is assumed to be familiar with this model.

VKR assumes that its input is evenly distributed among the processes, one node per process. It uses a small number of arrays of size $n$, also distributed among the processes. Each step in its execution is either a global write instruction, a global read instruction, or some local processing.

### 4.3.1 Simulating random access on a mesh

Let us now consider the following model on which to implement VKR: a square mesh of $n$ processes, each one only having access to its local memory and communicating with its four neighbours (Figure 4.6). It is simplest to assume a synchronous operation, every process executing one local operations in parallel with up to four communications at every time step. The PRAM's shared-memory is implemented by the union of the processes' local storage. Because the PRAM algorithm only uses $O(1)$ storage per process, we may make the following simplification (modulo a constant factor in speed): that each process is only responsible for one memory location. PRAM steps involving only local processing can be directly implemented on the mesh and we will concentrate on simulating the global memory instructions.

The key to this simulation is the observation that exclusive memory access reduces to sorting. Let us define the map

$$\text{access} : \text{processes} \to \text{processes}$$

such that process $p$ is requesting access to the local memory of process access($p$). Then if the memory operation is exclusive, access must be an injective function because of the above simplification. Without loss of generality every process makes some memory request, so that access is a bijection [1]. It follows that sorting request packets according to their destination in the mesh network is sufficient to implement a global memory access.

[1] Sorting a partial access function is in fact slightly more difficult than sorting a total function. Appendix C describes a method for doing a partial sort in less than twice the time for a total sort.

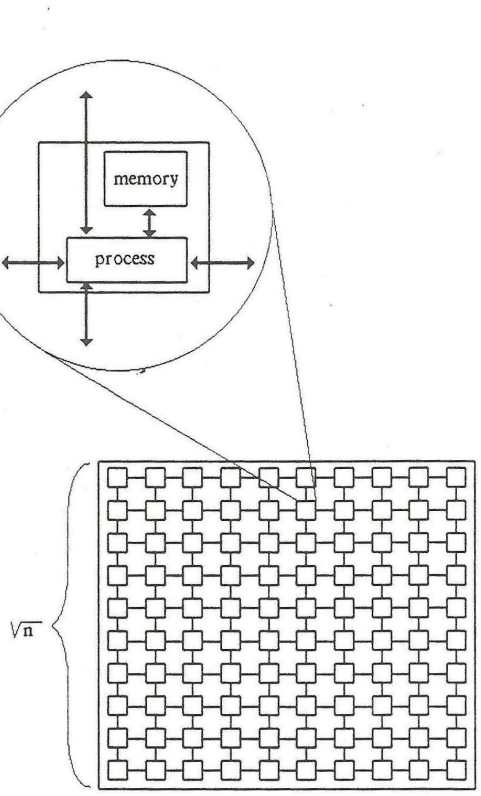


Figure 4.6: The two-dimensional square mesh

### 4.3.2 Exclusive read or exclusive write

The simplest operation to implement in this manner is the exclusive write. Every process then creates a packet containing both access($p$) and the write operation it requests. Packets are sorted and arrive at the appropriate processes, which then execute the write operation.

An exclusive read requires one more sorting phase. Every process creates a packet containing its address $p$ and the address of its read request access($p$). The packets are sorted by destination and then process access($p$) creates a packet containing the result of the read operation with its return address $p$. A second sort then returns the results to their requesting processes.

### 4.3.3 Shearsort

Sorting on the two-dimensional mesh is efficiently accomplished by the following algorithm.

**Algorithm 11** *Shearsort (Schnorr and Shamir 1986 [51]).*
*Given an $m \times m$ array of processes, each one containing one element of the sequence to sort, repeat* $\log m + 1$ *times:*

1. *Sort odd numbered rows to the right, even numbered rows to the left.*
2. *Sort the columns downwards.*

*At the end, the array is sorted into 'snake-like row-major order'.* [ ]

Each sorting of a row or column can be done by an odd-even exchange method in $m$ steps and in this application, $m = \sqrt{n}$. The complete shearsort algorithm therefore requires time $O((\log m + 1)m) = O(\sqrt{n} \log n)$ in total.

### 4.3.4 Concurrent read

To complete the simulation, it only remains to treat the case of a concurrent read, where several processes may request the contents of a memory location. This can be accomplished by a fixed number (approximately 6 here) of sorting operations in the following manner.

Let the processes be numbered from 1 to $n$ in snake-like order, as illustrated by the upper-left corners in Figure 4.7. Process $p$ has a request for reading the contents of location access($p$) for $p : (1..n)$, and access is non-injective (A). Each process then creates a request of the form (Send contents of access($p$) to $p$), abbreviated as the pair (access($p$) **to** $p$).

The requests are then sorted by their first component. Requests for reading the same location are thus placed in contiguous segments of the snake-like order

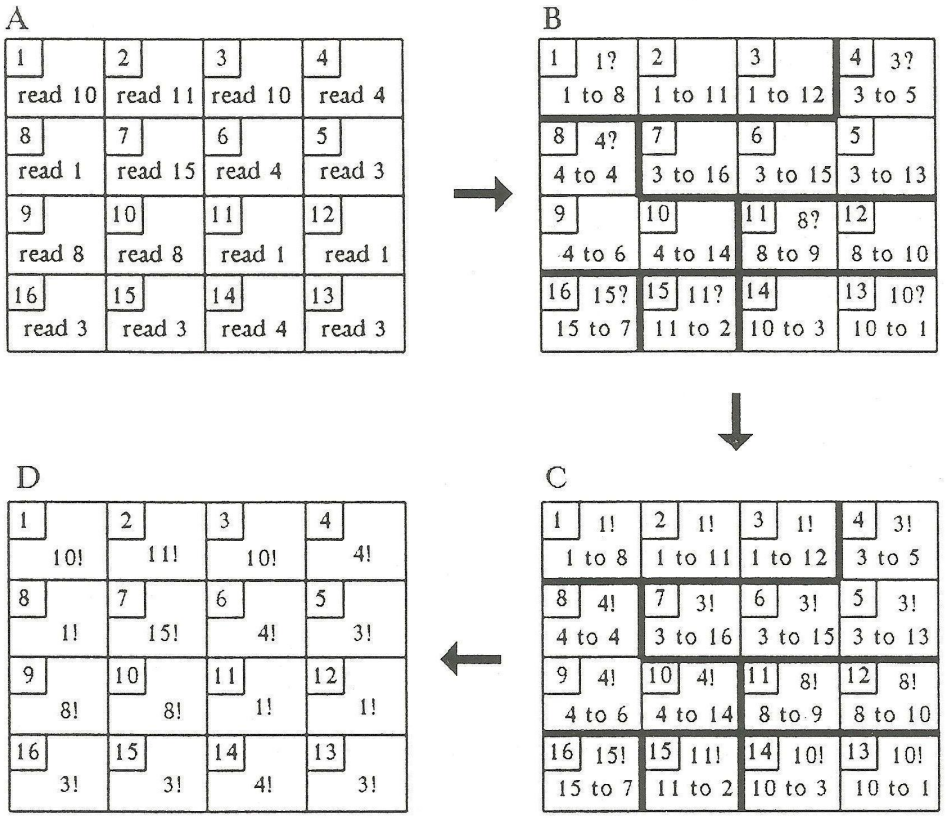


Figure 4.7: Simulation of a PRAM concurrent read

(B). Each process then tests whether it has the lowest index in its segment and if so will act as 'representative' for the segment. The set of read requests held by representatives is, by definition, injective and so can be handled by the procedure for an exclusive read. For example in (B), process 4 is requesting the value of location 3 (3?) as representative for processes 4, 5, 6 and 7, on behalf of processes 5, 13, 15 and 16 respectively.

Once the representatives have received the requested values, they simultaneously broadcast them to their respective segment. This is first done horizontally and then vertically in $2\sqrt{n}$ steps. For example in (C), the value of location 3 (3!) has been received by process 4 and then transmitted to processes 5, 6 and 7.

At this point it only remains to send the values to their original (distinct) requesting processes. This is done by a single sort operation. For example, the value of location 3 stored by process 6 is sorted to its destination, process 15 (D).

Like the exclusive operations, the concurrent write has been simulated in a constant number of calls to shearsort.

### 4.3.5 TMATCH in sublinear time

The complete VKR algorithm executes $O(\log^2 n)$ PRAM operations, and by using the above procedures, it can be simulated in $O(\sqrt{n}\log^3 n)$ parallel steps on the mesh.

A simple optimisation can reduce this complexity by a factor of $\log n$. Inspection of the VKR algorithm as described in [57] reveals that the most costly phase is in fact a sorting operation. At the heart of VKR is a shared-memory sorting algorithm which is repeated $\log n$ times. Since we are using sorting to simulate every elementary PRAM step, a PRAM sort can be counted as a unit cost operation within the simulated VKR. In other words, simply avoid simulating every step of a sorting algorithm by a shearsort. By this simplification, the overall complexity of the simulation is reduced to $O((\sqrt{n}\log n)\log n) = O(\sqrt{n}\log^2 n)$.

The simulation could be optimised further to $O(\sqrt{n}\log n)$ by using a more complex $O(\sqrt{n})$ sorting algorithm of Schnorr and Shamir [51].

## 4.4 UNIF and P-completeness

The following is a discussion of the practical consequences of theoretical results concerning UNIF.

As explained earlier, the P-completeness of the full unification problem UNIF suggests that this problem is not decidable in polylogarithmic time. In fact the best known algorithms for P-complete problems are linear in the number of nodes of the problem graph [60]. They are therefore useful for non-sparse graphs $G$ satisfying:

$$|V(G)| = o(|E(G)|).$$

In the case of unification however, we have assumed a bounded arity $k$ for all function symbols, so that the problem graph contains at most $k\,|V(G)|$ edges. The algorithm described in [60] therefore does not provide a sublinear-time algorithm for UNIF. In fact, the discovery of sublinear algorithms for sparse P-complete problems would be a theoretical breakthrough and we conjecture that such algorithms do not exist, at least in the case of UNIF.

Special cases of UNIF would appear to be different from the general problem. Several restrictions of the problem (other than TMATCH) are known to be in NC. The case where terms contain a fixed number of variables UNIF[k vars] is computable in time $O(\log^2 n)$ by Yasuura's algorithm [62][63]. Other special cases in NC are monadic unification UNIF[mon] (all function symbols having arity 1) [31][2] and unification with a fixed number of variables which occur more than once (UNIF[k reps]) [31].

The NC algorithms for UNIF[mon] and UNIF[k reps] reduce the problem to a sequence of transitive closure operations. Transitive closure is computable in $O(\log^2 n)$ with $O(n^3)$ processes on a PRAM. A practical algorithm could be constructed by simulating the $O(\log^2 n)$ shared-memory procedure on a three-dimensional mesh of $n^3$ processes. This simulation would take time $O(n \log^4(n))$ by using a three-dimensional version of shearsort, as in the previous section. Another possibility would be to use a systolic array of $n^2$ processes for the algebraic path problem [46][47][21].

In fact the solution of a sparse transitive closure problem is a prerequisite for solving UNIF, as shown in Proposition 3. The reduction given there also applies — with simple modifications if any — to UNIF[mon] and UNIF[k reps]. Moreover, the best known sublinear-time algorithms for this problem require $\Omega(\mathrm{MM})$ processes, where MM is the exponent of $n$ in the sequential complexity of $n \times n$ matrix multiplication. MM is known to be 2 or more (see references in [45]). This limitation therefore carries over to UNIF[mon] and UNIF[k reps]. We will not expand on this here as the use of even a quadratic number of processes is likely to make these methods prohibitive with anything but small inputs. On the other hand, small inputs are probably better handled with simpler algorithms such as PW.

Yasuura's algorithm UNIF[k vars] does not even allow such a simulation because it is based on a reduction to the directed hypergraph access problem (DHGRP) which is a generalisation of transitive closure. The instance of DHGRP constructed there contains a quadratic number of nodes, and so requires $\Omega(n^{2MM})$ processes to be solved in polylogarithmic time. The use of $\Omega(n^4)$ processes makes the diameter of any three-dimensional network $\Omega(n^{4/3})$. Any algorithm effectively using such a network must therefore have complexity $\Omega(n^{4/3})$ which makes it slower than PW.

In summary, solutions to the full UNIF problem are limited by the absence of sublinear solutions to sparse P-complete problems and straightforward solutions to its NC special cases are limited by the number of processes required for fast transitive closure.

# Chapter 5

# Associative-commutative matching

This chapter deals with the complexity and parallel solution of special cases of the TMATCH-AC problem.

As mentioned in chapter 2, TMATCH-AC is most often used as a decision problem with a single output unifier — just like UNIF — although its solution may have an exponential number of most general unifiers. Also, some special cases of TMATCH-AC are tractable and can be given efficient solutions such as the recursive algorithm of Benanav et al. [4].

On the other hand, UNIF-AC is usually implemented as an enumeration algorithm which outputs all most general unifiers. For this reason, tractable cases of the *decision* problem UNIF-AC are of no practical interest in themselves. We will therefore concentrate on the efficient solution of TMATCH-AC.

## 5.1 Restrictions of TMATCH-AC

This section raises again the question of the complexity of special cases for TMATCH-AC. In particular, the problem becomes NP-complete as soon as variables are allowed to appear twice in the input. This fills a gap left in the results of chapter 2.

### 5.1.1 Previous results

Let us first recall the relevant results given in chapter 2.

- **Theorem** 8 (Benanav, Kapur and Narendran 1985[4]).
  TMATCH-AC (decision problem) is computable in time $O(|s|\,|t|^3)$ for linear terms of sizes $|s|$ and $|t|$[]
  This restriction of the problem will be denoted by TMATCH-AC[linear].

- **Theorem** 10. TMATCH-AC remains NP-complete if each variable occurs at most three times in the terms to be matched [ ]
  This special case will be denoted by TMATCH-AC[3 occ].

- **Corollary** 1. TMATCH-AC remains NP-complete if the input terms are planar [ ]
  This special case will be denoted by TMATCH-AC[planar].

### 5.1.2 New definitions

As argued in subsection 4.2.1, variables which occur in both terms are irrelevant to the problem of term matching and we will assume the following problem definition.

**Definition 15** *TMATCH-AC, associative-commutative term matching.*
*INPUT: Terms* $s, t$ *where* $Var(t) = \emptyset$.
*OUTPUT: If* $s$ *and* $t$ *are not unifiable in the AC theory then 'NO'. Otherwise an idempotent substitution* $\sigma$ *such that* $\sigma(s) =_{AC} t$.

Special cases of the problem are defined in the same way. Without loss of generality we only consider idempotent unifiers, which bind variables to constant subterms and not to other variables. As in chapter 4, the first input term $s$ will be called the *variable term* and the other one the *constant term*.

In what follows we will assume that AC-terms are in flattened form (see Chapter 3 for definition).

For the purposes of discussing TMATCH-AC algorithms, let us make the following definitions. Given a term $t$ of the form $f[t_1, \ldots, t_m]$, $f \in F_{AC}$, any term $f[t_{i_1}, \ldots, t_{i_n}]$ such that $\{i_1, \ldots, i_n\}$ is a proper subset of $\{1, \ldots, m\}$ without repetitions, will be called an *L-subterm* of $t$. Recursively, any L-subterm of $t_i$ is also called an L-subterm of $t$. L-subterms are not subterms in the usual sense, but occur naturally in the images of unifiers for AC-terms. Also, the number of L-subterms can be exponential in the number of (usual) subterms. For example, if $f[x, y]$ is to be matched with $f[t_1, \ldots, t_m]$, then there are $2^m - 2$ possible L-subterms as images for either $x$ or $y$.

### 5.1.3 TMATCH-AC[2 occ] is NP-complete.

Let TMATCH-AC[2 occ] be that restriction of associative-commutative matching where variables occur at most twice. The tractability of TMATCH-AC[linear] and the NP-completeness of TMATCH-AC[3 occ] raise the question of whether TMATCH-AC[2 occ] is tractable or not. The answer is apparently negative, as suggested by theorem 11 below.

The following problem definition is needed in the proof of theorem 11.

**Definition 16** *EXACT COVER BY 3-SETS, (X3C).*
*INSTANCE: A set $S$ of size $|S| = 3q$ and a collection $C$ of 3-element subsets of $S$.*
*QUESTION: Does $C$ contain an exact cover for $S$ ?*
*An exact cover is a subcollection $C' \subseteq C$ such that every element of $S$ occurs in exactly one member of $C'$.* [ ]

Notice that an exact cover must be of size $q$. The X3C decision problem is known to be NP-complete [18].

Because a solution to X3C is specified by a union of disjoint sets, it can be represented as a commutative list without the need for idempotence i.e. as an L-subterm. This allows a direct reduction of X3C to TMATCH-AC, and moreover the reduction constructs a variable term where variables occur only twice.

**Theorem 11** *TMATCH-AC[2 occ] is NP-complete.*
**Proof**
Inclusion in NP follows from that of the general problem. NP-hardness follows from a polynomial (in fact $O(n)$) reduction from X3C, as described below.

Let $S$ and $C$ be given as an instance of X3C:

1. $S = \{s_1, s_2, \ldots, s_{3q}\}$
2. $C = \{c_1, c_2, \ldots, c_m\}$
3. $\forall i : (1..m)$.
   $c_i = \{s_{i1}, s_{i2}, s_{i3}\},\ s_{ij} \in S$.
   $|c_i| = 3$

Without loss of generality, let also $m \geq q$.

Let now $\widehat{s_1}, \widehat{s_2}, \ldots \widehat{s_{3q}}$ be distinct (syntactic) constants representing the elements of $S$ and also $f, g \in F_{AC}$, $h \in F_2 - F_{AC}$ and $x_1, \ldots, x_m \in X$.

For $i : (1..m)$ define

$$\widehat{c_i} = f[\widehat{s_{i1}}, \widehat{s_{i2}}, \widehat{s_{i3}}]$$

where $\widehat{s_{ij}}$ is the symbol corresponding to $s_{ij}$.

Define also the terms:

$$s = h(\quad g[x_1, \ldots, x_q, \ldots, x_m],\quad f[x_1, \ldots, x_q]\quad)$$
$$t = h(\quad g[\widehat{c_1}, \widehat{c_2}, \ldots, \widehat{c_m}],\qquad f[\widehat{s_1}, \ldots, \widehat{s_{3q}}]\quad)$$

and consider the problem TMATCH-AC$(s, t)$. Variables occur at most twice in $s$ and so this is an instance of TMATCH-AC[2 occ].

We will now prove that $s, t$ are matchable if and only if $C$ contains an exact cover for $S$, and so X3C reduces to TMATCH-AC[2 occ]. The reduction consists of the construction of term graphs for $s$ and $t$ and can be computed in linear time.

1. Suppose $\phi$ is a substitution such that $\phi(s) =_{AC} t$. Then

$$\phi(g[x_1, \ldots, x_m]) =_{AC} g[\widehat{c_1}, \ldots, \widehat{c_m}]$$

and so

$$g[\phi(x_1), \ldots, \phi(x_m)] =_{AC} g[\widehat{c_1}, \ldots, \widehat{c_m}]$$

so that $\{\phi(x_1), \ldots, \phi(x_q)\}$ is a subset of size $q$ of $\{\widehat{c_1}, \ldots, \widehat{c_m}\}$, the $\widehat{c_i}$ being distinct by definition. Without loss of generality let this subset be

$$\widehat{C'} = \{\widehat{c_1}, \ldots, \widehat{c_q}\}.$$

Now $\phi(s) =_{AC} t$ also implies

$$\phi(f[x_1, \ldots, x_q]) =_{AC} f[\widehat{s_1}, \ldots, \widehat{s_{3q}}]$$

and therefore

$$f[\phi(x_1), \ldots, \phi(x_q)] =_{AC} f[\widehat{s_1}, \ldots, \widehat{s_{3q}}].$$

Substitution of $\widehat{C'} = \{\widehat{c_1}, \ldots, \widehat{c_q}\}$ for $\{\phi(x_1), \ldots, \phi(x_q)\}$ gives

$$f[\widehat{c_1}, \ldots, \widehat{c_q}] =_{AC} f[\widehat{s_1}, \ldots, \widehat{s_{3q}}]$$

$$f[f[\widehat{s_{11}}, \widehat{s_{12}}, \widehat{s_{13}}], \ldots, f[\widehat{s_{q1}}, \widehat{s_{q2}}, \widehat{s_{q3}}]] =_{AC} f[\widehat{s_1}, \ldots, \widehat{s_{3q}}]$$

$$f[\widehat{s_{11}}, \widehat{s_{12}}, \widehat{s_{13}}, \ldots, \widehat{s_{q1}}, \widehat{s_{q2}}, \widehat{s_{q3}}] =_{AC} f[\widehat{s_1}, \ldots, \widehat{s_{3q}}]$$

which can only hold if the union of triples in $C'$ is $S$ i.e. if $C'$ is an exact cover for $S$.

2. Conversely, assume $C' = \{c_1, \ldots, c_q\}$ is an exact cover for $S$. Then

$$f[\widehat{c_1}, \ldots, \widehat{c_q}] =_{AC} f[\widehat{s_1}, \ldots, \widehat{s_{3q}}](*)$$

and the following substitution is a unifier for $s$ and $t$:

$$\phi = \{x_i \mapsto \widehat{c_i} \mid i : (1..m)\}.$$

$$\begin{array}{lllll}
\phi(s) & = h( & g[\phi(x_1), \ldots, \phi(x_m)], & f[\phi(x_1), \ldots, \phi(x_q)] & ) \\
& = h( & g[\widehat{c_1}, \ldots, \widehat{c_m}], & f[\widehat{c_1}, \ldots, \widehat{c_q}] & ) \quad \text{def'n.} \\
& = h( & g[\widehat{c_1}, \ldots, \widehat{c_m}], & f[\widehat{s_1}, \ldots, \widehat{s_{3q}}] & ) \quad (*) \\
& = t & & &
\end{array}$$

[]

### 5.1.4 Restricting the number of variables

Of the restrictions considered so far only one is known to be tractable: variables with single occurrences (linearity). It would be useful to extend the linear case to a more general problem while retaining tractability. A related possibility is to restrict the total number of variables in the terms.

Let TMATCH-AC[$k$ vars] be the associative-commutative term matching problem where the variable term $s$ contains a fixed number $k$ of distinct variables. It would appear that TMATCH-AC[$k$ vars] could be solved by exhaustive trial of at most $|t|^k$ sets of bindings for the variables (here $t$ is the constant input term). Unfortunately this is not the case because variables may be bound to L-subterms as well as to individual subterms in $t$. Exhaustive search would therefore have to attempt an exponential number of bindings: $\Omega(2^{k|t|})$.

A more complex method is required, using the structure of the constant term to constrain the search for a matching substitution. As a result, even the restriction to TMATCH-AC[1 var] has a non-trivial proof of tractability. Its solution reduces to transitive closure in term dags.

We will first need a technical lemma.

**Lemma 9** *Let $\prec$ be a strict partial order relation on the set $S$, and $n$ an integer in the range $(1..|S|)$. If there exists a subset $S' \subseteq S$ of size $n$ such that*

$$\forall s' : S'. \forall s : S - S'.\ s \prec s' \ (*)$$

*then this subset is unique.*

**Proof**

Assume $S'_1$ and $S'_2$ are two subsets of size $n$ which satisfy (*), and $S'_1 \neq S'_2$. Then there must be two elements $s_1 \in S'_1 - S'_2$ and $s_2 \in S'_2 - S'_1$. But then (*) both implies $s_1 \prec s_2$ and $s_2 \prec s_1$, which contradicts the acyclicity of $\prec$. [ ]

As argued above, the most difficult instances of the term matching problem are those where the variable can be bound to L-subterms.

**Lemma 10** *TMATCH-AC[1 var] is tractable for instances $(s, t)$ where*

$$s = f[x, \ldots, x, s_1, \ldots, s_n]\ ;\ t = f[t_1, \ldots, t_{n+m}].$$

**Proof**

If some of the $s_i$ are constant subterms, for example if $x$ does not occur in $s_1$ then matching succeeds if and only if

$$\exists j : (1..n+m).\ t_j = s_1$$

and

$$\text{TMATCH-AC}(f[x, \ldots, x, s_2, \ldots, s_n],\ f[t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_{n+m}])$$

succeeds. If $s$ and $t$ are precompacted *together* then testing for equality can be done in constant time and the elimination of constant subterms can be done in linear time. We therefore assume that $x$ occurs in every $s_i$.

Similarly, we may assume that $x$ does not repeat as top-level argument in $s$ and that $s_1, \ldots, s_n$ are distinct subterms. If some of the top-level arguments in $s$ repeat, it is sufficient to group them into equality classes, do the same for $t_1, \ldots, t_m$ and solve the matching problem without repetitions.

We are therefore left with a problem of the form TMATCH-AC$(s, t)$ where

$$s = f[x, s_1(x), \ldots, s_n(x)] \text{ and } t = f[t_1, \ldots, t_{n+m}].$$

If $m = 0$ then matching trivially fails, so let $m \geq 1$. A unifier $\sigma$ must be such that $\sigma(x)$ is the function $f$ applied to a subset of size $m$ of $T = \{t_1, \ldots, t_{n+m}\}$ and $T_s = \{\sigma(s_1), \ldots, \sigma(s_n)\}$ is the rest of $T$.

Assuming the existence of a unifier $\sigma$, let $t'_1, \ldots, t'_{n+m}$ be the permutation of $t_1, \ldots, t_{n+m}$ such that

1. $T_s = \{t'_1, \ldots, t'_n\}$, $T - T_s = \{t'_{n+1}, \ldots, t'_{n+m}\}$
2. $\forall i : (1..n).\ \sigma(s_i) = t'_i$ and
3. $\sigma(x) = f[t'_{n+1}, \ldots, t'_{n+m}]$.

Define $\prec$ as the strict subterm relation on $T$ so that $t_a \prec t_b$ if $a, b \in \{1, \ldots, n+m\}$, $t_a$ is a subterm of $t_b$ and $t_a \neq t_b$. Notice that $\prec$ is acyclic and therefore has maximal elements, not included in any others.

Condition (3.) implies that all of $T - T_s$ occurs in $\sigma(x)$ and since $x$ occurs in every $s_i$, $\sigma(x)$ must also occur in every $\sigma(s_i)$:

$$\forall i : (1..n).f[t'_{n+1}, \ldots, t'_{n+m}] \text{ is a subterm of } t'_i$$

and therefore

$$\forall i : (1..n).\forall j : (n+1..n+m).\ t'_j \prec t'_i.$$

For all $t'_i \in T_s$, every $t'_j \in T - T_s$ is a subterm and we will then call $t'_i$ a *cover* for $T - T_s$.

The problem has now been reduced to the following decision: is there a subset $T_s$ of size $n$ in $T$ such that every element of $T_s$ is a cover for $T - T_s$. Lemma 9 showed that such a set $T_s$ is unique if it exists. Moreover, $T_s$ must contain all of the maximal elements for $\prec$. Therefore a greedy strategy can be applied to verify its existence and eventually construct it.

Initially $T_s$ is set to contain the maximal elements of $\prec$. If there are more than $n$, then fail. If there are exactly $n$ then stop, $T_s$ has been found. While the size of $T_s$ is less than $n$, find a $t_j \in T - T_s$, such that it covers $T - T_s - \{t_j\}$. If found, include it in $T_s$. If no such $t_j$ is found, then fail. When $T_s$ reaches size $n$ stop with success.

If $T_s$ exists and has been computed as above, then the only possible unifier $\sigma$ is defined by (3.) It only remains to apply it to $s$ and test $\sigma(s)$ and $t$ for AC-equality.

The whole calculation is based on precomputing $\prec$ — the transitive closure of the dag for $t$ — and can be completed in polynomial time. [ ]

The general case simply involves finding an occurrence of the variable as argument to an AC-function.

**Theorem 12** *TMATCH-AC[1 var] is tractable.*
**Proof**
Let $s, t$ be the input terms, $\mathrm{Var}(s) = \{x\}$ and $\mathrm{Var}(t) = \emptyset$. If $s$ and $t$ are to be matched, all occurrences of $x$ in $s$ must be bound to the same subterm or L-subterm of $t$.

A first possibility is that $x$ occurs as argument to a free function symbol; for example $s$ could contain the subterm $h(x, s_1, s_2)$. In that case, binding $x$ to an L-subterm of $t$ will prevent $h(x, s_1, s_2)$ from matching any subterm of $t$. In other words, $h$ cannot concatenate L-subterms into proper subterms of $t$. This implies that $x$ can only be bound to a subterm, of which there are at most $|t|$. The decision problem is then answered by exhaustively taking every subterm of $t$, substituting it for $x$ and then testing $s, t$ for AC-equality. This can be done in quadratic time $O(|t|\,(|s| + |t|))$.

We will therefore assume that $x$ occurs as argument to $f \in F_{AC}$. Let $s'$ be a subterm of the form $f[x, \ldots, x, s_1, \ldots, s_n]$ in $s$, then $s'$ must match with a subterm $t' = f[t_1, \ldots, t_{n+m}]$ of $t$. There are at most $|t|$ such $t'$ and so the problem reduces to the decision TMATCH-AC$(s', t')$ with fixed $t'$, which has been shown to be tractable in Lemma 10. [ ]

When $k > 1$, TMATCH-AC[$k$ vars] is not known either to be tractable or NP-complete. An answer to this question is of great practical interest and would probably generalise to the problem TMATCH-AC[$k$ reps] where only a fixed number of variables have repeated occurrences. A polynomial-time solution to TMATCH-AC[$k$ reps] parametrised by $k$ could be used in term rewriting (or other) applications where $k$ may be precomputed. For small $k$ this would be a considerable improvement over exhaustive search methods.

Figure 5.1 summarises the relationships among TMATCH-AC, its restrictions and related problems. Solid arrows indicate polynomial-time reductions used in proofs. Dotted arrows indicate trivial $O(1)$ reductions. DC-AC is the dag compaction problem for associative-commutative terms, as described in Chapter 3.

## 5.2 Parallel methods

We now consider the use of parallelism in solving TMATCH-AC[linear] and TMATCH-AC[1 var]. Both problems are of sufficiently high complexity for com-

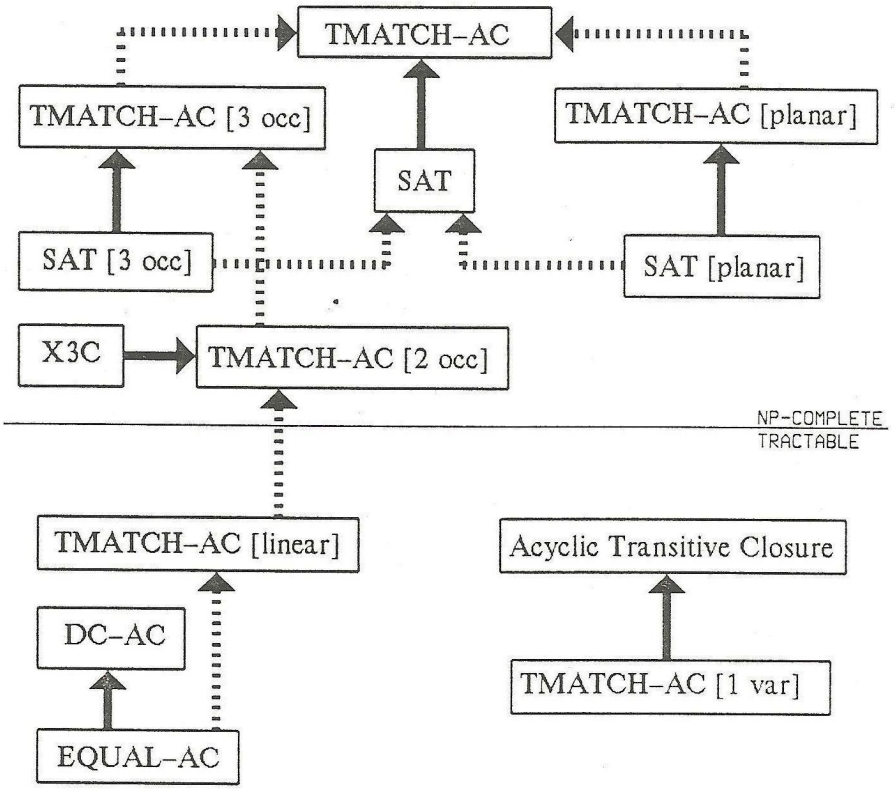


Figure 5.1: A hierarchy of problems related to associative-commutative term matching

munication costs to be small, given a suitable problem decomposition. The practical value of a specific algorithm for such a narrow special case as TMATCH-AC[1 var] is however uncertain. Moreover, the solution outlined in the proof of tractability for TMATCH-AC[1 var] will yield a static domain decomposition in a trivial way, so as to create up to $\Omega(|t|)$ processes. For this reason, we will not consider TMATCH-AC[1 var] in any more detail.

With TMATCH-AC[linear], a parallel version of the Benanav-Kapur-Narendran (BKN) algorithm [4] is possible and potentially more useful. Given terms $s = f[s_1, \ldots, s_n]$ and $t = f[t_1, \ldots, t_m]$ to match, BKN proceeds by recursively attempting matching for every pair $(s_i, t_j)$ and encoding the results as a bipartite graph $G$ on $(1..n) \times (1..m)$. The initial match succeeds if and only if $G$ contains a (graph) *matching* $M$ of size $n$, where a matching is defined to be a set of non-adjacent edges. A matching of size $n$ in $G$ is maximum, and its existence can be decided in cubic time (see references in [4]). The recursive nature of the BKN algorithm makes it ideal for a task farming implementation.

A given task will then create subtasks corresponding to the recursive calls of the sequential algorithm. An analysis similar to that given by Benanav et al. will show that this method can have complexity $O(|s|\,|t|^2)$ instead of the sequential $O(|s|\,|t|^3)$, using at most $|t|$ processes on a square mesh *and* provided the terms are relatively deep, i.e. their depth is linear in their size. For shallow and wide terms, the cost of distributing subtasks is superior to the gains in parallelism.

# Chapter 6

# Conclusion

The systolic dag compaction algorithm suggests an implementation as an off-line memory management function, perhaps integrated with a garbage collection processor. Experiment with applications will be needed to determine its area of applicability alone or combined with the term matching algorithm TM.

There appears to be little scope for acceleration of the full UNIF problem due to P-completeness. However, TMATCH and other NC restrictions of UNIF can be solved with distributed algorithms in sublinear time. It will be interesting to see how such methods can be applied.

The tractability of special cases for TMATCH-AC requires further study, so that parallelism can be applied to a wider range of applications. Of particular interest would be the classification of TMATCH-AC[$k$ vars] or TMATCH-AC[$k$ reps] as tractable or NP-complete.

# Appendix A

# Proof of the invariant for algorithm DC

**Algorithm 12** DC: dag compaction.
**Input** List of packets $P_1, P_2, \ldots, P_{n_0}$ for the nodes of term dag G.

$$P_i = [A_i, E_i, \text{arity}_i, \text{symbol}_i, \text{son}_i]$$

**Output** List of packets $P'_1, P'_2, \ldots, P'_{n_0}$ such that $\{P'_i | i \in (1..n_0), A_i = E_i\}$ is a representation for the compaction of G.

**Procedure**

```
1. for t=2 to 2n_0 − 2 do
2.     parallel (i, j) ∈ (1..n_0)^2 | i + j = t + 1
3.         if arity_i = arity_j
4.         and symbol_i = symbol_j
5.         and ∀r : (1..arity_i). son_i[r] = son_j[r]
6.         then { A_i ≡ A_j }
7.                 E_i:= min(E_i, A_j)
8.                 E_j:= min(E_j, A_i)
9.         else { A_i ≢ A_j, update the son values }
10.                for r=1 to arity_i do
11.                    if son_i[r] = A_j then son_i[r]:=E_j
12.                for r=1 to arity_j do
13.                    if son_j[r] = A_i then son_j[r]:=E_i [ ]
```

**Lemma 11** *In the execution of DC, when packet $P_i$ meets packet $P_j$, $P_i$ has met with $P_1, \ldots, P_{j-1}$ and $P_j$ has met with $P_1, \ldots, P_{i-1}$ [ ].*

**Lemma 12** *After every execution of line 13 in DC, the following invariant holds.*

$$\forall (i, j) : (1..n_0)^2.\ i + j = t + 1 \Rightarrow INV(i, j)$$

*where INV(i,j) is the conjunction of the two conditions:*

1. $E_i = \min(A_i, \min\{A_x | x \in (1..j), A_x \equiv A_i\})$

2. $\forall r : (1..arity_i).$
$$son_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & if\ \ s_i[r] \in \{A_1, \ldots, A_j\} \\ s_i[r] & otherwise \end{array} \right]$$

**Proof**
The proof is by induction on $t$. The base case $i = 1$ (or $j = 1$) is verified as a direct consequence of the initialisations $E_i{:=}A_i$ and $\text{son}_i[r]{:=}s_i[r]$, followed by the execution of lines 3–13.

Consider a pair $(i, j)$ such that $i + j = t + 1$ and $i, j > 1$, before execution of lines 3–13 at step t. By the first lemma, and because $i, j > 1$, lines 3–13 were executed at step $t - 1$ with the pairs $(i, j - 1)$ and $(i - 1, j)$, so that by induction hypothesis we have INV$(i, j - 1)$ and INV$(i - 1, j)$ as preconditions:

1. $E_i = \min(A_i, \min\{A_x | x \in (1..j - 1), A_x \equiv A_i\})$

2. $\forall r : (1..\text{arity}_i).$
$$\text{son}_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & \text{if}\ \ s_i[r] \in \{A_1, \ldots, A_{j-1}\} \\ s_i[r] & \text{otherwise} \end{array} \right]$$

3. $E_j = \min(A_j, \min\{A_x | x \in (1..i - 1), A_x \equiv A_j\})$

4. $\forall r : (1..\text{arity}_j).$
$$\text{son}_j[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_j[r]\} & \text{if}\ \ s_j[r] \in \{A_1, \ldots, A_{i-1}\} \\ s_j[r] & \text{otherwise} \end{array} \right]$$

Precondition 2 implies $\text{son}_i[r] \equiv s_i[r]$ and $\text{son}_i[r] \leq s_i[r]$, and similarly for $\text{son}_j$.

It will be sufficient to consider the following cases for execution of lines 3–13:
C1. $(\text{arity}_i = \text{arity}_j) \wedge (\text{symbol}_i = \text{symbol}_j) \wedge (\forall r.\ \text{son}_i[r] = \text{son}_j[r])$
C2. $\text{symbol}_i \neq \text{symbol}_j$
C3. $(\text{arity}_i = \text{arity}_j) \wedge (\text{symbol}_i = \text{symbol}_j) \wedge (\exists r.\ \text{son}_i[r] \neq \text{son}_j[r])$.

In each case we will show that INV(i,j) is established as a postcondition.

**Case C1**
If C1 holds then lines 3–5 evaluate to true and lines 7,8 are executed. Nodes $A_i$ and $A_j$ are labelled with the same symbol and for any $r \in (1..\text{arity}_i)$ we have:

$$\begin{array}{rcll} s_i[r] & \equiv & \text{son}_i[r] & (\Leftarrow 2) \\ \text{son}_i[r] & = & \text{son}_j[r] & (\Leftarrow C1) \\ \text{son}_j[r] & \equiv & s_j[r] & (\Leftarrow 4) \\ s_i[r] & \equiv & s_j[r] & (\text{conclusion}). \end{array}$$

Corresponding sons of $A_i$ and $A_j$ are therefore equivalent and it follows — by definition of the equivalence of nodes — that $A_i \equiv A_j$.

By symmetry, we will consider only the execution of line 7:

$$E_i := \min(E_i, A_j)$$

The result is as follows.

$$\begin{array}{rcll} E_i & = & \min(\min(A_i, \min\{A_x | x \in (1..j-1), A_x \equiv A_i\}), A_j) & (\Leftarrow 1) \\ E_i & = & \min(A_i, \min(\min\{A_x | x \in (1..j-1), A_x \equiv A_i\}, A_j) & \\ E_i & = & \min(A_i, \min\{A_x | x \in (1..j), A_x \equiv A_i\}) & (\Leftarrow (A_j \equiv A_i)) \ (i) \end{array}$$

For all $r$, the value of $\mathrm{son}_i[r]$ is unchanged from precondition 2:

$$\mathrm{son}_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & \text{if } s_i[r] \in \{A_1, \ldots, A_{j-1}\} \\ s_i[r] & \text{otherwise} \end{array} \right] (ii)$$

A node cannot be equivalent to one of its sons (by acyclicity) and so $A_i \equiv A_j$ implies $A_j \neq s_i[r]$. It follows that $s_i[r] \in \{A_1, \ldots, A_{j-1}\}$ if and only if $s_i[r] \in \{A_1, \ldots, A_j\}$ and equation (ii) can therefore be rewritten as

$$\mathrm{son}_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & \text{if } s_i[r] \in \{A_1, \ldots, A_j\} \\ s_i[r] & \text{otherwise} \end{array} \right] (iii)$$

The conjunction of (i) and (iii) is INV(i,j), which settles the case C1.

**Truth of C2 or C3 implies inequivalence of $A_i$ and $A_j$**

If C2 or C3 holds, then lines 3–5 evaluate to false, lines 10–13 are executed, and we will first show that $A_i \not\equiv A_j$ before proving that INV(i,j) is a postcondition.

When C2 is true, $A_i$ and $A_j$ are labeled with different symbols and so cannot be equivalent.

When C3 is true, $\exists r.\ \mathrm{son}_i[r] \neq \mathrm{son}_j[r]$ and we want to conclude that $s_i[r] \not\equiv s_j[r]$, which implies the inequivalence of $A_i$ and $A_j$. We will suppose, without loss of generality, that $i > j$. Precondition 4 is then:

$$\mathrm{son}_j[r] = \min\{A_y | A_y \equiv s_j[r]\} \ (iv)$$

because $s_j[r]$ must lie at a lower level than $A_j$ and so

$$s_j[r] \in \{A_1, \ldots, A_j\} \subseteq \{A_1, \ldots, A_{i-1}\}$$

which makes the first clause of precondition 4 apply.
Precondition 2 is:

$$\mathrm{son}_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & \text{if } s_i[r] \in \{A_1, \ldots, A_{j-1}\} \\ s_i[r] & \text{otherwise} \end{array} \right] \quad (v)$$

Let the if clause of (v) be true. Then $s_i[r] \equiv s_j[r]$ would imply they share the same representative node:

$$\min\{A_y | A_y \equiv s_i[r]\} = \min\{A_y | A_y \equiv s_j[r]\}.$$

The left side of this last equation is equal to $\text{son}_i[r]$ (by (v)), and the right side is equal to $\text{son}_j[r]$ by (iv), which contradicts C3. Therefore $s_i[r] \not\equiv s_i[r]$.

Suppose now that the if clause of (v) is false, then

$$\begin{array}{lcl} s_i[r] & \notin & \{A_1, \ldots, A_{j-1}\} \quad (\Leftarrow v) \\ s_i[r] & \in & \{A_j, \ldots, A_{i-1}\} \quad (i > j,\ A_i > s_i[r]) \end{array}$$

but then $s_i[r] \equiv s_j[r]$ means that $s_i[r]$ is at the same level as a son of $A_j$, which is impossible because it was just shown that $s_i[r]$ is in the interval $\{A_j, \ldots, A_{i-1}\}$ where every node must be of equal or higher level than $A_j$ (level sort). Again we conclude that $s_i[r] \not\equiv s_j[r]$.

**INV(i,j) is satisfied as postcondition to C2 or C3**

Let us consider now when $A_i \not\equiv A_j$, the effect of executing lines 3–13. By symmetry it is only necessary to consider the effect of lines 10,11.

Execution of lines 10,11 does not change the value of $E_i$ from precondition 1 and so

$$\begin{array}{rcl} E_i & = & \min(A_i, \min\{A_x | x \in (1..j-1), A_x \equiv A_i\}) \\ E_i & = & \min(A_i, \min\{A_x | x \in (1..j), A_x \equiv A_i\}),\ (\Leftarrow (A_j \not\equiv A_i)) \quad (vi) \end{array}$$

which verifies the first half of INV(i,j).

The execution of line 11 for a given value of $r$, is as follows. Before line 11, precondition 2 holds:

$$\text{son}_i[r] = \left[ \begin{array}{ll} \min\{A_y | A_y \equiv s_i[r]\} & \text{if } s_i[r] \in \{A_1, \ldots, A_{j-1}\} \\ s_i[r] & \text{otherwise.} \end{array} \right] \text{(vii)}$$

There are two possibilities to consider according to which clause of (vii) is true.

The first possibility is that

$$\text{son}_i[r] = \min\{A_y | A_y \equiv s_i[r]\} \text{ and } s_i[r] \in \{A_1, \ldots, A_{j-1}\}$$

If $\text{son}_i[r] = A_j$ then

$$A_j = \min\{A_y | A_y \equiv s_i[r]\} \text{ and } s_i[r] \in \{A_1, \ldots, A_{j-1}\} \quad (viii)$$

and the assignment $\text{son}_i[r] := E_j$ is executed.

The final value of $\text{son}_i[r]$ is then given by precondition 3:

$$\begin{array}{rcl} \text{son}_i[r] & = & \min(A_j, \min\{A_x | x \in (1..i-1), A_x \equiv A_j\}) \\ \text{son}_i[r] & = & \min(\min\{A_y | A_y \equiv s_i[r]\}, \min\{A_x | x \in (1..i-1), A_x \equiv s_i[r]\}), \\ & & \quad (\Leftarrow (A_j \equiv s_i[r]) \wedge (viii)) \\ \text{son}_i[r] & = & \min(\min\{A_y | A_y \equiv s_i[r]\}, \min\{A_x | A_x \equiv s_i[r]\}), \\ & & \quad (\Leftarrow (s_i[r] \in \{A_1, \ldots, A_{i-1}\})) \\ \text{son}_i[r] & = & \min\{A_y | A_y \equiv s_i[r]\}. \end{array}$$

Together with the second half of (viii), this implies

$$\text{son}_i[r] = \min\{A_y | A_y \equiv s_i[r]\} \wedge s_i[r] \in \{A_1, \ldots, A_j\} \quad (ix)$$

which in turn implies the second half of INV(i,j). Together with (vi), this establishes INV(i,j).

If $\mathrm{son}_i[r] \neq A_j$ then there is no assignment instruction executed, $\mathrm{son}_i[r]$ still equals $\min\{A_y | A_y \equiv s_i[r]\}$ and with the second half of (viii) implies the second half of INV(i,j). Together with (vi), this proves INV(i,j).

The second possibility for the value of $\mathrm{son}_i[r]$ in (vii) is that

$$(\mathrm{son}_i[r] = s_i[r]) \wedge (s_i[r] \notin \{A_1, \ldots, A_{j-1}\}). \quad (x)$$

If $\mathrm{son}_i[r] = A_j$ then

$$\mathrm{son}_i[r] = A_j = s_i[r] \quad (xi)$$

and $\mathrm{son}_i[r]{:=}E_j$ is executed.

The final value of $\mathrm{son}_i[r]$ is given by precondition 3:

$$\begin{array}{lcl}
\mathrm{son}_i[r] & = & \min(A_j, \min\{A_x | x \in (1..i-1), A_x \equiv A_j\}) \\
\mathrm{son}_i[r] & = & \min(s_i[r], \min\{A_x | x \in (1..i-1), A_x \equiv s_i[r]\}) \\
\mathrm{son}_i[r] & = & \min(s_i[r], \min\{A_x | A_x \equiv s_i[r]\}) \quad (\Leftarrow (s_i[r] \in \{A_1, \ldots, A_{i-1}\})) \\
\mathrm{son}_i[r] & = & \min\{A_x | A_x \equiv s_i[r]\} \qquad (xii)
\end{array}$$

Now (xi) trivially implies:

$$s_i[r] \in \{A_1, \ldots, A_j\} \qquad (xiii)$$

and so (xii) with (xiii) are equivalent to (ix), which together with (vi) implies INV(i,j).

If $\mathrm{son}_i[r] \neq A_j$ then there is no assignment instruction and (x) implies that

$$(\mathrm{son}_i[r] = s_i[r] \neq A_j) \wedge (s_i[r] \notin \{A_1, \ldots, A_{j-1}\})$$

therefore

$$(\mathrm{son}_i[r] = s_i[r]) \wedge (s_i[r] \notin \{A_1, \ldots, A_j\})$$

which implies (ix) and again INV(i,j) is verified.

This settles the cases C2, C3 and completes the proof. [ ]

# Appendix B

# Test cases for TM

This appendix compares the efficiency of TM with that of PW on a chosen set of test cases.

Like the analysis given by Jaffar in [28], this one is based on a set of efficiency measures rather than a single one such as average-case complexity. Unlike [28] however, we will not use experimental timings, making abstraction of implementation factors like the ratio of communication times to process instruction times.

To provide a detailed view of TM's strengths and weaknesses, a set of 11 test cases will be used. Although artificial and regular in structure, each test case clearly illustrates one aspect of the interplay between DEC, MER, the detection of clashes and input overheads in both algorithms. They are thus more helpful to the implementer than a random set of unstructured tests.

Each test case — an instance of the TMATCH problem — is parametrised by its depth or width. In each case, an *estimate* of the execution time (number of Random Access Machine operations) is given for both algorithms, taking into account concurrent instructions in TM. To make the test as general as possible, TM applies the compaction algorithm even to inputs like $A_0 I$ (Figure B.1) which are already compact. Some of these steps could be avoided in an environment which keeps track of substitutions and compactions, but we will not account for this here.

To comply with input requirements for PW, none of the test cases represents a variable as multiple nodes. Note that because of its use of dag compaction, TM would perform correctly and with the same speed, for inputs that would not satisfy this requirement.

The graph structure of each test case is illustrated by a figure in the following pages. For example, Figure B.1 shows test case $A_0 I$ for the parameter value $m = 3$ i.e. the terms

$$f(f(f(a,a),f(a,a)),f(f(a,a),f(a,a))$$

and

$$f(f(f(b,b),f(b,b)),f(f(b,b),f(b,b))$$

in compact form. The value $n = 6m + 2$ is the total dag size: number of nodes + number of edges.

The estimates are derived by reasoning along the same lines as the worst-case example in Subsection 4.2.5 . For TM, execution time is divided in four stages:

$T_1$ The delay when $DC_2$ is waiting for the level sorted constant term (number of idle time steps).

$T_2$ Computation time for $DC_2$ (from first input packet to last pair comparison).

$T_3$ Delay when $DFS_2$ is reading a packet from $DC_2$ which does not correspond to the next node in its synchronised traversal with $DFS_1$.

$T_4$ Traversal time for $DFS_1 - DFS_2$.

For PW, the calls to Finish are assumed to be made in top-down left-to-right order. Initially, only calls to function nodes in the variable term are made. Eventually, if subterms of the constant term are to be compared for equality, Finish is called on the leftmost subterm. Finally there can be calls of Finish for remaining variable nodes. Nodes are visited in pairs, the *left* node being visited by a call to Finish and the *right* one by a unification link (in the inner **While** loop of Finish). Computation time is divided in three parts:

$T_1'$ Preprocessing in $7(n_1 + n_2)$ steps, where $n_1 + n_2$ is the total number of nodes.

$T_2'$ For each 'left' node called by Finish, there are 6 instructions outside the **While** loop. Inside the loop there are 13 instructions for a function node. For a variable node there are the following 14 instructions as numbered in Subsection 4.2.5: 0, 1, 2, 4–10, 11(false), 12(true), write binding, 14.

$T_3'$ For each 'right' node visited through Links, there are 5 instructions if a clash is detected or $11 + k$ otherwise (see Subsection 4.2.5 for explanation). Here $k$ is the arity of that node.

The test cases are inspired by those of Jaffar in [28]. Like Martelli and Montanari [39] we will separate cases where matching fails from those where it succeeds. Accordingly there are two classes of test cases:

A: TM detects a clash.
B: TM succeeds.

The following function symbols are used: $f \in F_2$; $g \in F_1$; $a, b, c, d \in F_0$ and $x, y, z, v, w \in X$.

The instruction counts are summarised in Tables B.1, B.2, B.3 and B.4.

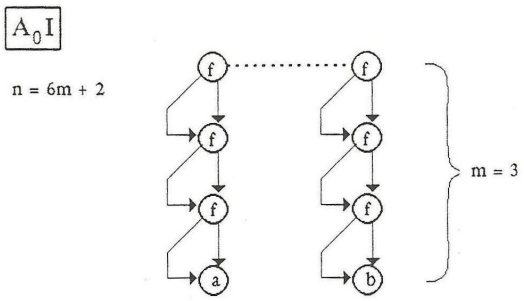


Figure B.1: Test case $A_0I$ for TM

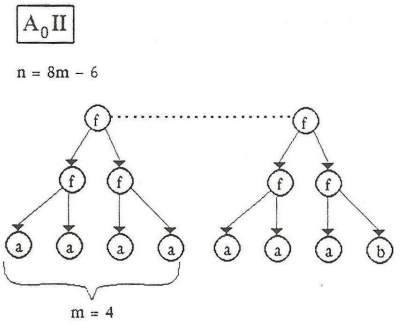


Figure B.2: Test case $A_0II$ for TM

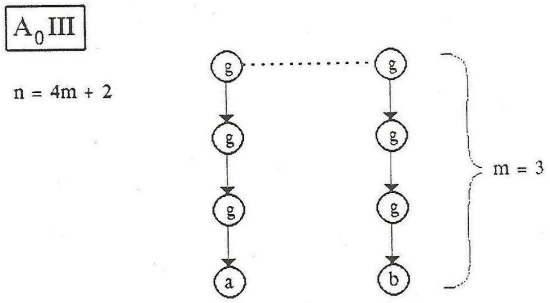


Figure B.3: Test case $A_0III$ for TM

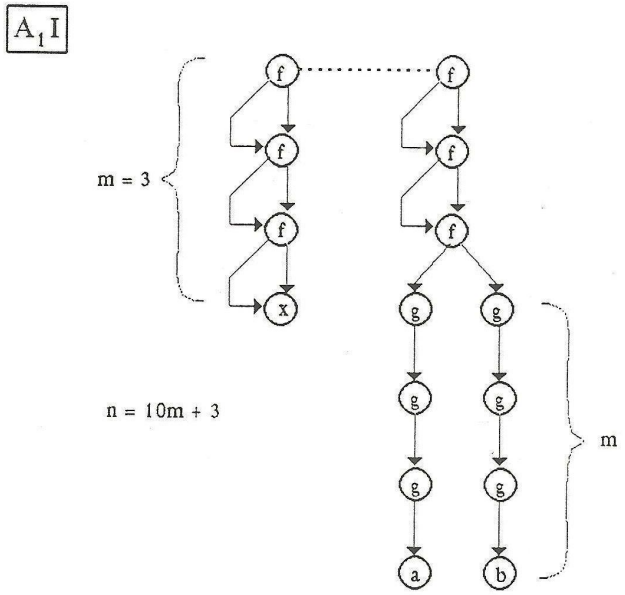


Figure B.4: Test case $A_1I$ for TM

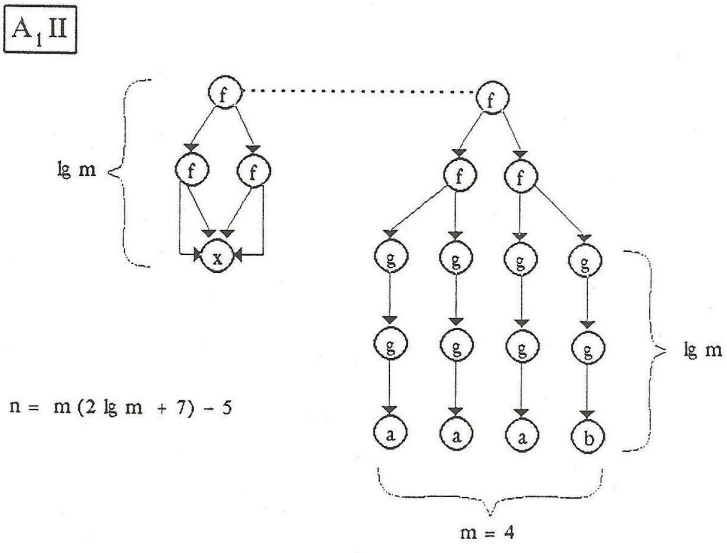


Figure B.5: Test case $A_1II$ for TM

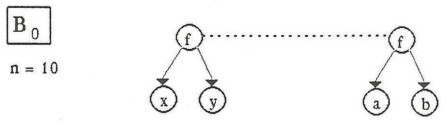


Figure B.6: Test case $B_0$ for TM

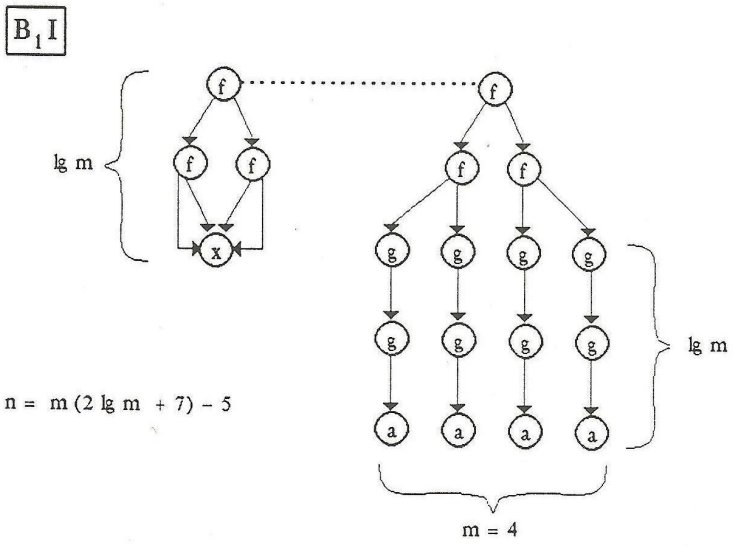


Figure B.7: Test case $B_1I$ for TM

Cases $A_0$ (I-III) cause failure because of a clash detected during the first application of DEC (before any application of MER).

Cases $A_1$ cause failure because of a clash after (one) application of MER.

Case $B_0$ is immediately successful and measures the algorithms' initialisation costs.

Cases $B_1$ (I,II) and $B_2$ are successful and involve either one or no application of MER.

Cases $B_3$ are successful and involve multiple applications of MER.

There are several conclusions suggested by Table B.4. The first one is that TM has half the startup overhead of PW (case $B_0$), 7 times the size of this small input.

On 7 of the 11 test cases, TM profits from dag compaction and lower overheads to execute 30–60% faster than PW.

PW takes advantage of cases like $A_1I$, $B_1I$ and $B_2$ where it does not need to traverse the bottom half of the constant term. On these three cases, it executes as fast as TM.

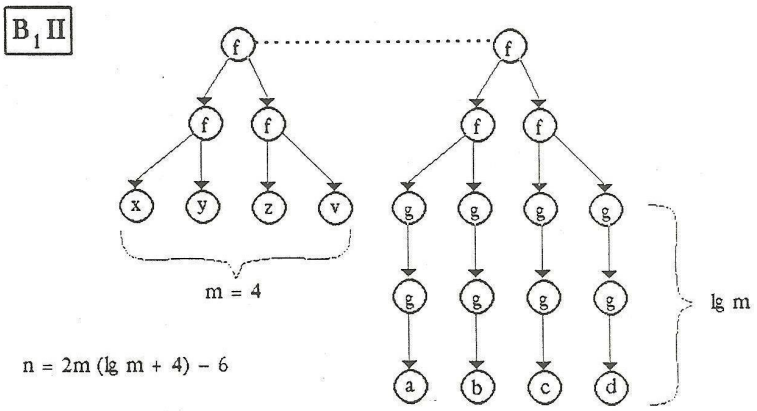


Figure B.8: Test case $B_1II$ for TM

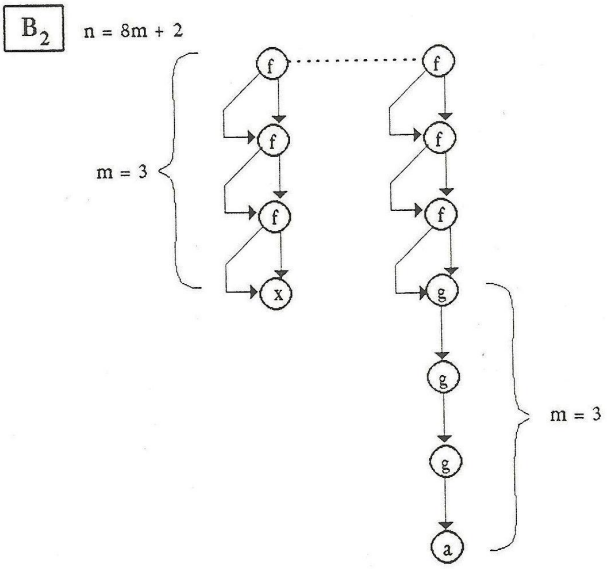


Figure B.9: Test case $B_2$ for TM

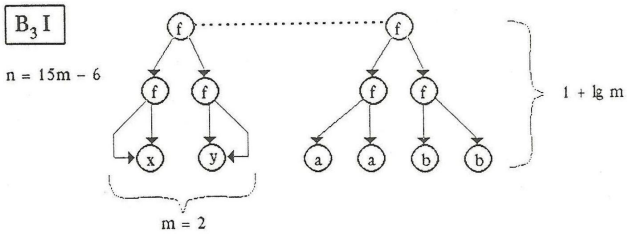


Figure B.10: Test case $B_3I$ for TM

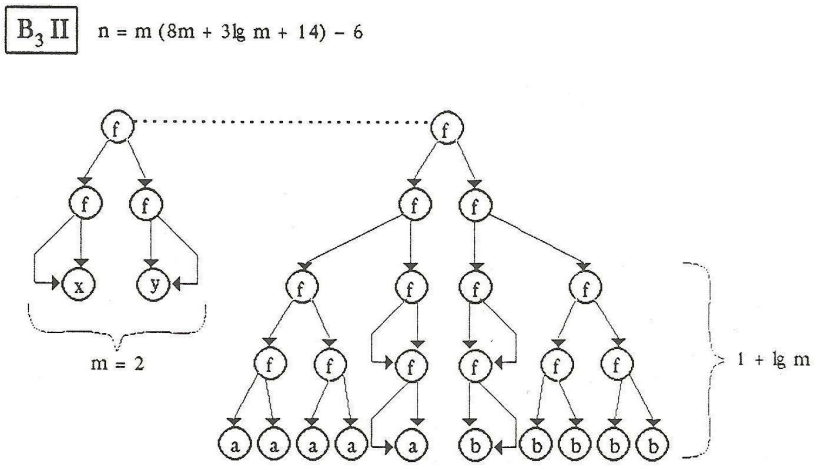


Figure B.11: Test case $B_3II$ for TM

| case | $T_1$ | $T_2$ |
|---|---|---|
| $A_0I$ | $3m$ | $16m$ |
| $A_0II$ | $6m$ | $32m$ |
| $A_0III$ | $2m$ | $8m$ |
| $A_1I$ | $3m$ | $48m$ |
| $A_1II$ | $3m + 2m \log m$ | $32m + 16m \log m$ |
| $B_0$ | 3 | 16(3) |
| $B_1I$ | $3m + 2m \log m$ | $32m + 16m \log m$ |
| $B_1II$ | $3m + 2m \log m$ | $32m + 16m \log m$ |
| $B_2$ | $5m$ | $32m$ |
| $B_3I$ | $6m$ | $64m$ |
| $B_3II$ | $6m^2 + 3m \log m + 3m$ | $64m^2 + 16m \log m + 32m$ |

Table B.1: Instruction counts for stages 1,2 of TM (additive constants discarded)

| case | $T_3$ | $T_4$ |
|---|---|---|
| $A_0I$ | 0 | $9m$ |
| $A_0II$ | $6(m - \log m)$ | $9\log m + 4m$ |
| $A_0III$ | 0 | $9m$ |
| $A_1I$ | 0 | $9m$ |
| $A_1II$ | $6(m - \log m)$ | $13\log m$ |
| $B_0$ | 0 | $9 + 5 + 5$ |
| $B_1I$ | $6(m - \log m)$ | $13\log m$ |
| $B_1II$ | $6(m - \log m)$ | $14m$ |
| $B_2$ | 0 | $13m$ |
| $B_3I$ | $12m - 6\log m$ | $27m$ |
| $B_3II$ | $12m - 6\log m$ | $27m$ |

Table B.2: Instruction counts for stages 3,4 of TM (additive constants discarded)

| case | $T'_1$ | $T'_2$ | $T'_3$ |
|---|---|---|---|
| $A_0I$ | $14m$ | $19m$ | $13m$ |
| $A_0II$ | $28m$ | $38m$ | $24m$ |
| $A_0III$ | $14m$ | $19m$ | $12m$ |
| $A_1I$ | $28m$ | $38m$ | $25m$ |
| $A_1II$ | $7m\log m + 21m$ | $19m + 19\log m$ | $12\log m(m+1) + 13m$ |
| $B_0$ | 42 | $19 + 19 + 19$ | $13 + 20 + 20$ |
| $B_1I$ | $7m\log m + 21m$ | $19m + 19\log m$ | $12\log m(m+1) + 13m$ |
| $B_1II$ | $7m\log m + 28m$ | $38m$ | $33m$ |
| $B_2$ | $21m$ | $19m$ | $13m$ |
| $B_3I$ | $49m$ | $57m$ | $57m$ |
| $B_3II$ | $56m^2 + 21m$ | $19m\log m + 57m$ | $48m^2 + 33m$ |

Table B.3: Instruction counts for PW (additive constants discarded)

| case | TM | PW |
|---|---|---|
| $A_0I$ | $28m$ | $46m$ |
| $A_0II$ | $49m$ | $89m$ |
| $A_0III$ | $19m$ | $45m$ |
| $A_1I$ | $60m$ | $91m$ |
| $A_1II$ | $18m \log m + 42m$ | $19m \log m + 53m$ |
| $B_0$ | 70 | 152 |
| $B_1I$ | $18m \log m + 42m$ | $19m \log m + 53m$ |
| $B_1II$ | $18m \log m + 55m$ | $7m \log m + 98m$ |
| $B_2$ | $50m$ | $53m$ |
| $B_3I$ | $109m$ | $162m$ |
| $B_3II$ | $72m^2 + 19m \log m + 74m$ | $104m^2 + 19m \log m + 110m$ |

Table B.4: Total instruction counts for both algorithms (upper bounds for TM, lower bounds for PW

On the example $B_1II$, TM also suffers from a useless application of dag compaction, where it becomes twice as slow as PW.

Inspection of Tables B.1 and B.2 shows that dag compaction ($T_1 + T_2$) either takes asymptotically longer or 200–300% longer than the rest of TM. In the former case, pipelining is useless, whereas it increases throughput in the later case.

# Appendix C

# Sorting a partial function on the mesh

We now describe a method for sorting an injective partial access function on the $\sqrt{n} \times \sqrt{n}$ mesh of processes. This method is of complexity $O(T + 3\sqrt{n})$, where $T$ is the time required for sorting a total injective function.

Processes will be numbered in snake-like order from 1 to $n$, as in section 4.3. Given a partial injective function

$$\text{access} : \ (1..n) \to (1..n),$$

process $i$ initially stores the value access($i$) for all $i \in$ domain(access) (see figure C.1). The algorithm proceeds in four sequential steps and terminates with all values access($i$) stored in process access($i$):

1. Empty processes are given the value $+\infty$ and the resulting complete function $(1..n) \to (1..n) \cup \{+\infty\}$ is sorted using shearsort (section 4.3) or any equivalent algorithm. All the finite values are then in the lower-indexed processes, in increasing order (figure C.2).

2. At this point, every column contains an increasing sequence of numbers with the following property. Let $a, b, c$ be three consecutive numbers in this sequence. If $c - b \leq \sqrt{n}$ then $b - a > \sqrt{n}$. As a result, if the destination row of $b$ and $c$ is the same, then the destination row of $a$ is at least two rows higher. Numbers may therefore be shifted downwards in their respective columns until each number is either in its destination row or in the one just above (figure C.3).

3. Numbers with a given destination row then slide along the snake-like order, at most $\sqrt{n}$ places, until every number reaches its destination row (figure C.4).

4. All that remains to do is then to sort the numbers within rows (figure C.5). This is done by a simple modification of the odd-even exchange sort: an

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 **16** |
| 8 **5** | 7 | 6 **9** | 5 **10** |
| 9 | 10 | 11 **15** | 12 **3** |
| 16 | 15 **2** | 14 **11** | 13 **7** |

Figure C.1: Initial sequence

| | | | |
|---|---|---|---|
| 1 **2** | 2 **3** | 3 **5** | 4 **7** |
| 8 **15** | 7 **11** | 6 **10** | 5 **9** |
| 9 **16** | 10 | 11 | 12 |
| 16 | 15 | 14 | 13 |

Figure C.2: After sort

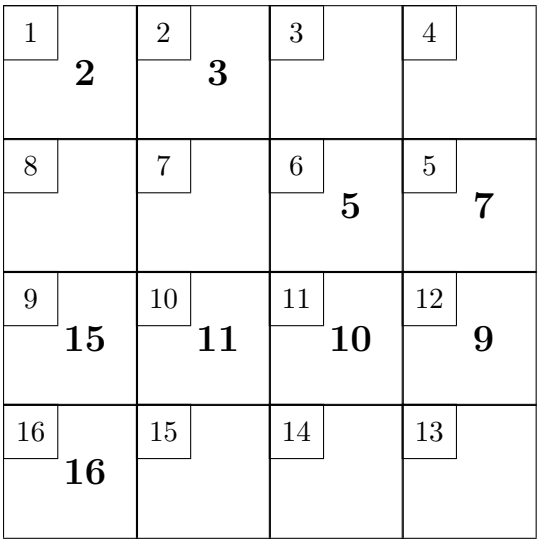


Figure C.3: After downward shift

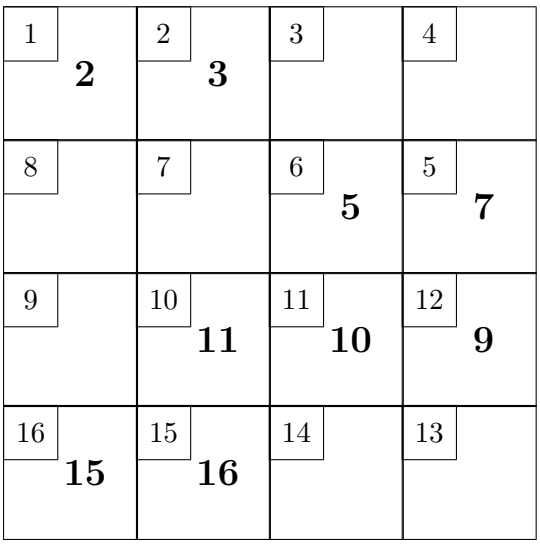


Figure C.4: After sliding

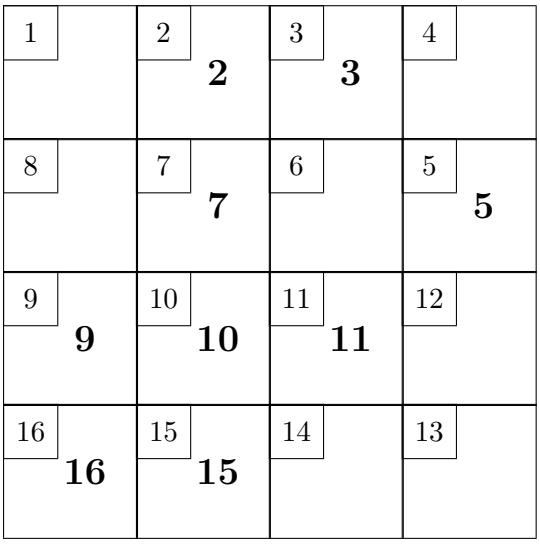


Figure C.5: Sorted sequence (after row-sort)

empty processe exchanges a number with its neighbour exactly when this brings the number closer to its destination process.

# Bibliography


[1] A.Aho, J.Hopcroft & J.Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] I.Auger & M.Krishnamoorthy, A parallel algorithm for the monadic unification problem, Bit no.25, 302-306, 1985.

[3] F.Baader, The theory of idempotent semigroups is of unification type zero, J. Aut. Reasoning 2, 283-286, 1986.

[4] D.Benanav, D.Kapur & P.Narendran, Complexity of matching problems, Proc. First International Conf. Rewrite Techniques and Applications, Dijon, France, May 1985.

[5] J.Bentley, *Programming Pearls*, Addison-Wesley,1986.

[6] W.Bibel, F.Kurfeß, K.Aspetsberger, P.Hinternaus & J.Schumann, Parallel inference machines, in Treleaven & Vanneschi eds., *Future Parallel Computers*, Springer LNCS no.272, 1987.

[7] P.Boizumault, *PROLOG, L'implantation*, Masson, 1988.

[8] Boyer & Moore, The sharing of structure in theorem-proving programs, in B.Meltzer and D.Michie editors, Machine Intelligence, Edinburgh Univ. Press, 101-116, 1972.

[9] W.Brauer, On minimizing finite automata, Bulletin of the EATCS, no.35, June 1988.

[10] S.A.Cook, A taxonomy of problems with fast parallel algorithms, in Proc. 1983 FCT Conf., Springer LNCS no.158, 1983.

[11] M.Cosnard & M.Tchuente, Designing systolic algorithms by top-down analysis, Third Int. Conf. on Supercomputing, Boston, 1988.

[12] D.DeChampeaux, About the Paterson-Wegman linear unification algorithm, JCSS 32, 79-90, 1986.

[13] P.J.Downey, R.Sethi & R.E.Tarjan, Variations on the common subexpression problem, JACM 27, no.4, 758-771, Oct. 1980.

[14] C.Dwork, P.Kanellakis & J.Mitchell, On the sequential nature of unification, J. Logic Programming 1, 35-50, 1984.

[15] C.Dwork, P.Kanellakis & L.Stockmeyer, Parallel algorithms for term matching, Conf. Aut. Deduction, Springer LNCS no.230, 1986.

[16] A.Fortenbacher, An algebraic approach to unification under associativity and commutativity, J. Symb. Comp. 3, 217-229, 1987.

[17] R.P.Gabriel, *Performance and evaluation of Lisp systems*, MIT Press, 1985.

[18] M.R.Garey & D.S.Johnson, *Computers and Intractability*, W.H.Freeman and Co., 1979.

[19] A.Gibbons & W.Rytter, *Efficient Parallel Algorithms*, Cambridge Univ. Press, 1988.

[20] W.D.Goldfarb, The undecidability of the second-order unification problem (note), TCS 13, 225-230, 1981.

[21] G.Hains, Distributed algorithms for restricted unification, MSc dissertation, Oxford Univ. Computing Lab., 1987.

[22] G.Hains & B.S.Todd, The parallel implementation of a medical diagnostic system, Third Int. Conf. on Supercomputing, Boston, 1988.

[23] W.J.Hansen, Compact list representation: definition, garbage collection, and system implementation, CACM 12, no.9, pp. 499-507, Sept. 1969.

[24] A.J.G.Hey & D.J.Pritchard, Parallel applications on the RTP supernode machine, Third Int. Conf on Supercomputing, Boston, 1988.

[25] J.E.Hopcroft & J.D.Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley, 1979.

[26] G.P.Huet, The undecidability of unification in third order logic, Information and Control 22, no.3, 257-267, 1973.

[27] INMOS ltd., *Occam 2 reference manual*, Prentice-Hall, 1988.

[28] J.Jaffar, Efficient unification over infinite terms, New Generation Computing 2, 207-219, 1984.

[29] Ph.Jorrand, Design and implementation of a parallel inference machine for first order logic: an overview, in deBakker et al. eds., Proc. PARLE87 vol.I, Springer LNCS no.258, 434-445, 1987.

[30] P.C.Kanellakis, Logic programming and parallel complexity, TR no. CS-86-23, Dept.Comp.Sc., Brown University, 1986.

[31] P.C.Kanellakis & P.Z.Revesz, On the relationship of congruence closure and unification, Brown Univ., Comp.Sc. Report no. CS-87-27, Nov. 1987.

[32] D.Kapur & P.Narendran, NP-completeness of the set unification and matching problems, in Siekmann ed., 8th Int. Conf. on Automated Deduction, Springer LNCS no.230, 1986.

[33] C.Kirchner, A new equational unification method: A generalisation of Martelli-Montanari's algorithm, in Shostak ed., 7th Int. Conf. on Automated Deduction, Springer LNCS no.170, pp. 224-247, 1984.

[34] C.Kirchner, Methods and tools for equational unification, in Proc. Colloquium on Resolution of Equations in Algebraic Structures, Austin, Texas, May 1987.

[35] D.Kozen, Complexity of finitely presented algebras, Proc. 9th annual ACM STOC, 164-177, 1977.

[36] P.Lincoln & J.Christian, Adventures in associative-commutative unification (summary), in Lusk and Overbeek eds., 9th Int. Conf. on Automated Deduction, Springer LNCS no.310, 358-367, 1988.

[37] P.McKenzie, Private communication, Université de Montréal, 1987.

[38] J.Maluszynski & H.J.Komorowski, Unification-free execution of logic programs, 2nd Symposium on Logic Programming, IEEE, 78-86, 1985.

[39] A.Martelli & U.Montanari, An efficient unification algorithm, ACM TOPLAS 4, 258-282, 1982.

[40] U.Montanari & J.Goguen, An abstract machine for fast parallel matching of linear patterns, TR no. SRI-CSL-87-3, Comp.Sc.Lab. SRI, 1987.

[41] M.Ohkubo, H.Yasuura & S.Yajima, On parallel computation time of unification for restricted terms, in RIMS Kokyuroku, Math. Found. Comp.Sc. and Applications, Kyoto University, 1987.

[42] I.Parberry, *Parallel Complexity Theory*, Pitman - John Wiley, 1987.

[43] M.S.Paterson & M.N.Wegman, Linear unification, JCSS 16, 158-167, 1978.

[44] S.L.Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[45] C.C.Ribeiro, Parallel computer models and combinatorial algorithms, Annals of Discrete Math. no.31 (North-Holland), 325-364, 1987.

[46] Y.Robert & D.Trystram, Parallel implementation of the algebraic path problem, CONPAR-86, Springer LNCS no.237, 1986.

[47] Y.Robert & D.Trystram, An orthogonal systolic array for the algebraic path problem, Computing, no.39, 187–199, 1987.

[48] J.A.Robinson, A machine-oriented logic based on the resolution principle, JACM 12, no.1, 23-41, 1965.

[49] J.A.Robinson, Computational logic: The unification computation, in Michie and Meltzer eds. Machine Intelligence 6, Edinburgh Univ. Press, 63-72, 1971.

[50] M.Schmidt-Schauss, Unification under associativity and idempotence is of type nullary, J. Aut. Reasoning 2, 277-281, 1986.

[51] C.P.Schnorr & A.Shamir, An optimal sorting algorithm for mesh connected computers, $18^{th}$ ACM STOC, 255-263, 1986.

[52] J.Siekmann, Universal unification, in Shostak ed., 7th Int. Conf. on Automated Deduction, Springer LNCS no.170, 1984.

[53] M.E.Stickel, A complete unification algorithm for associative-commutative functions, Proc. 4th Int. Joint Conf. A.I., Tbilisi USSR, 1975.

[54] M.E.Stickel, An introduction to automated deduction, in Bibel and Jorrand eds., Fundamentals of Artificial Intelligence, Springer LNCS no.232, 1985.

[55] J.D.Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.

[56] E.Upfal, Efficient schemes for parallel communication, JACM 31, no.3, 507-517, 1984.

[57] R.M.Verma, T.Krishnaprasad & I.V.Ramakrishnan, An efficient parallel algorithm for term matching, in Proc. 1986 Conf. FSTTCS, Springer LNCS no.241, 1986.

[58] R.M.Verma & Ramakrishnan, Optimal time bounds for parallel term matching, Proc. $9^{th}$ Int. Conf. Automated Deduction, Springer LNCS no.310, 1988.

[59] P.M.B.Vitányi, Locality, communication and interconnect length in multicomputers, Report CS-R8708, CWI Amsterdam, Feb.1987.

[60] J.S.Vitter & A.Simons, New classes for parallel complexity: A study of Unification and Other Complete Problems for P, IEEE Trans. on Computers, c-35, no.5, 403-418, 1986.

[61] L.C.West, Picosecond integrated optical logic, IEEE Computer, pp.34-46, December 1987.

[62] H.Yasuura, The reachability problem on directed hypergraphs and computational complexity, Report no. AL83-42 IECE of Japan, November 1983.

[63] H.Yasuura, On parallel computational complexity of unification, Proc. Conf. on Fifth Gener. Comp.Sys. (FGCS), Tokyo, 1984.

[64] K.A.Yelick, Unification in combinations of collapse-free regular theories, J. Symb. Comp. 3, 153-181, 1987.