A GRAPHICAL DEDUCTION SYSTEM

by

Philip T. Cox
T. Pietrzykowski

Research Report CS-76-35

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

July 1976

ABSTRACT

A graphical deduction system is described for performing linear deduction in first-order predicate calculus. No ordering is imposed on the solution of subgoals, and structure is shared in such a way that economy of representation is maximal, consistent with the full use of lemmas. An efficient unification algorithm is used which allows substitution to be avoided, and allows the source of conflict to be located when a subgoal is found to be unsolveable, so that backtracking can be intelligently performed. Because of the graphical structure, only the offending parts of the deduction need be removed in backtracking, rather than the entire proof which was constructed after the cutting point.

1. INTRODUCTION

With the advent of Robinson's resolution system of first-order logic in 1965 [21], there was much enthusiasm and optimism concerning the future of mechanical theorem proving. However, it soon became obvious that this optimism was largely ill-founded, when early theorem proving programs were found to be unable to prove any but the simplest theorems without exceeding their usually generous storage limits. Consequently, there followed in the late sixties a rash of strategies for limiting the size of the search space generated by the resolution rule [17]. Unfortunately, little improvement was obtained, and many researchers, disillusioned with predicate logic as a problem-solving tool, took a more pragmatic approach. As a result, some new programming languages were developed especially for problem solving, for example [8, 18 and 23].

An important feature of these new problem-solving systems, was their use of problem reduction, a technique whereby a problem to be solved is replaced by a set of subproblems, the simultaneous solution of which implies the solution of the original problem. In the late sixties and early seventies, a more promising refinement of resolution appeared which also used this goal-subgoal structure, and is generally classified as "linear resolution" [15, 19]. These systems in turn have led to the development of programming languages based on predicate logic [6, 7, 10, 11].

Unfortunately, when the power of linear deduction systems is increased by introducing new deduction rules, in particular, a rule which allows the use of lemmas to obtain shorter proofs, it becomes necessary to impose a strict ordering on the solution of subgoals to preserve completeness [13, 14, 16]. Consequently, one of the most attractive features of the problem reduction approach, the parallel processing of subgoals, is lost.

Furthermore, linear deduction, like other resolution systems, suffers from two problems associated with "backtracking". Firstly, when a sub-problem is encountered which cannot be solved, the system must return to an earlier point in the proof to try an alternative solution for an earlier sub-problem. The usual way of doing this is to return to the last point at which there was an alternative solution: this may not, in fact, be the right place to try an alternative, and although the correct point will eventually be found, much effort will be expended in exhaustively generating an irrelevant part of the search space. Secondly, when backtracking occurs to a particular point, the entire proof after that point is discarded even though some of it may be correct.

The system proposed in this paper retains the power of the linear systems, but does not suffer from any of the problems described above.

Most theorem proving systems also suffer from the proliferation of data due to unnecessary repetition. Some attempts have been made to reduce this repetitiousness by structure-sharing, e.g. [4]; these attempts, however, have concentrated on representation but have not solved any of the other problems of theorem provers. Fortunately, economy of representation is a natural by-product of our solution to these problems.

In section 2 of this paper, we present some preliminary definitions, define our notation, and quote some well-known results.

In section 3, the structure of $\mathcal{S}$-graphs is defined, and results concerning the soundness and completeness of $\mathcal{S}$-graph deduction are quoted.

We discuss unification in section 4, and show how the algorithm we use may be modified to guide backtracking.

Finally, in section 5, we demonstrate the advantages of our deduction system.

All proofs have been omitted from this paper, and will be supplied in a later publication.

2. PRELIMINARIES

2.1 Notation for graphs

Definition: a labelled, directed graph Γ is an ordered triple $\langle N, A, T\rangle$, where N is the set of nodes, T is the set of labels, and $A \subseteq N \times T \times N$ is the set of arcs.

To avoid confusion at times when several graphs are being considered, we will use the expressions $N(\Gamma)$ and $A(\Gamma)$ to refer to these sets. Since T is mostly constant throughout the discussion, we will use $T(\Gamma)$ instead of T only when ambiguity is likely.

Definition: If Γ is a labelled, direct graph and $n \in N(\Gamma)$, the in-degree (out-degree) of n is the cardinality of the set

$$\{(x,\mathcal{L}, n) \mid (x,\mathcal{L}, n) \in A(\Gamma)\}$$

$$\left(\{(n,\mathcal{L}, x) \mid (n,\mathcal{L}, x) \in A(\Gamma)\}\right)$$

Definition: Γ' is a subgraph of a graph Γ if Γ' is a graph, and $N(\Gamma') \subseteq N(\Gamma)$, and $A(\Gamma') \subseteq A(\Gamma)$.

2.2 Notation and vocabulary from theorem-proving

All words such as "literal", "clause", "substitution", "variant", "term" and "expression" are used with their standard meanings familiar to readers of the literature on mechanical theorem-proving. The reader should refer to [21] for clarification.

Substitutions will be denoted by lower-case Greek letters. The composition of substitutions will be denoted by the symbol $\circ$, as in $\sigma\circ\gamma$. The application of a substitution σ to an expression e is written $e\sigma$.

A substitution is a set of ordered pairs $\{\langle v_1,e_1\rangle,\ldots,\langle v_n,e_n\rangle\}$ in which the v_i's are distinct variables and the e_i's are expressions. We

will normally write such a substitution as $\{v_1 \leftarrow e_1,\ldots,v_n \leftarrow e_n\}$.

If ℓ is a literal, $\tilde{\ell}$ denotes the literal identical to ℓ except for sign.

2.3 Unification

Some familiar definitions and results are presented here.

Definition: A set of expressions $E = \{e_1,\ldots,e_n\}$ is unifiable iff there is a substitution σ such that $e_1\sigma = \ldots = e_n\sigma$. σ is called a unifier of E. σ is a most general unifier (mgu) of E iff it is a unifier of E, and for any unifier γ of E, there is a substitution μ such that $\gamma = \sigma\circ\mu$

If E is a set of expressions and σ is any substitution, we use $E\sigma$ to denote the set of expressions obtained by applying σ to each element of E. In the case when σ unifies E, $E\sigma$ will also be used to denote the single expression in the set E .

Lemma 2.3.1: If γ and σ are both mgu's for a set E, then $E\sigma$ is a variant of E^{γ}.

The notion of unification is extended as follows.

Definition: If $\mathcal{E} = \{E_1,\ldots,E_n\}$ is a set of sets of expressions, $\mathcal{E}$ is unifiable iff there is a substitution σ which is a unifier of each of the sets E_i. σ is called a unifier of $\mathcal{E}$. σ is a most general unifier of $\mathcal{E}$ iff it is a unifier of $\mathcal{E}$ and for any unifier γ of $\mathcal{E}$, there is a substitution μ such that $\gamma = \sigma\circ\mu$.

Lemma 2.3.2: If $\mathcal{E} = \{E_1,\ldots,E_n\}$ is a set of sets of expressions, there exists a set $E = \{e_1,\ldots,e_k\}$ of expressions such that $\mathcal{E}$ is unifiable with mgu σ iff E is unifiable with mgu σ.

Lemma 2.3.3: If E_1 and E_2 are two sets of expressions, and if σ is an mgu for E_1 and γ is an mgu for $E_1\sigma \cup E_2\sigma$, then $\sigma\circ\gamma$ is an mgu for $E_1 \cup E_2$.

Corollary 2.3.4: Applying lemma 2.3.2 to lemma 2.3.3, we obtain the analogous result for sets of sets of expressions.

3. DEDUCTION GRAPHS

In this section, the rules for constructing deduction graphs are given, and results concerning the soundness and completeness of our deduction system are quoted.

Definition: If $\mathcal{S}$ is a set of clauses, a deduction graph from $\mathcal{S}$ ($\mathcal{S}$-graph) is a labelled, directed graph $\Gamma = \langle N, A, T\rangle$, where:

(a) $T = \{REPL, SUB, RED, FACT\}$

(b) TOP $\in N$, and TOP is a symbol which does not occur in $\mathcal{S}$.

(c) Γ is constructed recursively, using only those rules defined below.

Before we can define the rules for constructing $\mathcal{S}$-graphs, we must digress with the following four definitions.

Definition: If Γ is an $\mathcal{S}$-graph, the top clause of Γ is the set $\{n \mid (TOP, \mathcal{L}, n) \in A(\Gamma)\}$

Definition: If Γ is an $\mathcal{S}$-graph, and $n_0, n_{k+1} \in N(\Gamma)$, then n_0 is said to be an ancestor of n_{k+1} iff there exists a sequence of arcs $a_0,\ldots,a_k \in A(\Gamma)$ such that $a_i = (n_i, \mathcal{L}_i, n_{i+1})$ for $i=0,\ldots,k$. Such a sequence of arcs is called a path from n_0 to n_{k+1}. The path is said to have labels $\mathcal{L}_i$, for $i=0,\ldots,k$.

Definition: If Γ is an $\mathcal{S}$-graph, and n, m $\in$ N(Γ), then n is said to be a direct ancestor of m iff there is a path from n to m having no labels from the set {FACT, RED}.

Definition: If Γ is an $\mathcal{S}$-graph, and Γ' is a subgraph of Γ which is also an $\mathcal{S}$-graph, then Γ' is called a sub-$\mathcal{S}$-graph of Γ.

Certain objects are associated with each $\mathcal{S}$-graph: namely, the set of solved nodes, S(Γ); the set of leaves, L(Γ); and the constraint set, C(Γ). The constraint set is, in fact, a set of sets of expressions, and if it is unifiable, we denote its mgu by $\sigma(\Gamma)$.

We now define these objects, and the rules for constructing $\mathcal{S}$-graphs, as follows:

(0) If $\mathcal{L} = \{n_1, \ldots, n_k\}$ is any clause in $\mathcal{S}$, then Γ_0 is an $\mathcal{S}$-graph, where:

$N(\Gamma_0) = \{TOP\} \cup \mathcal{L}$

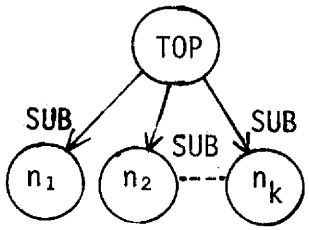
$A(\Gamma_0) = \{(TOP, SUB, n_i) \mid i=1,\ldots,k\}$

(i) $S(\Gamma_0)$ is empty

(ii) $C(\Gamma_0)$ is empty

(iii) $L(\Gamma_0) = \mathcal{L}$

Pictorially, we have:



(1) If Γ_{k-1} is an $\mathcal{S}$-graph, then Γ_k is an $\mathcal{S}$-graph, where Γ_k is derived from Γ_{k-1} by the application of one of the following rules:

Rule (1) Replacement

A. Simple replacement

If (a) $n \in L(\Gamma_{k-1})$

(b) $\mathcal{L} = \{\ell_1, \ldots, \ell_m\}$ is a variant of some clause in $\mathcal{S}$, and contains no variables which occur in any nodes of Γ_{k-1}.

(c) $C(\Gamma_{k-1}) \cup \{\{n, \tilde{\ell}_j\}\}$ is unifiable for some $j \in \{1,\ldots,m\}$

then $N(\Gamma_k) = N(\Gamma_{k-1}) \cup \mathcal{L}$

$A(\Gamma_k) = A(\Gamma_{k-1}) \cup \{(n, \text{REPL}, \ell_j)\}$

$\cup \{(\ell_j, \text{SUB}, \ell_i) \mid i \in \{1,\ldots,m\} - \{j\}\}$

B. Ancestor replacement

If (a) $n \in L(\Gamma_{k-1})$

(b) $\mathcal{L} = \{\ell_1,\ldots,\ell_m\}$ is a variant of $L(\Gamma')$ where Γ' is a sub-$\mathcal{L}$-graph of Γ_{k-1}, and $C(\Gamma')'$ is the corresponding variant of $C(\Gamma')$, such that no variables of $\mathcal{L}$ and $C(\Gamma')'$ occur in $C(\Gamma_{k-1})$ or in any nodes of Γ_{k-1}.

(c) $C(\Gamma_{k-1}) \cup \{\{n, \tilde{\ell}_j\}\} \cup C(\Gamma')'$ is unifiable for some $j \in \{1,\ldots,m\}$

then $N(\Gamma_k)$ and $A(\Gamma_k)$ are defined as for simple replacement.

(i) $S(\Gamma_k) = S(\Gamma_{k-1}) \cup \{n\}$

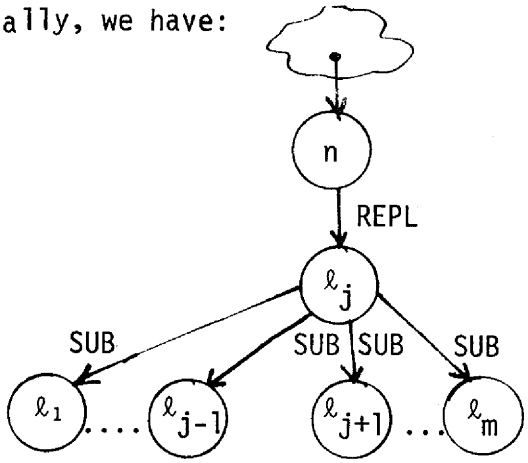
(ii) $C(\Gamma_k) = C(\Gamma_{k-1}) \cup \{\{n, \tilde{\ell}_j\}\}$ for simple replacement

$C(\Gamma_k) = C(\Gamma_{k-1}) \cup \{\{n, \tilde{\ell}_j\}\} \cup C(\Gamma')'$ for ancestor replacement.

(iii) $L(\Gamma_k) = (L(\Gamma_{k-1}) - \{n\}) \cup (\{\ell_1,\ldots,\ell_m\} - \{\ell_j\})$

We say that n is <u>replaced by subgoals</u> $\ell_1,\ldots,\ell_{j-1}, \ell_{j+1},\ldots,\ell_m$.

Pictorially, we have:



Rule (2) Reduction

If (a) $n \in L(\Gamma_{k-1})$

(b) $m \in S(\Gamma_{k-1})$ is a direct ancestor of n

(c) $C(\Gamma_{k-1}) \cup \{\{n, \tilde{m}\}\}$ is unifiable

(d) For every $p, q \in N(\Gamma_{k-1})$,

if (i) $(p, FACT, q) \in A(\Gamma_{k-1})$,

(ii) either $q \equiv n$, or there is a path from q to n which does not pass through m

then m is a direct ancestor of p

Then: $N(\Gamma_k) = N(\Gamma_{k-1})$

$A(\Gamma_k) = A(\Gamma_{k-1}) \cup \{(n, RED, m)\}$

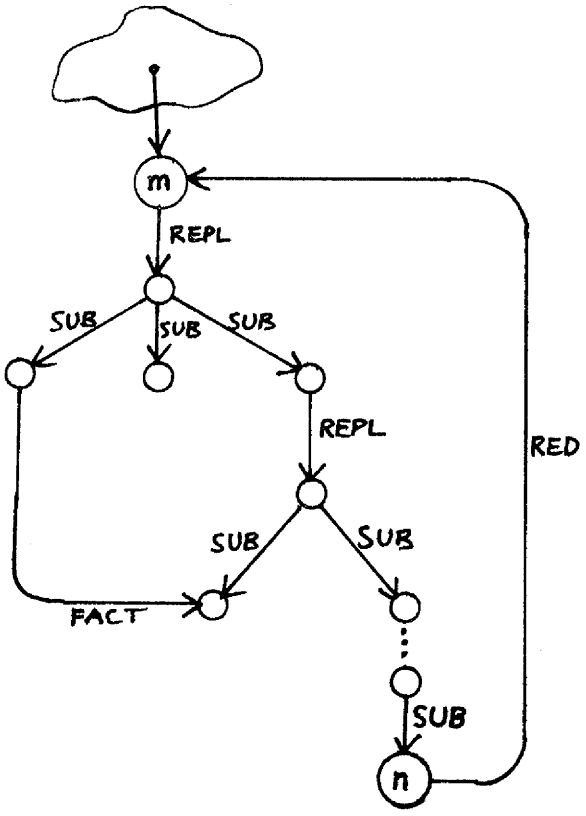
(i) $S(\Gamma_k) = S(\Gamma_{k-1}) \cup \{n\}$

(ii) $C(\Gamma_k) = C(\Gamma_{k-1}) \cup \{\{n, \tilde{m}\}\}$

(iii) $L(\Gamma_k) = L(\Gamma_{k-1}) - \{n\}$

We say that n is reduced to m

Pictorially:



Rule (3) Factoring

Like replacement, factoring divides into two cases. Since for both cases, the definitions of N, A, S, C and L are identical, we give the two sets of conditions followed by the single set of definitions.

A. Simple factoring

If (a) $p \in L(\Gamma_{k-1})$

(b) $q \in L(\Gamma_{k-1})$

(c) $C(\Gamma_{k-1}) \cup \{\{p, q\}\}$ is unifiable.

B. Back factoring

If (a) $p \in L(\Gamma_{k-1})$

(b) $q \in S(\Gamma_{k-1})$

(c) $C(\Gamma_{k-1}) \cup \{\{p, q\}\}$ is unifiable.

(d) Either q is not an ancestor of p

Or every path from q to p contains a RED label,

and for every $n, m \in N(\Gamma_{k-1})$

if (i) $(n, RED, m) \in A(\Gamma_{k-1})$

(ii) either $n \equiv q$ or there is a path from q to n which does not pass through m

then m is a direct ancestor of p.

Then: $N(\Gamma_k) = N(\Gamma_{k-1})$

$A(\Gamma_k) = A(\Gamma_{k-1}) \cup \{(p, FACT, q)\}$

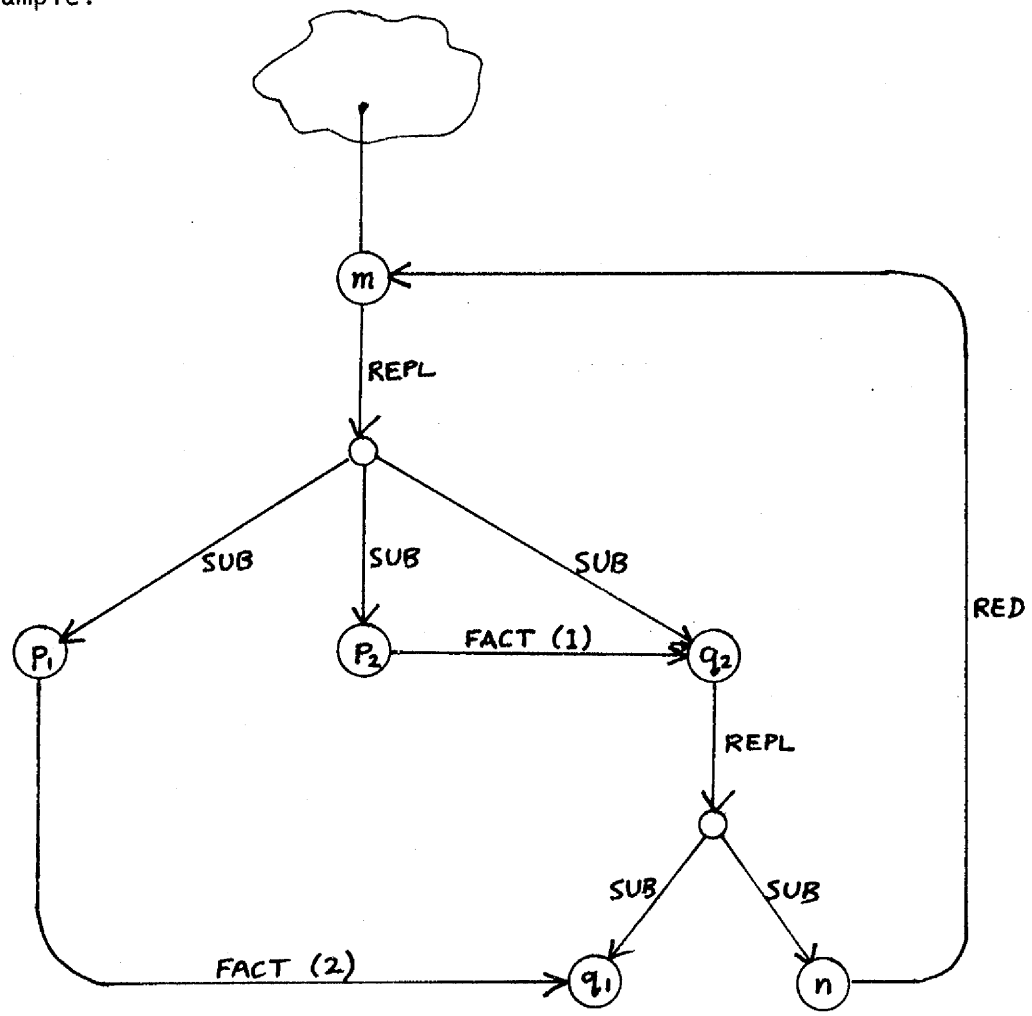
(i) $S(\Gamma_k) = S(\Gamma_{k-1}) \cup \{p\}$

(ii) $C(\Gamma_k) = C(\Gamma_{k-1}) \cup \{\{p, q\}\}$

(iii) $L(\Gamma_k) = L(\Gamma_{k-1}) - \{p\}$

We say that p is factored to q

Example:



If we assume that the two FACT arcs were the last arcs constructed, then FACT (2) is an example of simple factoring, and FACT (1) is an example of back factoring.

3.1 Some notes on $\mathcal{S}$-graphs

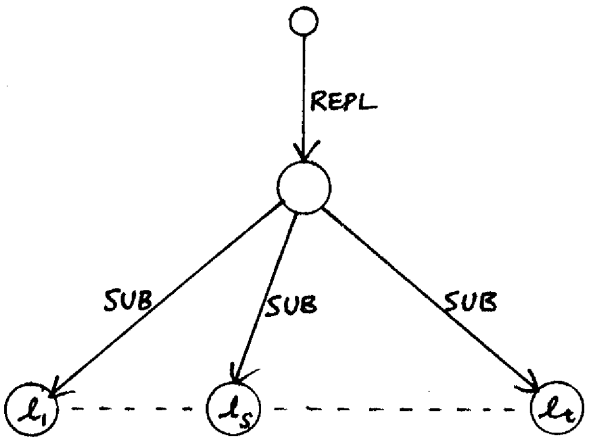
Definition: If Γ is an $\mathcal{S}$-graph, the clause $L(\Gamma)\sigma(\Gamma)$ is called the clause deduced by Γ.

If $L(\Gamma) = \phi$, then $L(\Gamma)\sigma(\Gamma)$ is $\square$, the empty clause, and Γ is said to be closed.

Every node of an $\mathcal{S}$- graph, except TOP, has in-degree 1; TOP has in-degree 0. Leaves have out-degree 0, and solved nodes have out-degree 1,

in fact, solved nodes are former leaves, each of which has been moved from $L(\Gamma)$ to $S(\Gamma)$ by one application of a rule.

We defined a sub-$\mathcal{S}$-graph of an $\mathcal{S}$-graph Γ to be any subgraph of Γ which is also a $\mathcal{S}$-graph. This implies that TOP $\in N(\Gamma')$, if Γ' is a sub-$\mathcal{S}$-graph of Γ . Also, if $(n, \text{REPL}, m) \in A(\Gamma')$, then every arc of the form (m, SUB, p) in $A(\Gamma)$ must also be in $A(\Gamma')$. Note, however, that if $\mathcal{S}$ were to contain clauses $\mathcal{C}_1 = \{m, \ell_1, \ldots, \ell_s\}$ and $\mathcal{C}_2 = \mathcal{C}_1 \cup \{\ell_{s+1}, \ldots, \ell_t\}$, then this condition would not hold, since $\mathcal{C}_2$ could be used for replacement with the literal m, thus:



In this case we could include arcs $\{(m, \text{SUB}, \ell_i) \mid i=1,\ldots,s\}$ in $A(\Gamma')$ but exclude the other SUB arcs out of m. In effect, we are doing a replacement with $\mathcal{C}_1$ in Γ' rather than with $\mathcal{C}_2$ as in Γ. Since, however, the omission of a subsumed clause from a set $\mathcal{S}$ of clauses does not alter the satisfiability or unsatisfiability of $\mathcal{S}$, we can assume that the above condition on sub-$\mathcal{S}$-graphs holds.

In every $\mathcal{S}$-graph, there are nodes which are neither solved nodes nor leaves: TOP is one of these, and the others are the literals in clauses from $\mathcal{S}$ which are used in applying the replacement operation to leaves.

If $\mathcal{S}$ contains the empty clause, $\square$, then we may use $\square$ as the top clause for constructing an $\mathcal{S}$-graph, and we obtain the graph:

$$\Gamma = \langle\{TOP\}, \phi, T\rangle$$

for which $L(\Gamma) = \phi$. So that Γ is a closed $\mathcal{S}$-graph.

Rules (2) and (3) for the construction of $\mathcal{S}$-graphs are intimately related. In the absence of rule (3), condition (d) on rule (2) can be removed. Similarly, in the absence of rule (2), the or part of condition (d) on rule (3)B can be removed.

Although rule (3)B, back factoring, allows greater flexibility in constructing $\mathcal{S}$-graphs, it is unnecessary in that we can construct exactly the same graphs without it. More formally, we have:

Lemma 3.1.1: If Γ is any $\mathcal{S}$-graph, then Γ can be generated by rules (1), (2) and (3)A.

3.2 Soundness and completeness of deduction using $\mathcal{S}$-graphs

Initially we restrict our attention to $\mathcal{S}$-graphs constructed using only two of the rules presented above: namely, reduction and simple replacement. We will quote without proof, lemmas establishing the equivalance of deduction using such restricted $\mathcal{S}$-graphs, with the model elimination deduction system due to Loveland [13, 14, 16]. A brief account of the latter system is now presented.

3.2.1 Model elimination

Definition: A chain is an ordered finite set of literals. The literals of a chain K are divided into two disjoint sets, A_K and B_K, the elements of which are termed A-literals and B-literals of K, respectively.

Definition: An elementary chain is one containing no A-literals.

Definition: A chain is preadmissible iff:

(i) any two complementary B-literals are separated by an A-literal;

(ii) no B-literal identical to an A-literal appears to the right of the A-literal;

(iii) no two A-literals are identical or complementary.

Definition: A chain is admissible if it is preadmissible and its last literal is a B-literal. The empty chain is defined to be admissible.

Definition: If M is a set of elementary chains, a finite sequence $K_0, K_1, \ldots, K_n$ of chains is called an ME-deduction of K_n from M if $K_0 \in M$, and for $i=1,\ldots,n$, K_i is derived from K_{i-1} by one of the following rules:

(i) Extension

If K_{i-1} is admissible and the last literal ℓ_1 of K_{i-1} is unifiable with $\tilde{\ell}_2$ with mgu σ, where ℓ_2 is a literal of some chain $K \in M$; derive K_i from K_{i-1} by deleting $\ell_2\sigma$ from $K\sigma$, and concatenating the remaining chain to the end of $K_{i-1}\sigma$. Each literal in K_i which derives from K_{i-1} inherits the classification of its parent literal in K_i, except for $\ell_1\sigma$, which is an A-literal. All literals which come from chain K are B-literals.

(ii) Reduction

If K_{i-1} is admissible and the last literal ℓ_1 unifies with $\tilde{\ell}_2$, where $\ell_2 \in A_{K_{i-1}}$, and if σ is an mgu of $\{\ell_1 \quad \tilde{\ell}_2\}$, then K_i is formed by deleting the

last literal from K_{i-1} σ. Each literal in K_i inherits the classification of its parent literal in K_{i-1}.

(iii) Contraction:

If K_{i-1} is not admissible, derive K_i by deleting all A-literals that follow the last B-literal in K_{i-1}. The classification of the remaining literals is as in K_{i-1}. Note that K_i may be empty.

Definition: If $\mathcal{S}$ is a set of clauses, then any set of chains obtained by imposing some ordering on each clause of $\mathcal{S}$, is called a matrix set of $\mathcal{S}$.

Definition: A clause $\mathcal{L}$ is said to be ME-deducible from a set of clauses $\mathcal{S}$, if there exists an ME-deduction $K_0,\ldots,K_n$ from some matrix set M of $\mathcal{S}$, such that $\mathcal{L} = B_{K_n}$.

Theorem 3.2.1.1: (Loveland) $\mathcal{S}$ is unsatisfiable iff the empty clause $\square$ is ME-deducible from $\mathcal{S}$. This is proved in [14].

3.2.2 The soundness and completeness of $\mathcal{S}$-graph deduction

Lemma 3.2.2.1: If a clause $\mathcal{L}$ is ME-deducible from $\mathcal{S}$, then there exists an $\mathcal{S}$-graph Γ, constructed using rules (1)A and (2) only, such that $L(\Gamma)\sigma(\Gamma) = \mathcal{L}$.

Lemma 3.2.2.2: If there exists a closed $\mathcal{S}$-graph, constructed using (1)A and (2) only, then there is an ME-deduction of the empty clause $\square$ from $\mathcal{S}$.

Combining these two lemmas, we obtain the following:

Theorem 3.2.2.3: $\mathcal{S}$ is unsatisfiable iff there exists a closed $\mathcal{S}$-graph, constructed using rules (1)A and (2) only.

We now extend the soundness property to general $\mathcal{S}$-graph deduction through the following two lemmas.

Lemma 3.2.2.4: If there exists a closed $\mathcal{S}$-graph, constructed using rules (1)A, (2) and (3) only, then $\mathcal{S}$ is unsatisfiable.

Lemma 3.2.2.5: If there exists a closed $\mathcal{S}$-graph, then $\mathcal{S}$ is unsatisfiable.

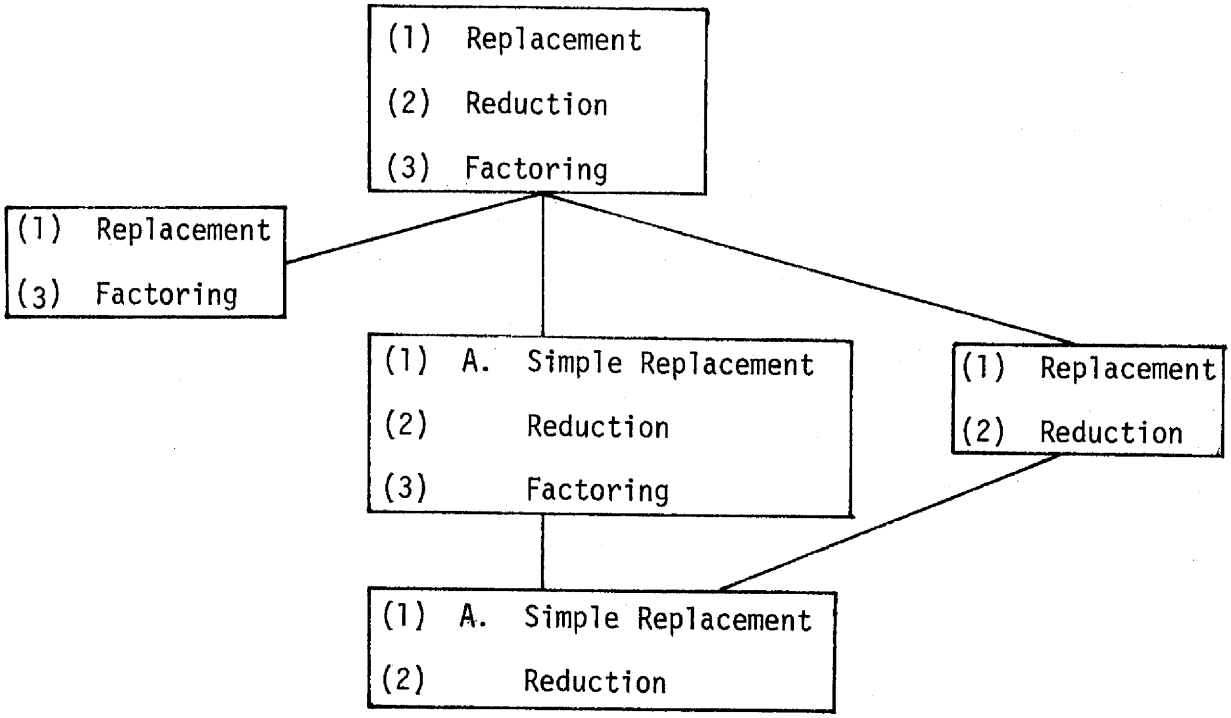
3.2.2.1 Minimal complete sets of rules

As has already been shown in lemma 3.2.2.1, $\mathcal{S}$-graph deduction with rules (1)A and (2) is complete. Obviously, no subset of this set of rules provides us with a complete deduction system.

There is one other subset of the rules which provides us with a complete deduction system, and is minimal in the sense that we lose completeness by taking any subset of it: namely, rules (1) and (3). Rules (1)A and (3) together provide a deduction system which is equivalent to ordinary linear resolution without the use of centre clauses as side clauses, which is not complete. Similarly, rule (1) alone is equivalent to binary linear resolution, with the use of centre clauses as side clauses; and of course binary resolution is not complete. Rules (1) and (3), however, provide a system which is equivalent to ordinary linear resolution, which is complete.

3.2.2.2 Summary of soundness and completeness results

The results of this section are summarised by the following partial-ordering of $\mathcal{S}$-graph deduction systems. The completeness of lower members implies the completeness of higher ones; the soundness of higher members implies soundness of lower ones.



4. CONSTRAINT PROCESSING

In this section, we demonstrate the need for an efficient method of processing the constraint set $C(\Gamma)$ of an $\mathcal{S}$-graph Γ. We then digress with a discussion of unification algorithms, presenting an algorithm which is particularly suited to our purposes, and show how it may be modified to allow detection of arcs in the $\mathcal{S}$-graph which block the solution of later subproblems.

Example 4.1

Let $\mathcal{S}$ be the set of clauses:

$-P(y, a), -P(y, w), -P(w, y)$

$P(x, a), \quad P(x, f(x))$

$P(z, a), \quad P(f(z), z)$

In this, and all following examples, we adopt the convention that lower case letters from the end of the alphabet, with or without subscripts,

denote variables, while those from the beginning of the alphabet denote constants.

Figure 4.1.1 illustrates a closed $\mathcal{S}$-graph. The arcs which introduce constraints are numbered, and the corresponding constraints (Figure 4.1.2) are labelled with these numbers.

The constraint set of Figure 4.1.2 is unifiable, so the set $\mathcal{S}$ is unsatisfiable. This can be verified by applying one of several well-known unification algorithms to the set of constraints. We have, however, presented the above $\mathcal{S}$-graph as a *fait accompli*, ignoring the fact that at each application of a rule we must check that the constraint set is still unifiable when the new constraint is added. Obviously, we would prefer not to re-unify the whole set at each additon. Similarly, if, during the construction of the $\mathcal{S}$-graph, it had been necessary to remove some arcs from the graph, we would prefer not to re-unify the set after the removal of the corresponding constraints. Furthermore, if ununifiability occurs at some stage, we need to be able to determine the cause in order to remedy the situation: most existing unification algorithms merely return the answer "ununifiable" in such cases.

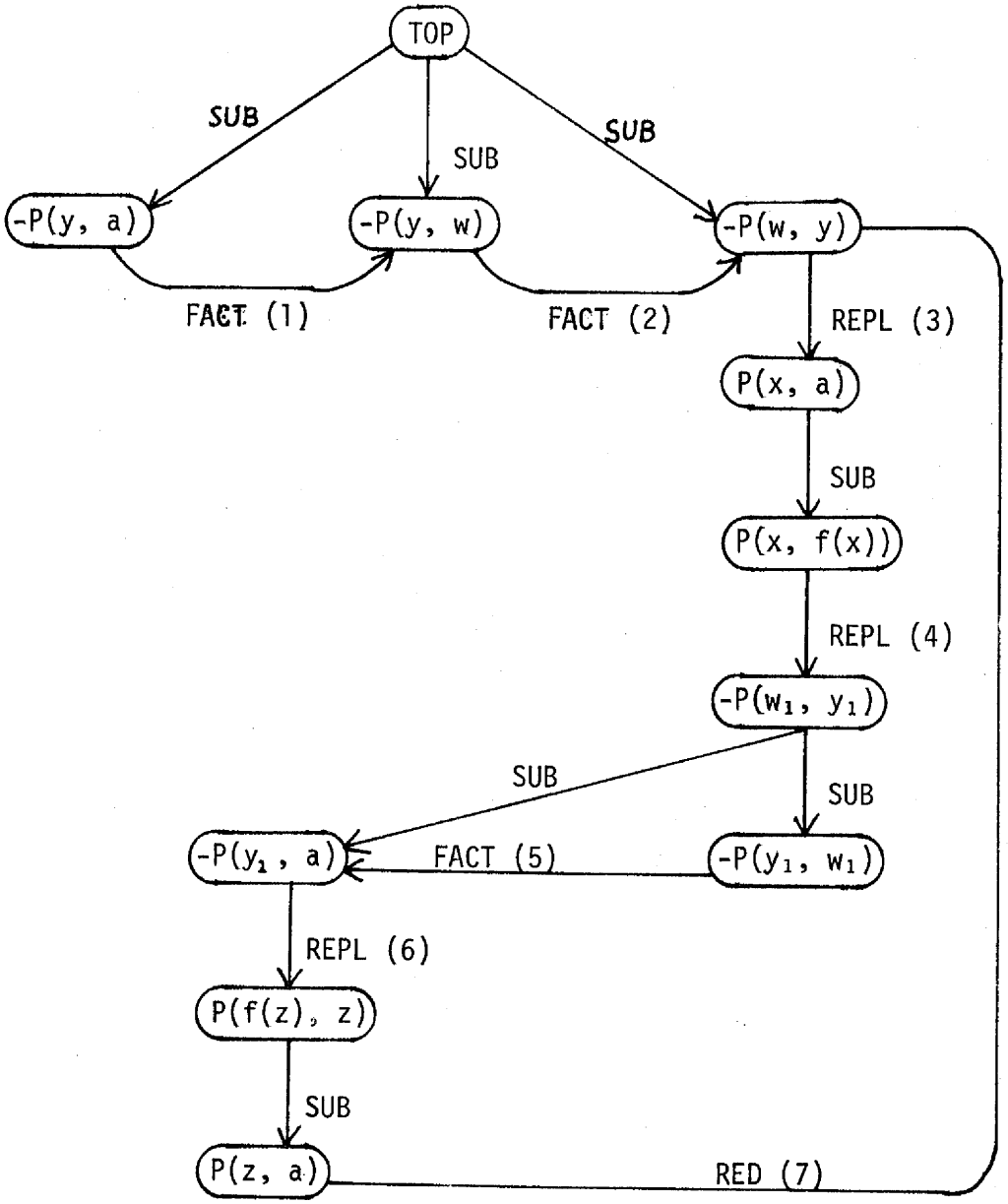
Obviously, the utility of $\mathcal{S}$-graph deduction depends heavily on efficient processing of the constraint set, and with this in mind, we now digress a little with a discussion of unification algorithms.

4.1 Unification

As has already been mentioned in section 2.3, unification is the process of determining whether two expressions have a common instance, and is an integral part of any system which makes logical deductions.

Robinson [21] gave an algorithm for determining whether or not two expressions are unifiable, and for producing their most general unifier.

Figure 4.1.1



A closed $\mathcal{S}$- graph for the set $\mathcal{S}$ of example 4.1

Figure 4.1.2

1: {-P(y, a), -P(y, w)}

2: {-P(y, w), -P(w, y)}

3: { P(w, y), P(x, a)}

4: { P(x, f(x)), P(w_1, y_1)}

5: {-P(y_1, w_1), -P(y_1, a)}

6: { P(y_1, a), P(f(z))}

7: { P(z, a), P(w,y)}

The constraint set for the $\mathcal{S}$-graph of Figure 4.1.1

Unfortunately, this algorithm is an exponential one; that is, its execution time and storage requirements depend exponentially on the length of the expressions under consideration. This inefficiency has long plagued theorem-proving programs.

Since the introduction of Robinson's algorithm, various improvements have been made [2, 3, 20, 22, 24], and at present there exist two efficient algorithms. One of these [20] is linear, and the other [3] is almost linear, its execution time being proportional to nG(n), where n is the sum of the lengths of the input expressions. G(n) is an almost constant function of n, being, for example, less than 5 when n is less than 2^{65536}.

The Baxter algorithm is particularly suitable for our purposes, since it allows us to add new constraints to a constraint set which has already been unified, with a minimal amount of reprocessing. The Paterson and Wegman algorithm, on the other hand, requires that the new set resulting from the addition of a new constraint be entirely reunified; this is an obvious waste of time. The report [3] contains a detailed description of the Baxter algorithm; however, a short description is included here so that

we may describe how the algorithm may be modified to suit our purposes.

4.1.1 The Baxter unification algorithm

The algorithm divides unification into two stages, the transformational stage and the sorting stage.

The transformational stage takes as input a set of pairs of expressions to be simultaneously unified; a set of constraints in fact. The output is a collection of sets of expressions. In the following abstract description, we use S to denote the set of pairs of expressions yet to be unified, and F to denote the current output collection of expressions.

The following algorithm is taken directly from [3], where S is initially S_I, the original constraint set to be unified; F is initially the set $\{\{e\} \mid e \text{ is a subexpression from } S_I\}$

repeat until S is empty:
begin delete any pair $\{e_1, e_2\}$ from S;
let $e_1 \in T_1$ and $e_2 \in T_2$ where
$T_1, T_2 \in F$;
if T_1 contains a term $f'(e'_1,\ldots,e'_n)$
and
T_2 contains a term $f''(e''_1,\ldots,e''_m)$
then if $f' \neq f''$
then UNIFICATION FAILS
else add to S the pairs
$\{e'_1,e''_1\},\ldots,\{e'_n,e''_n\}$
merge T_1 and T_2, that is replace
T_1 and T_2 by $T_1 \cup T_2$ in F;
end.

Note that constants are treated as 0-ary functions.

We now give a simple example to illustrate the above algorithm.

Example 4.2

Let S_I = {{x, f(y, z)}, {x, f(g(a), g(y))}, {y, g(w)}}

Then the table of figure 4.2.1 demonstrates the application of the algorithm. The numbers in the columns labelled "added" and "deleted" indicate the cycle at which sets are added to and deleted from S and F.

The sorting stage of the algorithm takes the output set F_o from the transformational stage and constructs a directed graph G from it. The nodes of G are the elements of F_o, and the arcs are constructed as follows: if $T \in F_o$, select one term of the form $f(e_1,\ldots,e_n)$ from T (i.e., a term which is not a variable), and for each i=1,...,n construct one directed edge from T to T_i, where $e_i \in T_i$.

If the resulting graph can be topologically sorted, then F_o (and S_I) is unifiable, and we can construct the most general unifier from the resulting partial order. Otherwise F_o (and S_I) is unifiable.

The directed graph constructed from the output set of example 4.2 is illustrated in figure 4.2.2. Figure 4.2.3 shows the result of the topological sort of this graph, and figure 4.2.4 gives the most general unifier obtained from the partial order.

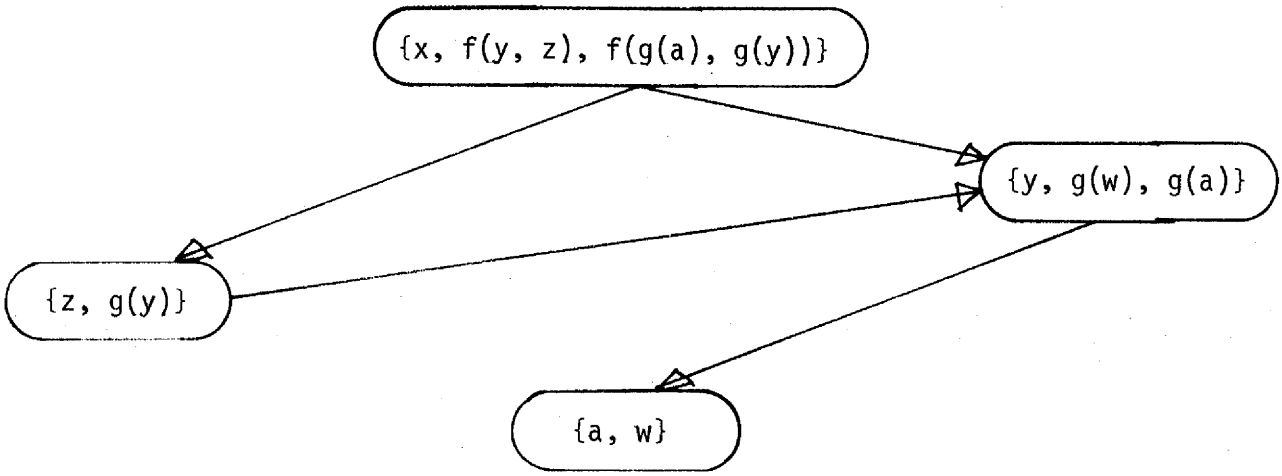
Figure 4.2.1

ADDED	DELETED	F
0	1	{x}
0	1	{f(y, z)}
0	3	{y}
0	5	{z}
0	2	{f(g(a), g(y))}
0	4	{g(a)}
0	6	{a}
0	5	{g(y)}
0	3	{g(w)}
0	6	{w}
1	2	{x, f(y, z)}
2		{x, f(y, z), f(g(a), g(y))}
3	4	{y, g(w)}
4		{y, g(w), g(a)}
5		{z, g(y)}
6		{a, w}

ADDED	DELETED	S
0	1	{x, f(y, z)}
0	2	{x, f(g(a), g(y))}
0	3	{y, g(w)}
2	4	{y, g(a)}
2	5	{z, g(y)}
4	6	{w, a}

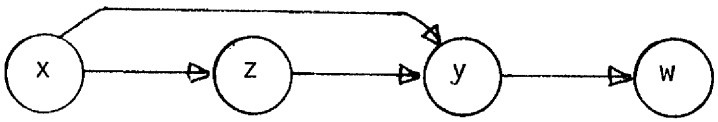
The algorithm terminates with S being empty, and F containing the four undeleted sets in the left column.

Figure 4.2.2



The directed graph constructed from the output collection of sets of example 4.2.

Figure 4.2.3



The linear ordering produced by the topological sort of the graph of figure 4.2.2.

Figure 4.2.4

The most general unifier for the set of constraints of example 4.2 is $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3 \circ \sigma_4$, where:

$$\sigma_1 = \{x \leftarrow f(y, z)\}$$

$$\sigma_2 = \{z \leftarrow g(y)\}$$

$$\sigma_3 = \{y \leftarrow g(w)\}$$

$$\sigma_4 = \{w \leftarrow a\}$$

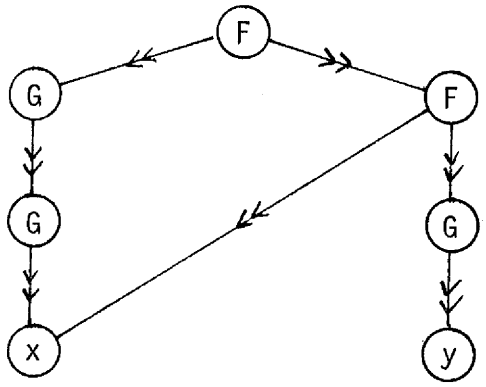
4.1.2 Data structures

We now describe the data structure used in [3], since it too is particularly suited to our purposes.

Expressions are represented as trees in which common variables are shared: the pointers from any vertex of an expression tree, point to the subexpressions of the expression represented by that vertex.

Example 4.3

Suppose F is a binary function symbol and G is a unary function symbol, then the expression F(G(G(x)), F(x, G(y))) is represented by the tree:



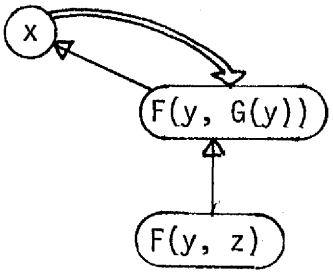
Obviously, the arcs from a vertex to its subexpressions are ordered, so that we know which argument is which. For instance, in the above example, the left descendent of the topmost (F) node is the first argument of the function F represented by that node.

We will consistently use the double-headed arrow →→ to denote subexpressions.

Sets of expressions are also represented as trees in which each vertex is an expression of the set, and the arcs join members of the set. In the unification algorithm, it is necessary to know if a set of expressions contains a term which is not a variable. Consequently, there is a special arc from the root of the tree to a particular non-variable term, if one exists.

Example 4.4

The set of expressions {x, F(y, G(y)), F(y, z)} is represented by the tree:

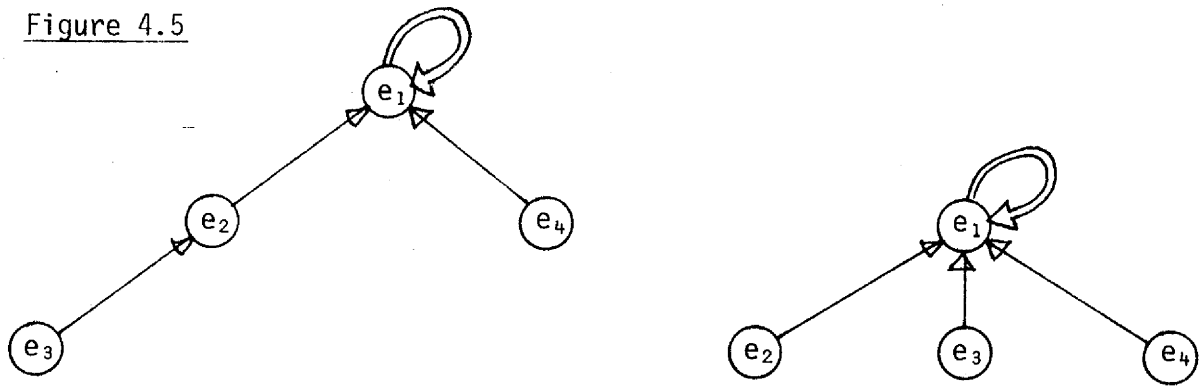


Note that such a representation is not unique. We have arbitrarily chosen the expression x as the root of the tree, and the expression F(y, G(y)) as the distinguished expression of the set.

We will use the single-headed arrow → to denote the membership of an expression in a set, and the double arrow ⇒ to denote the distinguished expression of a set.

The near-linearity of the unification algorithm depends on these trees of expressions being balanced: that is, having depth approximately equal to the logarithm of the number of nodes. For instance, although the trees of figure 4.5 both represent the same set, only the right-hand one is balanced.

Figure 4.5



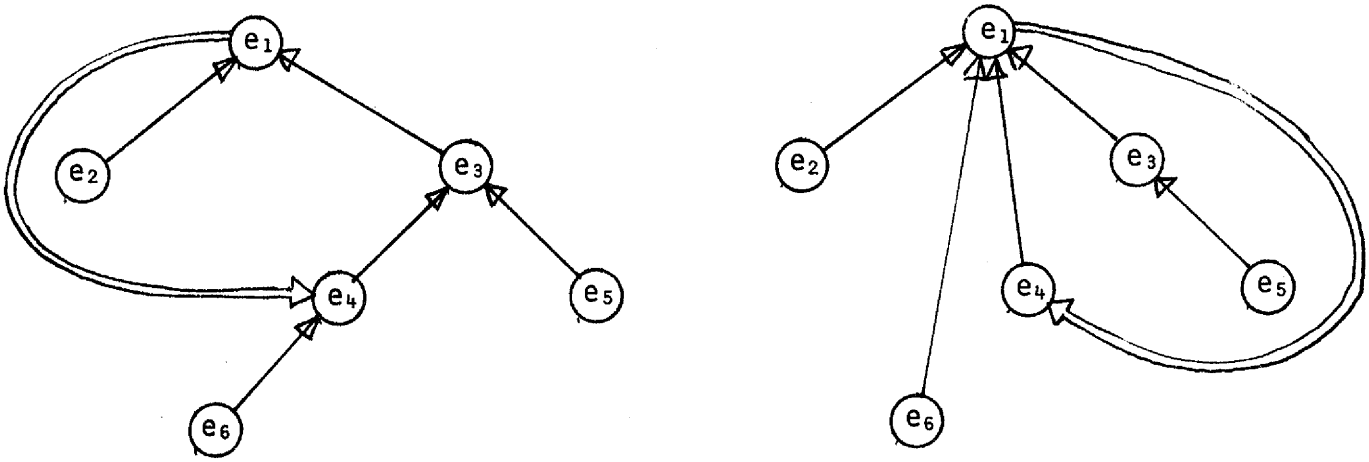
Unbalanced and balanced tree representations for the set of expressions $\{e_1, e_2, e_3, e_4\}$

4.1.3 The FIND and MERGE algorithms

These two procedures, which are fundamental to the transformational stage of the unification algorithm, are described in detail in [3]. We now give an informal description of both.

FIND locates the root of the tree to which a given expression belongs. It does this by tracing the path leading from the expression to the root. Also, in order to assist in balancing the trees, it replaces every arc it encounters with a new arc pointing directly to the root. This is illustrated in figure 4.6.

Figure 4.6

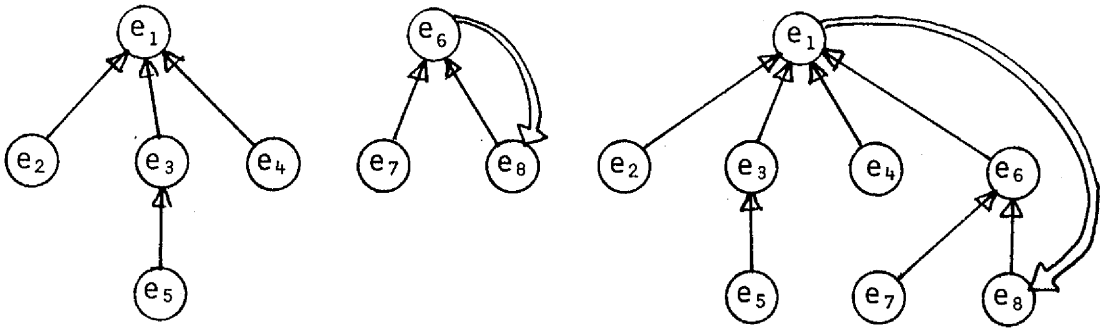


The left-hand tree is transformed into the right-hand tree during the FINDing of e_6.

MERGE connects two trees by creating a new arc directed from the root of the "smaller" tree to that of the "larger" tree, where the size of a tree is defined to be the number of nodes it contains. Arcs created by MERGE will from now on be referred to as "merge arcs". That is, every arrow of the type ⇢ is a merge arc. The distinguished expression of the new tree which results from a MERGE, is the distinguished expression of the larger input tree, or if that tree has no distinguished expression, it is the distinguished expression of the smaller tree. If neither input tree has a distinguished

expression, then neither does the new tree. MERGing is illustrated in figure 4.7.

Figure 4.7



The tree on the right results from MERGing the two trees on the left.

4.2 The use of the Baxter algorithm in constraint processing

After example 4.2, we discussed constraint processing, and presented the following criteria for a good constraint processing system:

(A) Reprocessing should be minimal following the addition of new constraints.

(B) The cause of any un-unifiability should be easy to find.

(C) Reprocessing should be minimal following the removal of constraints.

4.2.1 Criterion (A)

If we use the Baxter algorithm to process the constraint set, criterion (A) is satisfied, at least for the transformational stage of the algorithm. This is because the constraints are processed serially, so that the work done in unifying a set of constraints, adding a new constraint, then unifying it with the original set, is exactly the same as would have been done in unifying the whole set.

Similar economy can be obtained in constructing the directed graph

to be sorted in the sorting stage of the algorithm. If sets T_0 and T_1 in F are merged, yielding T_2, we obtain the new directed graph by:

(i) replacing all arcs of the form (T, T_0) and (T, T_1) by arcs of the form (T, T_2);

(ii) replacing all arcs of the form (T_i, T) by arcs of the form (T_2, T), where the distinguished term of T_2 is the distinguished term of T_i $(i=0, 1)$;

(iii) removing all arcs of the form (T_j, T) where $j=0$ if $i=1$ and vice versa.

This is illustrated in figure 4.8

Fortunately, we do not in general need to re-sort the whole directed graph. Suppose the linear ordering of the nodes of the current directed graph is:

$$T_0, T_1, \ldots, T_n$$

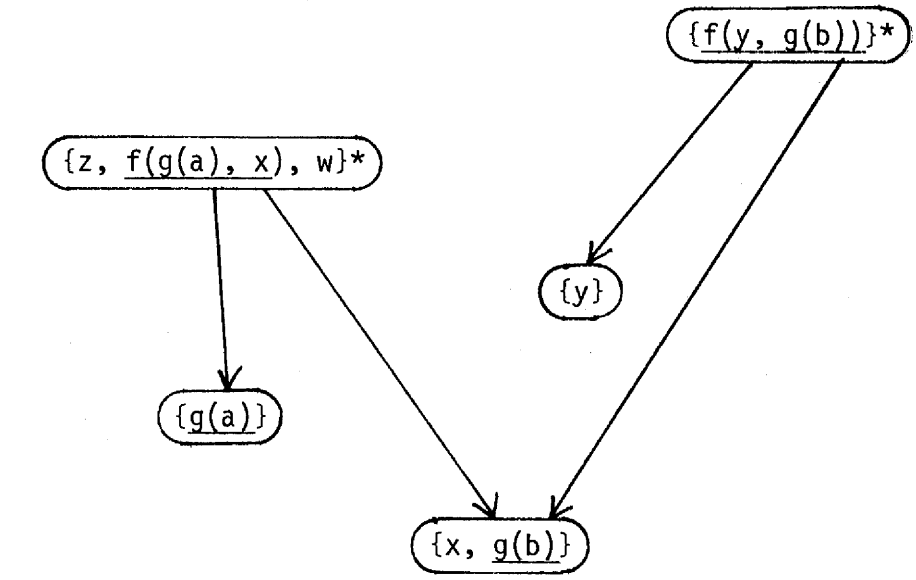
When a new constraint is added, the transformational stage of the algorithm will cause a number of merges to be performed. If T_i and T_j are respectively the least and greatest elements in the above linear ordering which are involved in these merges, then the new directed graph contains nodes $T_0, T_1, \ldots, T_{i-1}, T_i', \ldots, T_j', T_{j+1}, \ldots, T_n$.

Note that $\{T_i', \ldots, T_j'\}$ contains less elements than $\{T_i, \ldots, T_j\}$. Obviously, we need to sort only the subgraph consisting of nodes $\{T_i', \ldots, T_j'\}$, and arcs of the form (T_x, T_y), where T_x and T_y are in $\{T_i', \ldots, T_j'\}$.

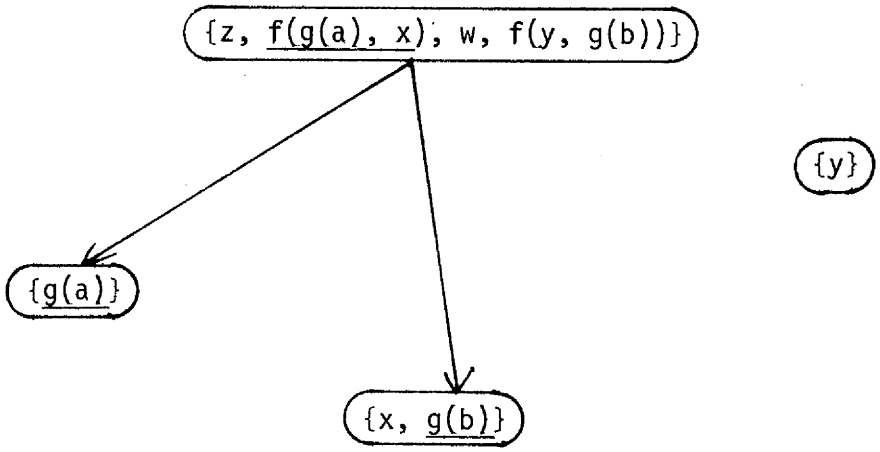
This is shown in figure 4.9.

In the example of figure 4.9, we see that more computation is involved in sorting graph (a) then (c), than in sorting graph (b) directly. It is hoped that further research will reduce the amount of extra processing required in the sorting stage when a new constraint is added.

Figure 4.8



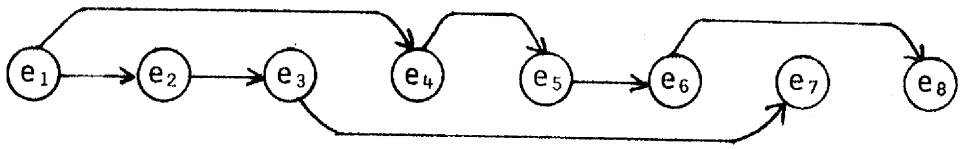
(a)



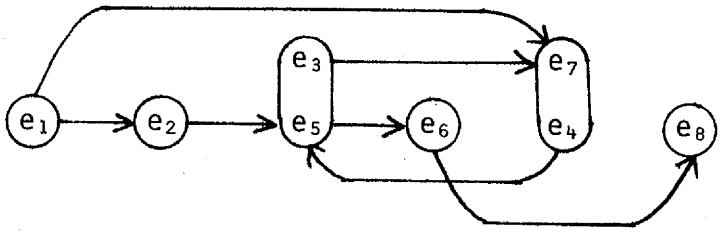
(b)

When the marked sets in (a) are merged, the directed graph (b) results. The distinguished term in each set is underlined. Note that this merge will cause the sets {g(a)} and {y} to be merged.

Figure 4.9

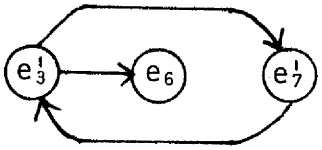


(a)



(b)

The directed graph (b) results from merging e_3 and e_5 , and e_4 and e_7 in graph (a).



(7)

Graph (c) is all that requires sorting.

4.2.2 Criterion(B)

Definition: If Γ is an $\mathcal{S}$-graph, we call the associated graph built by the transformational stage of the unification algorithm, the unification graph, $U(\Gamma)$. Note that $U(\Gamma)$ is not unique since its structure depends on the order in which $C(\Gamma)$ is processed (i.e., on the order in which nodes of Γ are solved).

Criterion (B) requires that we are able to locate the cause of any un-unifiability that may arise. To facilitate this, we will introduce a system of labels for the merge arcs of a unification graph. It is also necessary that in each set of expressions in F, the variables are kept separate from the other terms, which will be referred to as "non-variables" from now on. This requires some modifications to the transformational stage of the unification algorithm. The abstract description of the algorithm remains as before, but the **MERGE** procedure must be altered.

We will now assume that all trees of expressions either:

(i) have all variable nodes;

or (ii) have all non-variable nodes;

or (iii) have nodes of both types, and

- (a) the root is a variable;
- (b) the root has only one non-variable parent, which is the distinguished term;
- (c) the parents of any variable node other than the root, are variables;
- (d) the parents of any non-variable node are non-variables.

With these restrictions in mind, for any tree T of expressions, with root v(T) and distinguished term e(T), we will denote the tree of variable nodes rooted in v(T) by V(T); and the tree of non-variable nodes rooted in e(T) by E(T). Note that if T is of type (i), then V(T) = T, and E(T) is empty; similarly, for T of type (ii), V(T) is empty and E(T) = T.

If we use $MERGE_1$ to denote the old merging procedure, then the new procedure is defined as follows:

To MERGE T_1 and T_2

(1) Delete merge arcs $(e(T_1), v(T_1))$ and $(e(T_2), v(T_2))$

(2) $MERGE_1$ $V(T_1)$ and $V(T_2)$, obtaining V(T)

(3) $MERGE_1$ $E(T_1)$ and $E(T_2)$, obtaining E(T)

(4) Add merge arc (e(T), v(T))

(5) e(T) is the distinguished term of T

Figure 4.10 illustrates this procedure.

We must, of course, allow for the possibility that some of the trees in the above description are empty. The effect on the procedure is obvious, and will not be described here.

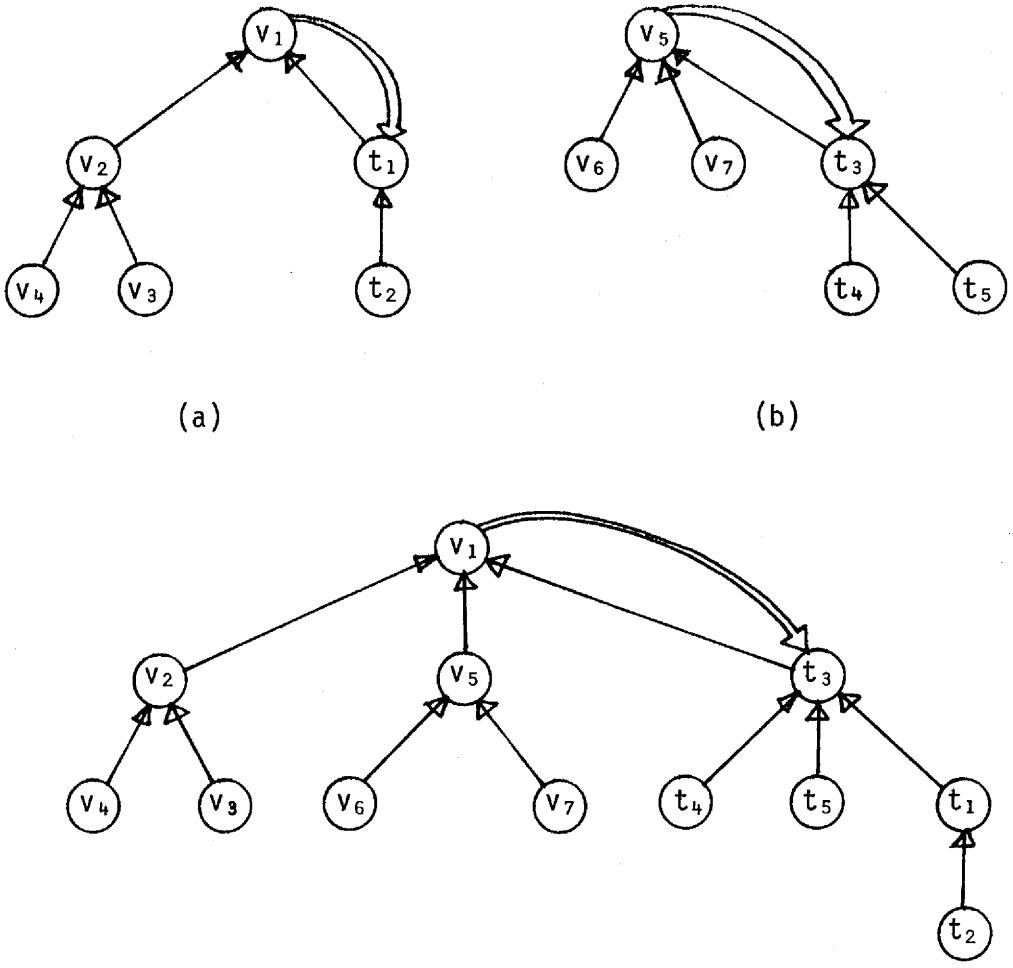
It should also be obvious that if the two trees being merged obey the restrictions described above, then so does the tree output by the MERGE procedure.

FIND also requires a minor modification. When FIND traverses the path $b_1, b_2, \ldots, b_n$ to root node b_n while FINDing b_1, then:

(1) if b_1 and b_n are either both variables or both non-variables, each arc (b_i, b_{i+1}), i=1,...,n-2 is replaced by arc (b_i, b_n)

(2) otherwise, each arc (b_i, b_{i+1}) i=1,...,n-3 is replaced by arc (b_i, b_{n-1}). Note that in this case, b_1 must be a non-variable.

Figure 4.10



MERGing (a) and (b) gives (c)

It is easy to see that this modified FIND procedure, like the modified MERGE, preserves the restricted form of the trees on which it operates.

The time bound for the modified algorithm is still of order nG(n) since the time required to execute a MERGE is constant if the time for a $MERGE_1$ is constant, and the FIND procedure still operates on balanced trees, so the time analysis of [1] quoted in [3] holds.

4.2.2.1 Merge arc labelling

From now on we assume that the algorithm does not delete constraints from S, but merely marks them as having been processed.

Definition: A primitive merge label is an ordered pair <a,b> or <a,a> where {a,b} is a constraint on S. Recall that S is the set of constraints processed in the construction of $U(\Gamma)$. If ℓ = <a,b> is a primitive merge label, we define:

$$\text{tail}\ (\ell) = a$$

$$\text{head}\ (\ell) = b$$

Definition: A merge label is

(i) the identity label 1;

or (ii) a primitive merge label

or (iii) $(\ell_1 + \ell_2)$ where ℓ_1 and ℓ_2 are merge labels such that:

$$\text{head}\ (\ell_1) = \text{head}\ (\ell_2), \text{ and}$$

$$\text{tail}\ (\ell_1) = \text{tail}\ (\ell_2):$$

in this case $\text{head}\ ((\ell_1 + \ell_2)) = \text{head}\ (\ell_1)$

and $\text{tail}\ ((\ell_1 + \ell_2)) = \text{tail}\ (\ell_1)$

or (iv) $(\ell_1 \cdot \ell_2)$ where ℓ_1 and ℓ_2 are merge labels such that:

$$\text{head } (\ell_1) = \text{tail } (\ell_2):$$

in this case $\text{head } ((\ell_1 \cdot \ell_2)) = \text{head } (\ell_2)$

and $\text{tail } ((\ell_1 \cdot \ell_2)) = \text{tail } (\ell_1)$

or (v) $\bar{\ell}$ where ℓ is a merge label:

in this case $\text{head } (\bar{\ell}) = \text{tail } (\ell)$

and $\text{tail } (\bar{\ell}) = \text{head } (\ell)$

Note that for each constraint, there are two primitive merge labels, which are "complementary": that is, if {a,b} is a constraint in S, then <a,b> and <b,a> are both primitive merge labels, and $\overline{\langle a,b \rangle} = \langle b,a \rangle$, $\overline{\langle b,a \rangle} = \langle a,b \rangle$.

The identity label, 1, has the property head (1) = tail (1).

Definition: If m and n are merge labels, then $m \equiv n$ iff m can be can be transformed into n by a finite number of applications of the transformation rules which follow. In each rule, the double arrow, $\leftrightarrow$, indicates that the transformation may be performed in either direction.

(i) Distributivity:

(a) $(\ell_1 \cdot (\ell_2 + \ell_3)) \leftrightarrow ((\ell_1 \cdot \ell_2) + (\ell_1 \cdot \ell_3)$

(b) $((\ell_1 + \ell_2) \cdot \ell_3) \leftrightarrow ((\ell_1 \cdot \ell_3) + (\ell_2 \cdot \ell_3))$

(ii) Associativity:

(a) $(\ell_1 + (\ell_2 + \ell_3)) \leftrightarrow ((\ell_1 + \ell_2) + \ell_3)$

(b) $(\ell_1 \cdot (\ell_2 \cdot \ell_3)) \leftrightarrow ((\ell_1 \cdot \ell_2) \cdot \ell_3)$

(iii) Complementation:

(a) $\overline{(\ell_1 + \ell_2)} \leftrightarrow (\bar{\ell}_1 + \bar{\ell}_2)$

(b) $\overline{(\ell_1 \cdot \ell_2)} \leftrightarrow (\bar{\ell}_2 \cdot \bar{\ell}_1)$

(c) $\overline{\bar{\ell}} \leftrightarrow \ell$

(iv) Commutativity:

$(\ell_1 + \ell_2) \leftrightarrow (\ell_2 + \ell_1)$

(v) Contraction:

If head (ℓ) = tail (ℓ)

then $\ell \leftrightarrow 1$

(vi) Identity:

(a) $(\ell \cdot \bar{\ell}) \leftrightarrow 1$

(b) $(1 \cdot \ell) \leftrightarrow \ell \leftrightarrow (\ell \cdot 1)$

(c) $(1 + \ell) \leftrightarrow 1$

(vii) Absorption:

$(\ell + \ell) \leftrightarrow \ell$

Lemma 4.2.2.1.1: $\equiv$ is an equivalence relation.

Example 4.11

$$(\langle b,a\rangle \cdot \overline{(\overline{\langle a,b\rangle} + (\langle b,c\rangle \cdot \overline{(\langle a,d\rangle \cdot \overline{\langle c,d\rangle})})))}$$

$$\equiv (\langle b,a\rangle \cdot (\overline{\overline{\langle a,b\rangle}} + \overline{(\langle b,c\rangle \cdot \overline{(\langle a,d\rangle \cdot \overline{\langle c,d\rangle})})})) \qquad \text{(iii) (a)}$$

$$\equiv (\langle b,a\rangle \cdot (\langle a,b\rangle + (\overline{\overline{(\langle a,d\rangle \cdot \overline{\langle c,d\rangle})}} \cdot \overline{\langle b,c\rangle}))) \qquad \text{(iii) (c) \& (b)}$$

$$\equiv (\langle b,a\rangle \cdot (\langle a,b\rangle + ((\langle a,d\rangle \cdot \langle d,c\rangle) \cdot \langle c,b\rangle)))$$

(iii) (c) and two primitive label complementations

$$\equiv ((\langle b,a\rangle \cdot \langle a,b\rangle) + (\langle b,a\rangle \cdot ((\langle a,d\rangle \cdot \langle d,c\rangle) \cdot \langle c,b\rangle))) \qquad \text{(i) (a)}$$

$$\equiv (1 + 1) \qquad \text{two applications of (v)}$$

$$\equiv 1 \qquad \text{by (vii) or (vi) (c)}$$

Note that this conclusion could have been reached immediately after the first transformation by noting that:

tail $(\overline{\overline{\langle a,b\rangle}}) = \text{head } (\langle b,a\rangle)$

and head $(\overline{\overline{\langle a,b\rangle}}) = \text{tail } (\langle b,a\rangle)$

Therefore, by the definition of $\cdot$

tail $(\ell) = \text{head } (\langle b,a\rangle)$

and head (ℓ) = tail (<b a>),

where ℓ is the second label in the product. So, by contraction, the entire label is equivalent to 1.

Because of associativity, we will henceforth write extended products and sums without brackets. Also we will assume precedence of $\cdot$ over +, so that products do not need to be bracketed in sums of products.

Definition: A merge label

$$\ell_{11}\cdot\ell_{12}\cdot\ldots\cdot\ell_{1n_1} + \ldots + \ell_{k1}\cdot\ell_{k2}\cdot\ldots\cdot\ell_{kn_k}$$

is said to be in canonical form iff:

(i) each ℓ_{ij} is primitive

(ii) no two products are identical

(iii) for each i=1,...,k there is no s and t such that $1 \leq s \leq t \leq n_i$, and tail (ℓ_{is}) = head (ℓ_{it})

Theorem 4.2.2.1.2: If ℓ is a merge label, there is a merge label k in canonical form such that $\ell \equiv k$, and k is unique modulo commutativity of +.

Definition: If S contains constraints

$$\{a_1, a_2\}, \{a_2, a_3\}, \ldots, \{a_{n-1}, a_n\}, \{a_n, a_1\}$$

such that $a_i \neq a_j$ if $i \neq j$, then the ordered n-tuple $(a_1, a_2, \ldots, a_n)$ is called a circuit of U(Γ), and each of the above constraints is called an adjacent pair of the circuit. Note that the above set of constraints actually gives rise to 2n circuits of U(Γ). If c_1 and c_2 are two circuits generated from the same set of constraints, we write $c_1 \sim c_2$.

Definition: If ℓ is a merge label and c is a circuit, we define $c(\ell)$ as follows:

(i) $c(1) = 1$

(ii) $c(\langle a,b\rangle) = \langle a,b\rangle$ if $\{a,b\}$ is not an adjacent pair of c

(iii) $c(\langle a_1, a_n\rangle) = (\langle a_1, a_n\rangle + \langle a_1, a_2\rangle \cdot \langle a_2, a_3\rangle \cdot \ldots \cdot \langle a_{n-1}, a_n\rangle)$

$c(\langle a_{i+1}, a_i\rangle) = (\langle a_{i+1}, a_i\rangle + \langle a_{i+1}, a_{i+2}\rangle \cdot \ldots \cdot \langle a_{n-1}, a_n\rangle \cdot \langle a_n, a_1\rangle \cdot \ldots \cdot \langle a_{i-1}, a_i\rangle)$

where $c = (a_1, \ldots, a_n)$

(iv) $c(\ell_1 + \ell_2) = c(\ell_1) + c(\ell_2)$

$c(\ell_1 \cdot \ell_2) = c(\ell_1) \cdot c(\ell_2)$

$c(\bar{\ell}) = \overline{c(\ell)}$

Lemma 4.2.2.1.3: If c is a circuit and ℓ is a merge label, head $(c(\ell))$ = head (ℓ) and tail $(c(\ell))$ = tail (ℓ)

Lemma 4.2.2.1.4: For any circuits c_1, c_2 of U (Γ):

$c_1 \sim c_2$ iff $c_1(\ell) = c_2(\ell)$ for all merge labels ℓ.

Corollary 4.2.2.1.5: $\sim$ is an equivalence relation on the set of circuits of $U(\Gamma)$. We denote the set of equivalence classes under $\sim$ by $C_{U(\Gamma)}$.

From now on, we use the term "set of circuits" to mean "set of equivalence classes of circuits". Consequently, we call $C_{U(\Gamma)}$ the set of all circuits of $U(\Gamma)$.

Definition: If K is a set of circuits, we define $K(\ell)$ for any merge label ℓ as follows:

If K is empty, $K(\ell) = \ell$

otherwise (i) $K(1) = 1$

(ii) $K(\langle a,b\rangle) = \sum_{c \in K} c(\langle a,b\rangle)$

(iii) $K(\ell_1 + \ell_2) = K(\ell_1) + K(\ell_2)$

$K(\ell_1 \cdot \ell_2) = K(\ell_1) \cdot K(\ell_2)$

$K(\bar{\ell}) = \overline{K(\ell)}$

Lemma 4.2.2.1.6: If ℓ and p are two merge labels such that $\ell \equiv p$, then $K(\ell) \equiv K(p)$, where K is any set of circuits.

Example 4.12

$$\overline{\overline{\langle a,b\rangle} + \langle b,c\rangle \cdot \langle a,d\rangle \cdot \overline{\langle c,d\rangle}} \equiv \langle a,b\rangle + \langle a,d\rangle \cdot \langle d,c\rangle \cdot \langle c,b\rangle$$

Let k be the cycle (c, b, e, d)

Then:

$$\mathbf{k}(\text{R.H.S.}) = \langle a,b\rangle + \langle a,d\rangle\cdot[\langle d,c\rangle + \langle d,e\rangle\cdot\langle e,b\rangle\cdot\langle b,c\rangle]\cdot[\langle c,b\rangle + \langle c,d\rangle\cdot\langle d,e\rangle\cdot\langle e,b\rangle]$$

and

$$\mathbf{k}(\text{L.H.S.}) = \overline{\overline{\langle a,b\rangle} + [\langle b,c\rangle + \langle b,e\rangle\cdot\langle e,d\rangle\cdot\langle d,c\rangle]\cdot\langle a,d\rangle\cdot\overline{[\langle c,d\rangle + \langle c,b\rangle\cdot\langle b,e\rangle\cdot\langle e,d\rangle]}}$$

$$\equiv \langle a,b\rangle + \langle a,d\rangle \cdot \overline{[\langle c,d\rangle + \langle c,b\rangle\cdot\langle b,e\rangle\cdot\langle e,d\rangle]}\cdot\overline{[\langle b,c\rangle + \langle b,e\rangle\cdot\langle e,d\rangle\cdot\langle d,c\rangle]}$$

$$\equiv \langle a,b\rangle + \langle a,d\rangle \cdot[\langle d,c\rangle + \langle d,e\rangle\cdot\langle e,b\rangle\cdot\langle b,c\rangle]\cdot[\langle c,b\rangle + \langle c,d\rangle\cdot\langle d,e\rangle\cdot\langle e,b\rangle]$$

$$= \mathbf{k}(\text{R.H.S.})$$

Definition: Let M be the set of all merge labels. A constraint label is any subset of $M \cup A'(\Gamma)$, where $A'(\Gamma) \subseteq A(\Gamma)$ is the set of all FACT, RED, and REPL arcs of Γ.

If x is any merge arc or constraint, we will use label(x) to refer to its label. We will also use (a,b) to denote the merge arc joining node a to node b.

If b is an expression, we will refer to the tree in $U(\Gamma)$ to which it belongs as t(b); we will also abbreviate v(t(b)) and e(t(b)) to v(b) and e(b) respectively. So v(b) and e(b) are the variable root and distinguished

term of the tree containing b. We also define:

$$r(b) = \begin{cases} v(b) \text{ if } b \text{ is a variable} \\ e(b) \text{ otherwise.} \end{cases}$$

The labelling procedure

As Γ and $U(\Gamma)$ are constructed, we also construct a set $K_{U(\Gamma)}$ of circuits of $U(\Gamma)$, and label the merge arcs of $U(\Gamma)$ and the constraints of S. We assume that $K_{U(\Gamma)}$ is initially empty: we also adopt the conventions that if s is a constraint not already in S, then label(s) is empty, and that label $((x,y))$ is 1 if $x = y$. The labelling procedure is as follows:

(1) If $\{a,b\}$ is placed in S because of the addition of arc x to Γ (where x is a FACT, RED, or REPL arc) then label($\{a,b\}$) := label($\{a,b\}$) $\cup$ $\{x\}$.

(2) If FIND is applied to an expression b_1, causing the addition of new merge arcs $(b_1, b_n), (b_2, b_n), \ldots, (b_{n-2}, b_n)$ to $U(\Gamma)$, then for each $i=1,\ldots,n-2$

$$\text{label}((b_i, b_n)) := \text{label}((b_i, b_{i+1})) \cdot \text{label}((b_{i+1}, b_{i+2})) \cdot \ldots \cdot \text{label}((b_{n-1}, b_n))$$

(3) If $\{a,b\}$ is selected for processing from S, then:

(i) if $t(a) \neq t(b)$, then t(a) and t(b) will be MERGED. Suppose $v(x)$ is MERGED_1 to $v(y)$ and that $e(u)$ is MERGED_1 to $e(w)$, where $u, w, x, y \in \{a,b\}$, and $u \neq w$, $x \neq y$.

Then:

(a) $$\text{label}((v(x),v(y))) := \overline{\text{label}((x,r(x))) \cdot \text{label}((r(x),v(x)))} \cdot \langle x,y \rangle \cdot \overline{\text{label}((y,r(y))) \cdot \text{label}((r(y),v(y)))}$$

(b) $$\text{label}((e(u),e(w))) := \overline{\text{label}((u,r(u))) \cdot \overline{\text{label}((e(u),r(u)))}} \cdot \langle u,w \rangle \cdot \overline{\text{label}((w,r(w))) \cdot \overline{\text{label}((e(w),r(w)))}}$$

(c) $$\text{label}((e(w),v(y))) := \overline{\text{label}((w,r(w))) \cdot \overline{\text{label}((e(w),r(w)))}} \cdot \langle w,y \rangle \cdot \overline{\text{label}((y,r(y))) \cdot \text{label}((r(y),v(y)))}$$

(ii) if $t(a) = t(b)$, let k be the canonical form of:

$$K_{U(\Gamma)} \quad (\text{label}((a,r(a))) \cdot \langle r(a), r(b)\rangle \cdot \overline{\text{label}((b,r(b)))})$$

then:

$$K_{U(\Gamma)} := K_{U(\Gamma)} \cup \{[(a, a_1, \ldots, a_n, b)] \mid \langle a, a_1\rangle \cdot \langle a_1, a_2\rangle \cdot \ldots \cdot \langle a_{n-1}, a_n\rangle \cdot \langle a_n, b\rangle \text{ is a product in k}\}$$

where [ ] denotes an equivalence class under $\sim$

(4) If $\{a,b\}$ is added to S because of merging trees T_1 and T_2, during which the arc $(e(T_1), e(T_2))$ is created, then if we abbreviate this new arc as (e_1, e_2):

$$\text{label}(\{a,b\}) := \text{label}(\{a,b\}) \cup \{\text{label}((e_1, e_2))\}$$

Example 4.13

Let $C(\Gamma)$ be the set of constraints:

$\{w, e_1\}$	$\{a_1\}$
$\{u, e_1\}$	$\{a_2\}$
$\{e_2, e_3\}$	$\{a_3\}$
$\{y, x\}$	$\{a_4\}$
$\{x, z\}$	$\{a_5\}$
$\{x, u\}$	$\{a_6\}$
$\{z, e_2\}$	$\{a_7\}$

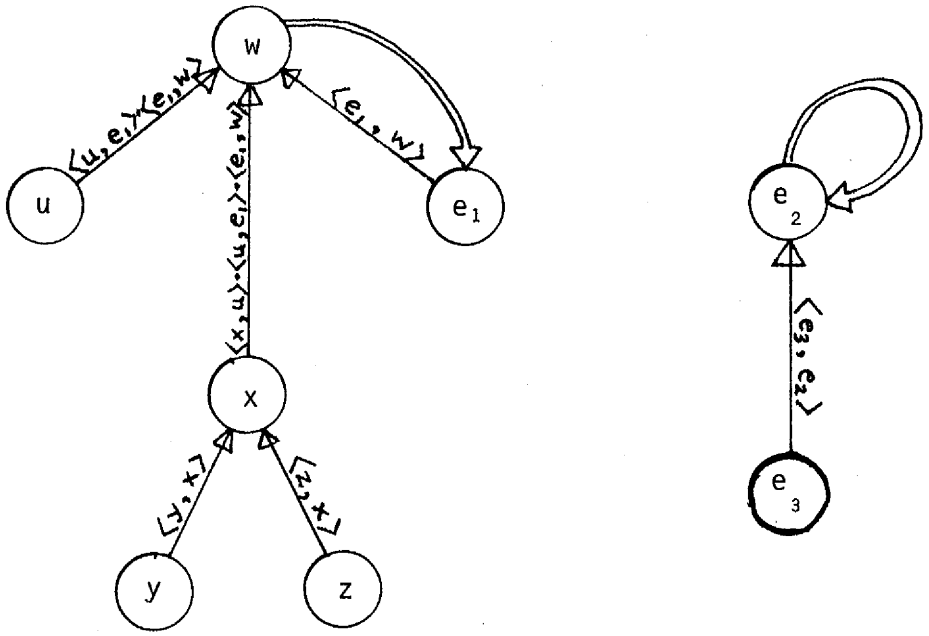
where $e_1 = f(w, z)$, $e_2 = f(x, x)$, $e_3 = f(y, y)$ and $a_1, \ldots, a_7$ are the arcs in Γ which give rise to the constraints. The sequence of diagrams of figure 4.13.1 illustrates the construction of $U(\Gamma)$, the labelling of constraints and merge arcs, and the construction of $K_{U(\Gamma)}$.

$\Gamma_0, \ldots, \Gamma_7 = \Gamma$ is the sequence of $\mathcal{S}$-graphs generated by the construction of the arcs $a_1, \ldots, a_7$.

Figure 4.13.1

The construction of $U(\Gamma)$ from the constraints of example 4.13 is illustrated here. In the sequence of diagrams, a merge arc is labelled only the first time it appears.

(a) (i) $U(\Gamma_6)$



(ii) S: *{w, e_1} {a_1}

*{u, e_1} {a_2}

*{e_2, e_3} {a_3}

*{y, x} {a_4, <e_2, e_3>}

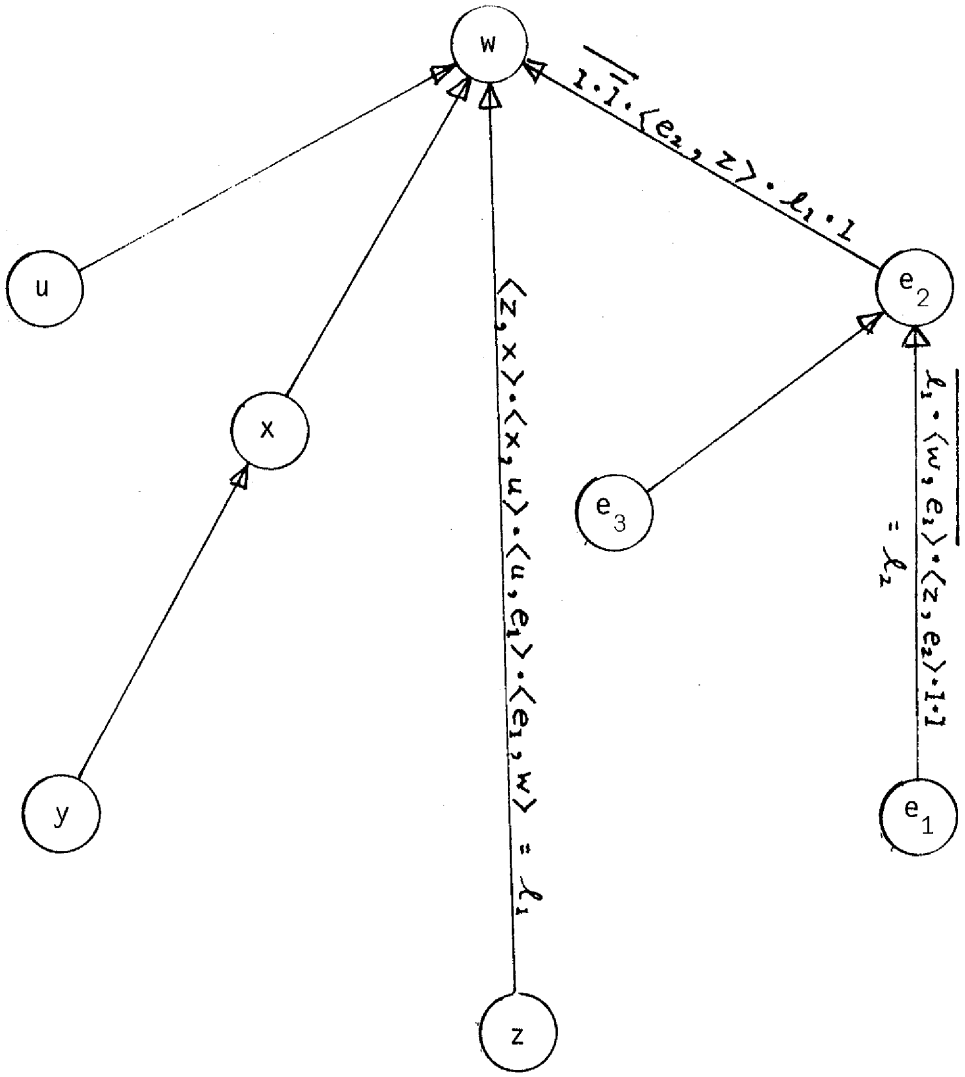
*{x, z} {a_5}

*{x, u} {a_6}

*indicates that the constraint has been processed.

(iii) $K_{U(\Gamma_6)} = \phi$

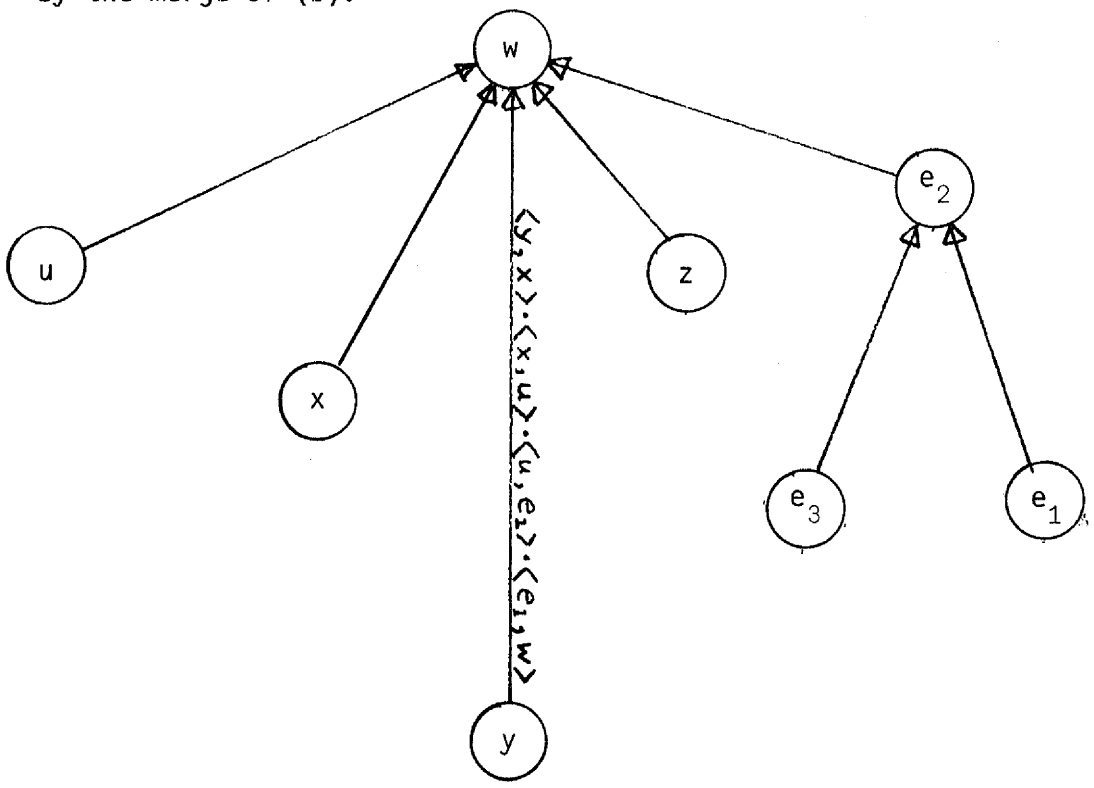
(b) (i) Status of the unification graph after performing the merge indicated by the last input constraint.



(ii) S is as in (a) (ii) with the following additions:

*	$\{z, e_2\}$	$\{a_7\}$
	$\{w, y\}$	$\{\ell_2\}$
	$\{z, y\}$	$\{\ell_2\}$

(c) (i) Status of graph after processing the first constraint produced by the merge of (b).



(ii) The first seven elements of S are as in (b) (ii); the rest are:

* {w, y} $\{\ell_2\}$

{z, y} $\{\ell_2\}$

(iii) This is case (3) (ii) of the labelling procedure. We have:

$K_{U(\Gamma)}$ (label((w,r(w))) · <r(w),r(y)> · $\overline{\text{label}((y,r(y)))}$

= $1 \cdot \langle w,w\rangle \cdot \overline{\langle y,x\rangle \cdot \langle x,u\rangle \cdot \langle u,e_2\rangle \cdot \langle e_1,w\rangle}$

$\equiv \langle w,e_1\rangle \cdot \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,y\rangle$

$\therefore K_{U(\Gamma)} = \{(w, e_1, u, x, y)\}$

(d) (i) $U(\Gamma_7)$ is the same as the graph in (c) (i).

(ii) S is as in (b) (ii) except all constraints are marked as processed.

(iii) Again we have case (3) (ii) of the labelling procedure, and:

$K_{U(\Gamma)}$ (label((z, r(z))) $\cdot$ <r(z), r(y)> $\cdot$ $\overline{\text{label}((y,r(y)))}$)

$\equiv K_{U(\Gamma)}$ (<z, x> $\cdot$ <x,u> $\cdot$ <u,e_1> $\cdot$ <e_1,w> $\cdot$ <w,w> $\cdot$ <w,e_1> $\cdot$ <e_1,u> $\cdot$ <u,x> $\cdot$ <x,y>)

$\equiv K_{U(\Gamma)}$ (<z, x> $\cdot$ <x,y>)

$= \langle z, x\rangle \cdot [\langle x,y\rangle + \langle x,u\rangle \cdot \langle u,e_1\rangle \cdot \langle e_1,w\rangle \cdot \langle w,y\rangle]$

$\equiv \langle z, x\rangle \cdot \langle x,y\rangle + \langle z,x\rangle \cdot \langle x,u\rangle \cdot \langle u,e_1\rangle \cdot \langle e_1,w\rangle \cdot \langle w,y\rangle$

$\therefore K_{U(\Gamma_7)} + \{(w, e_1, u, x, y), (z, x, y), (z, x, u, e_1, w, y)\}$

Lemma 4.2.2.1.7: If (a, b) is a merge arc of $U(\Gamma)$ then:

head (label(a, b)) = b

and

tail (label(a, b)) = a

Lemma 4.2.2.1.8: $K_{U(\Gamma)} = C_{U(\Gamma)}$

Theorem 4.2.2.1.9: If (a,b) is a merge arc of $U(\Gamma)$ let k be the canonical form of $C_{U(\Gamma)}$ (label((a,b))). Then $\langle a_0, a_1\rangle \cdot \ldots \cdot \langle a_{n-1}, a_n\rangle$ is a product in k iff there exist constraints $\{a_0, a_1\},\ldots,\{a_{n-1}, a_n\}$ in S where $a_0 = a$, $a_n = b$ and $a_i \neq a_j$ if $i \neq j$.

Corollary 4.2.2.1.10: As for the above theorem but with $C_{U(\Gamma)}$ replaced by $K_{U(\Gamma)}$.

Example 4.14

Consider the merge arc (e_1, e_2) of $U(\Gamma_7)$ in example 4.13

label $((e_1, e_2)) = \overline{\ell_1 \cdot \langle w,e_1\rangle} \cdot \langle z,e_2\rangle \cdot 1 \cdot 1$

$\equiv \langle e_1,w\rangle \cdot \langle w,e_1\rangle \cdot \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,z\rangle \cdot \langle z,e_2\rangle$

$\equiv \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,z\rangle \cdot \langle z,e_2\rangle$

$$\therefore K_{U(\Gamma)}(\text{label}((e_1, e_2))) \equiv [\, \langle e_1,u\rangle + \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,x\rangle \cdot \langle x,u\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle x,z\rangle \cdot \langle z,x\rangle \cdot \langle x,u\rangle \,]$$
$$\cdot [\, \langle u,x\rangle + \langle u,e_1\rangle \cdot \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,x\rangle$$
$$+ \langle u,e_1\rangle \cdot \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle \cdot \langle z,x\rangle \,]$$
$$\cdot [\, \langle x,z\rangle + \langle x,y\rangle \cdot \langle y,z\rangle$$
$$+ \langle x,u\rangle \cdot \langle u,e_1\rangle \cdot \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle \,]$$
$$\cdot \langle z,e_2\rangle$$
$$\equiv [\, \langle e_1,u\rangle \cdot \langle u,x\rangle + \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,x\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle \cdot \langle z,x\rangle \,]$$
$$\cdot [\, \langle x,z\rangle + \langle x,y\rangle \cdot \langle y,z\rangle$$
$$+ \langle x,u\rangle \cdot \langle u,e_1\rangle \cdot \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle \,]$$
$$\cdot \langle z,e_2\rangle$$
$$\equiv \cdot [\, \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,z\rangle + \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,y\rangle \cdot \langle y,z\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,x\rangle \cdot \langle x,y\rangle \,]$$
$$\cdot \langle z,e_2\rangle$$
$$\equiv \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,z\rangle \cdot \langle z,e_2\rangle$$
$$+ \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,y\rangle \cdot \langle y,z\rangle \cdot \langle z,e_2\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle \cdot \langle z,e_2\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,x\rangle \cdot \langle x,z\rangle \cdot \langle z,e_2\rangle$$

The reader should check each of the products in this canonical form to satisfy himself that the corresponding set of constraints is actually in S, and that there exists no set of constraints in S defining any other path between e_1 and e_2.

The above theorem shows that we have partially achieved our goal of modifying the Baxter algorithm to satisfy criterion (B). It says that if we process the label on a merge arc with the set of circuits generated during

the labelling of $U(\Gamma)$, and convert the result to canonical form, we can determine exactly which constraints must be removed from S in order to remove each path between the nodes that the merge arc connects. This is still not sufficient for our purposes since not all the constraints in S are placed there as the result of a new arc being constructed in Γ, so that we are still unable to determine from a merge arc label, exactly which arcs of Γ must be removed in order to delete the merge arc from $U(\Gamma)$.

Definition: Let $B(X)$ be the set of all Boolean functions over a set of variables X. We now define a function Ω: $M \to B(\{a_1,\ldots,a_n\})$ where M is the set of merge labels, and $\{a_1,\ldots,a_n\}$ is the set of RED, REPL and FACT arcs of Γ. Ω is defined as follows:

(i) $\Omega(1) = 0$

(ii) $\Omega(\langle a,b\rangle) = a_{i_1} \cdot a_{i_2} \cdot \ldots \cdot a_{i_j} \cdot \Omega(\ell_1) \cdot \ldots \cdot \Omega(\ell_k)$

where label $(\{a,b\}) = \{a_{i_1},\ldots,a_{i_j}, \ell_1,\ldots,\ell_k\}$

$\Omega(\langle a,a\rangle) = 0$

(iii) $\Omega((\ell_1 + \ell_2)) = \Omega(\ell_1) \cdot \Omega(\ell_2)$

(iv) $\Omega((\ell_1 \cdot \ell_2)) = \Omega(\ell_1) + \Omega(\ell_2)$

(v) $\Omega(\bar{\ell}) = \Omega(\ell)$

Definition: If ℓ is the merge label of arc (a,b), the Boolean expression $\Omega(K_{U(\Gamma)}(\ell))$ is called the covering expression of ℓ, and also, the covering expression of (a,b).

Definition: By the usual techniques for manipulating Boolean expressions (see [5]), we reduce the covering expression of a label ℓ to a unique sum of products of the form:

$$\sum_{i=1}^{n} a_{i1} \cdot a_{i2} \cdot \ldots \cdot a_{ik_i}$$

(where Σ denotes Boolean sum) such that

(i) for each i, $a_{ir} = a_{is} \Rightarrow r = s$ and

(ii) for each i, there is no j such that

$$\{a_{j1},\ldots,a_{jk_j}\} \subseteq \{a_{i1},\ldots,a_{ik_i}\}$$

This sum-of-products Boolean expression is called the reduced covering expression of ℓ, and also the reduced covering expression of (a,b), if label((a,b)) = ℓ.

Theorem 4.2.2.1.11: If ℓ_1 and ℓ_2 are two merge labels, let E_1 and E_2 be their reduced covering expressions, then $\ell_1 \equiv \ell_2 \Rightarrow E_1 = E_2$, modulo commutativity of Boolean sum and product.

Since we are ultimately interested in the reduced covering expressions of labels, rather than the labels themselves, this theorem allows us to simplify merge labels at our convenience.

We now state the central result of this section.

Theorem 4.2.2.1.12: If (a,b) is a merge arc of $U(\Gamma)$, and

$\sum_{i=1}^{n} a_{i1} \cdot \ldots \cdot a_{ik_i}$ is the reduced covering expression of (a,b),

then (a,b) is deleted from $U(\Gamma)$ iff the arcs $a_{i1},\ldots,a_{ik_i}$ are deleted from Γ for some $i \in \{1,\ldots,n\}$

Example 4.15

Using the unification graph $U(\Gamma_7)$ constructed in example 4.13, we have, from example 4.14:

$$K_{U(\Gamma_7)}\ (\text{label}((e_1, e_2))) \equiv \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,z\rangle \cdot \langle z,e_2\rangle$$
$$+ \langle e_1,u\rangle \cdot \langle u,x\rangle \cdot \langle x,y\rangle \cdot \langle y,z\rangle \cdot \langle z,e_2\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,z\rangle \cdot \langle z,e_2\rangle$$
$$+ \langle e_1,w\rangle \cdot \langle w,y\rangle \cdot \langle y,x\rangle \cdot \langle x,z\rangle \cdot \langle z,e_2\rangle$$

Now $\Omega(\langle e_1,u\rangle) = a_2$

$\Omega(\langle u,x\rangle) = a_6$

$\Omega(\langle x,z\rangle) = a_5$

$\Omega(\langle z,e_2\rangle) = a_7$

$\Omega(\langle x,y\rangle) = a_4 \cdot \Omega(\langle e_2,e_3\rangle) = a_4 \cdot a_3$

$\Omega(\langle y,z\rangle) = \Omega(\ell_2)$

$= \Omega(\langle e_1,w\rangle \cdot \bar{\ell}_1 \cdot \langle z,e_2\rangle)$

$= a_1 + \Omega(\bar{\ell}_1) + a_7$

$= a_1 + \Omega(\ell_1) + a_7$

$= a_1 + \Omega(\langle z,w\rangle \cdot \langle x,u\rangle \cdot \langle u,e\rangle \cdot \langle e_1,w\rangle) + a_7$

$= a_1 + a_5 + a_6 + a_2 + a_1 + a_7$

$= a_1 + a_2 + a_5 + a_6 + a_7$

$\Omega(\langle e_1,w\rangle) = a_1$

$\Omega(\langle w,y\rangle) = \Omega(\ell_2)$

$= a_1 + a_2 + a_5 + a_6 + a_7$

Now let k be the canonical form of $K_{U(\Gamma_7)}\ (\text{label}((e_1,e_2)))$.

Then $\Omega(k) = (a_2 + a_6 + a_5 + a_7)$

$\cdot\ (a_2 + a_6 + a_4 \cdot a_3 + (a_1 + a_2 + a_5 + a_6 + a_7) + a_7)$

$\cdot\ (a_1 + (a_1 + a_2 + a_5 + a_6 + a_7) + (a_1 + a_2 + a_5 + a_6 + a_7) + a_7)$

$\cdot\ (a_1 + (a_1 + a_2 + a_5 + a_6 + a_7) + a_4 \cdot a_3 + a_5 + a_7)$

$= a_2 + a_5 + a_6 + a_7$

But by theorem 4.2.2.1.11, since k ≡ label((e_1, e_2)), this Boolean expression is the reduced covering expression of label ((e_1, e_2)). Consequently, by theorem 4.2.2.1.12, removing any of the arcs a_2, a_5, a_6 or a_7 from Γ_7 will cause the deletion of (e_1, e_2), and possibly some other arcs of $U(\Gamma_7)$

4.2.2.2 Using merge arc labelling

If, during the construction of an $\mathcal{S}$-graph Γ, we encounter a leaf n, which has become unsolveable, we can use the labelling of merge arcs as described above, to determine which arcs of Γ must be removed in order that the attempted solution of n will work. Obviously, we do not wish to remove any arcs of Γ which solve direct ancestors of n, since n itself will then disappear. In order to keep track of these arcs, we label every leaf and solved node of Γ with a Boolean expression from $B(\{a_1,\ldots,a_k\})$ where $a_1,\ldots,a_k$ are the REPL arcs of Γ, as follows:

label(n) := 0, if (TOP, SUB, n) ε $A(\Gamma)$

otherwise

label (n) := a + label (m)

where (m, REPL, x)

and (x, SUB, n) ε $A(\Gamma)$

for some x ε $N(\Gamma)$

Now suppose the solution for n being attempted, causes the transformational stage of the algorithm to build a merge arc $(f(e_1,\ldots,e_s), g(t_1,\ldots,t_r))$ where $f \neq g$. Now the reduced covering expression of this arc is of the form $a + \sum_{i=1}^{k} a_{i1}\cdot\ldots\cdot a_{ij_i}$, where a is the FACT, RED or REPL arc which supposedly solves n, and $a_{ip} \neq a$ for any i and p. The expression must have this form since the constraint set was unifiable before the addition of arc a to Γ. We now obtain a new Boolean expression E, thus:

(i) Let $E_1 = \left(\sum_{i=1}^{k} a_{i1}\cdot\ldots\cdot a_{ij_i}\right) \cdot (\text{label}(a))'$

where ' denotes Boolean complementation.

(ii) Let E_2 = sum of products form of E_1, with all products containing complementary literals deleted.

(iii) Let E = E_2 with all negated variables deleted.

Say $E = \sum_{i=1}^{p} b_{i1}\cdot\ldots\cdot b_{ir_i}$

Deleting all the arcs $b_{i1},\ldots,b_{ir_i}$ for some i will remove the ununifiability without removing n from Γ, so that the current attempt at solving n will succeed.

Example 4.16

Let $\mathcal{S}$ be the set of clauses:

(i) P(x, y), R(y), R(b)

(ii) -P(u, v), -Q(f(a)), -Q(f(u))

(iii) Q(w), -P(z, w)

(iv) -R(s)

where a and b are constants.

Then the graph of figure 4.16.1 is an $\mathcal{S}$-graph.

For simplicity, we reduce the constraints by removing the predicate symbols, and unifying the corresponding arguments. Hence we have the following set of constraints and labels for the $\mathcal{S}$-graph of figure 4.16.1:

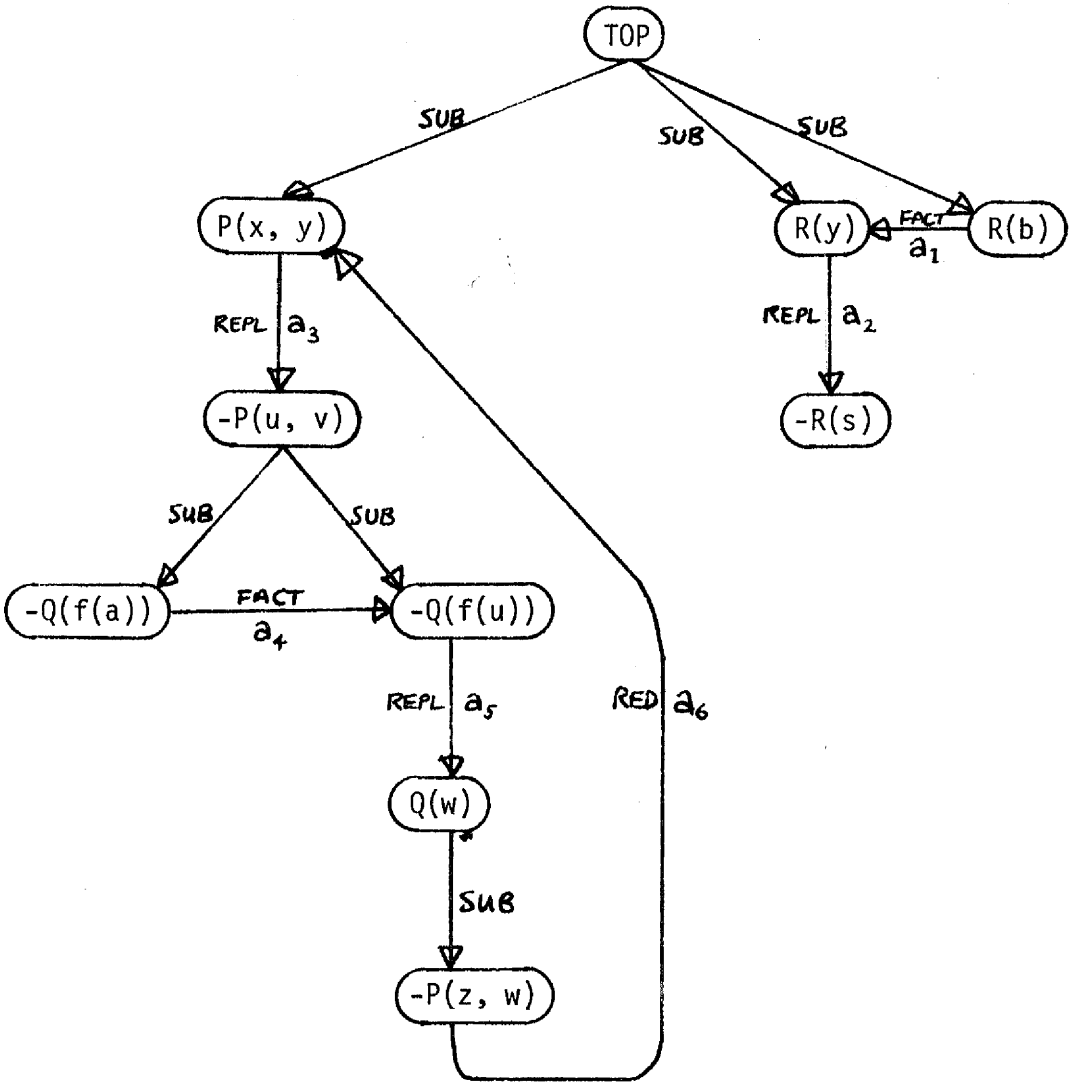
{y, b}	$\{a_1\}$
{y, s}	$\{a_2\}$
{x, u}	$\{a_3\}$
{y, v}	$\{a_3\}$
{f(a), f(u)}	$\{a_4\}$
{f(u), w}	$\{a_5\}$

{z, x} {a_6}

{w, y} {a_6}

The construction of the unification graph is illustrated in figure 4.16.2.

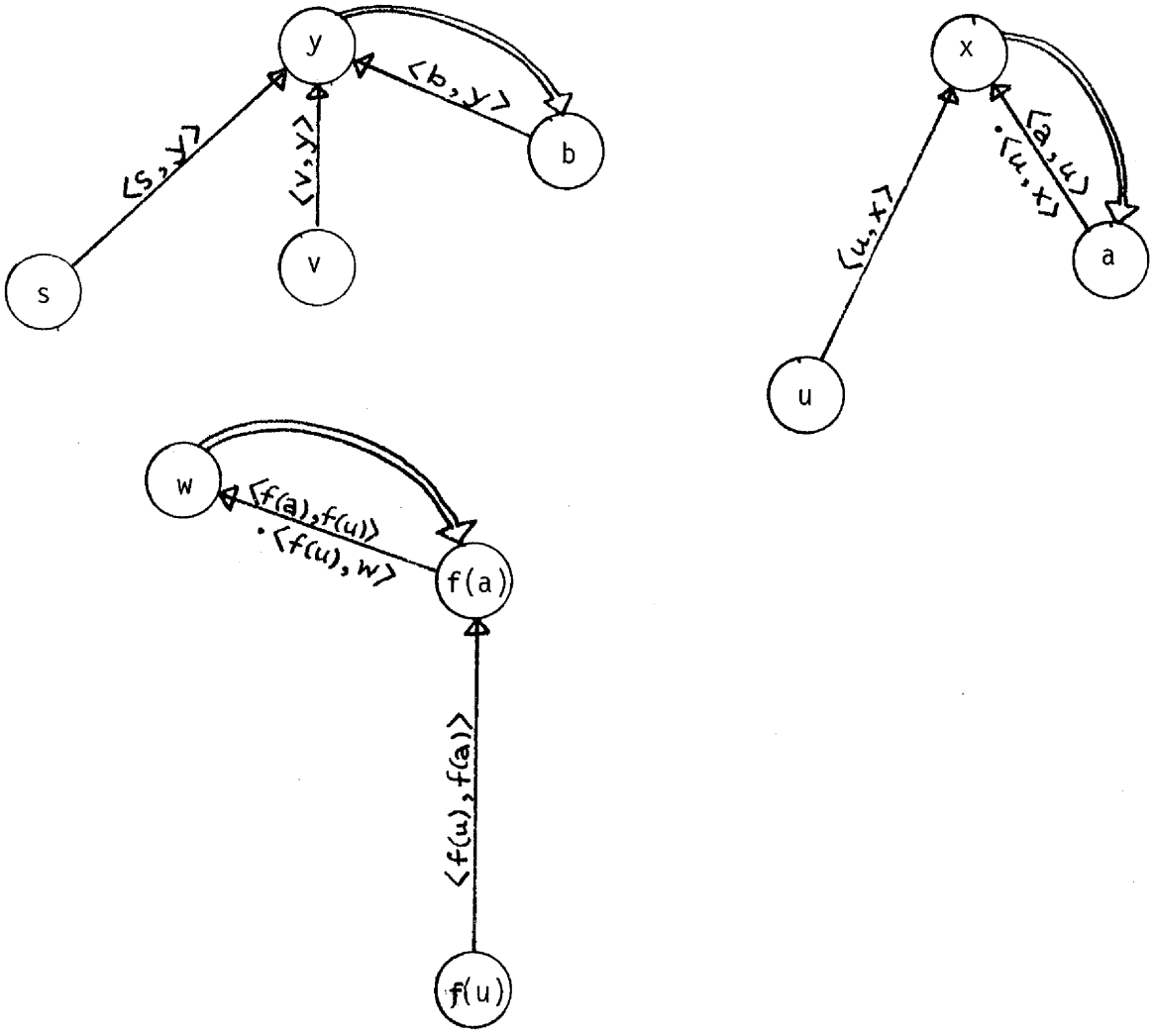
Figure 4.16.1



An $\mathcal{S}$-graph for $\mathcal{S}$ of example 4.16.

Figure 4.16.2

(a) The unification graph $U(\Gamma_5)$, where Γ_5 is the $\mathcal{S}$-graph of figure 4.16.1 with arc a_6 deleted.

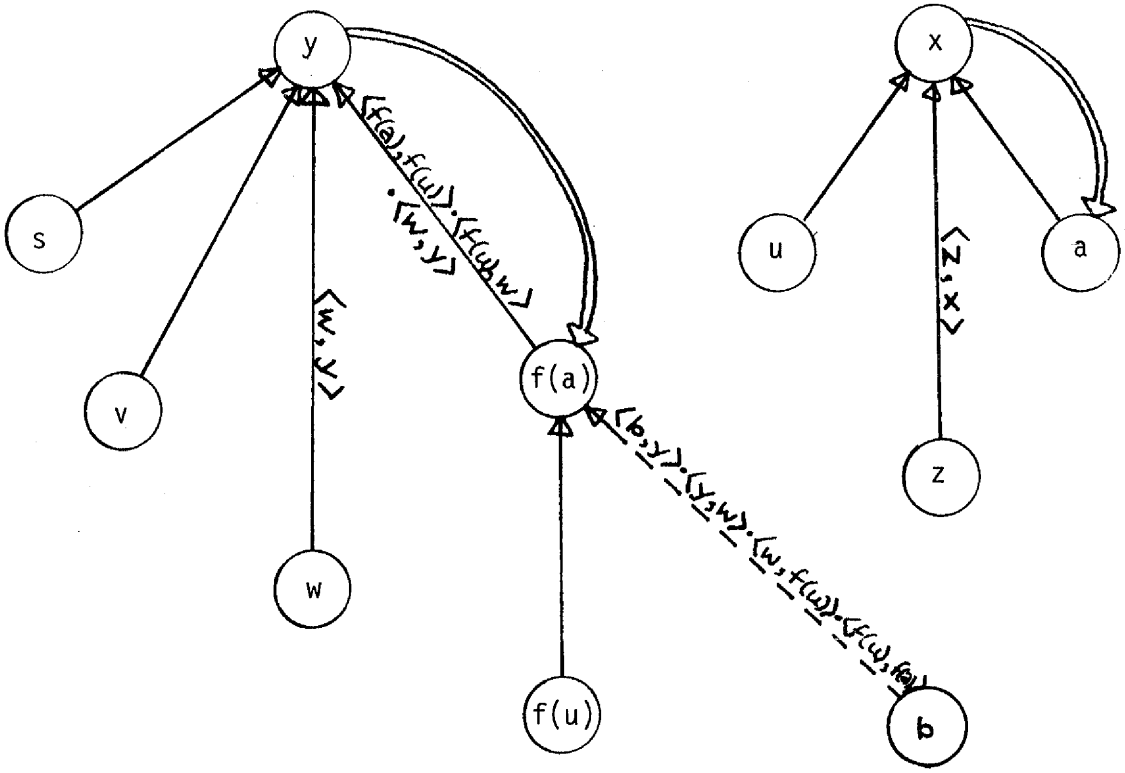


(b) S for $U(\Gamma_5)$

*{y, b}	$\{a_1\}$
*{y, s}	$\{a_2\}$
*{x, u}	$\{a_3\}$
*{y, v}	$\{a_3\}$

*{f(a), f(w)}	{a_4}
*{a, u}	{<f(u), f(a)>}
*{f(u), w}	{a_5}

(c) The attempted unification graph for Γ_6 of figure 4.16.1. The unsuccessful merge is indicated by the dotted line.



(d) S for U(Γ_6) is as for U(Γ_5) with the addition of *{z, x} {a_6}

*{w, y} {a_6}

The reduced covering expression for the failed merge arc is

$\Omega(\langle b,y\rangle \cdot \langle y,w\rangle \cdot \langle w,f(u)\rangle \cdot \langle f(u),f(a)\rangle) = a_1 + a_6 + a_5 + a_4$

Also label(-P(z, w)) = $a_3 + a_5$

So following the above construction:

$E_1 = (a_1 + a_5 + a_4) \cdot (a_3 + a_5)'$

$\therefore E_2 = a_1 \cdot a_3' \cdot a_5' + a_4 \cdot a_3' \cdot a_5'$

$\therefore E = a_1 + a_4$

Hence, to construct a_6 we must first delete either a_1 or a_4.

With the modifications described in this section, then, the Baxter algorithm satisfies criterion (B) at least for cases of unifiability detected during the transformational stage. As yet it is not clear how ununifiability detected in the sorting stage is to be handled. Research is proceeding on that problem.

4.2.3 Criterion (C)

When an arc is removed from Γ and all the corresponding arcs removed from $U(\Gamma)$, several unfortunate things may happen to the unification graph. For example, nodes which should be in the same tree may not be. In example 4.16, for instance, if a_4 is removed from Γ, nodes b and y of $U(\Gamma)$ will be in different trees, although the constraint {y, b} is still in S. Trees may also become unbalanced, impairing the efficiency of further applications of the unification algorithm. It is hoped that further research will produce a method for repairing $U(\Gamma)$ with the minimum of reprocessing.

5. $\mathcal{S}$-GRAPHS IN MECHANICAL THEOREM PROVING

In this section, we will attempt to show that $\mathcal{S}$-graph deduction has definite advantages over the linear deduction systems on which it is based.

5.1 The problem reduction approach

It has long been realised by those researching automatic deduction, plan formation, question answering systems, etc., that an extremely powerful technique in problem solving, is the method of problem reduction, in which a problem is replaced by a set of (hopefully simpler) sub-problems, which must be simultaneously solved. Many of the new languages of artificial intelligence are based on the problem-reduction method; for example [6, 8, 10, 11, 18].

A major difficulty with using problem reduction in predicate calculus, is that the subproblems are rarely independent: finding a certain solution to a particular subproblem may destroy our chances of solving one of the other subproblems. Consequently, to take full advantage of the problem-reduction method, we must process the subproblems in as parallel a way as possible, so that if our work on subproblem A blocks the solution of subproblem B, then this fact is discovered as soon as possible, before great effort is expended on a solution for A that will eventually have to be erased.

Simple linear deduction, that is, linear deduction with factoring and ancestor resolution [16], allows subgoals to be processed in any order; however, it lacks the power of $\mathcal{S}$-graph deduction, in that its use of lemmas is restricted (see section 5.2) and it has no reduction rule. If reduction is used in an ordinary linear format, then to ensure completeness an ordering must be imposed on the subgoals, and the system then suffers from the shortcomings mentioned above.

Every leaf in an $\mathcal{S}$-graph is a currently unsolved subproblem, and the leaves may be solved in any order so $\mathcal{S}$-graph deduction has the "parallel processing" advantages of simple linear deduction, but is still more powerful than the more sophisticated linear deduction schemes.

5.2 Use of lemmas

Most linear deduction systems allow the use of lemmas: that is, any clause which has been deduced in the course of the current proof, may be used as an input clause. In fact, simple linear deduction requires this for completeness. In every linear deduction system which allows the use of lemmas, however, the very linearity precludes the use of many lemmas which we have available for use in $\mathcal{S}$-graph deduction.

Each $\mathcal{S}$-graph actually corresponds to a set of linear deductions; in fact, each possible sequence of rules for building Γ, corresponds to one linear deduction of $L(\Gamma)\sigma(\Gamma)$. Consequently, if Γ_1 is any sub-$\mathcal{S}$-graph of Γ, then $L(\Gamma_1)\sigma(\Gamma_1)$ is available for ancestor replacement in Γ, regardless of whether or not Γ_1 was actually generated during the construction of Γ.

Example 5.1

Let $\mathcal{S}$ be the set of clauses:

$$-P(x),\ Q(x)$$

$$-Q(x),\ P(f(x))$$

$$P(x),\ -Q(h(x))$$

$$-P(f(f(a)))$$

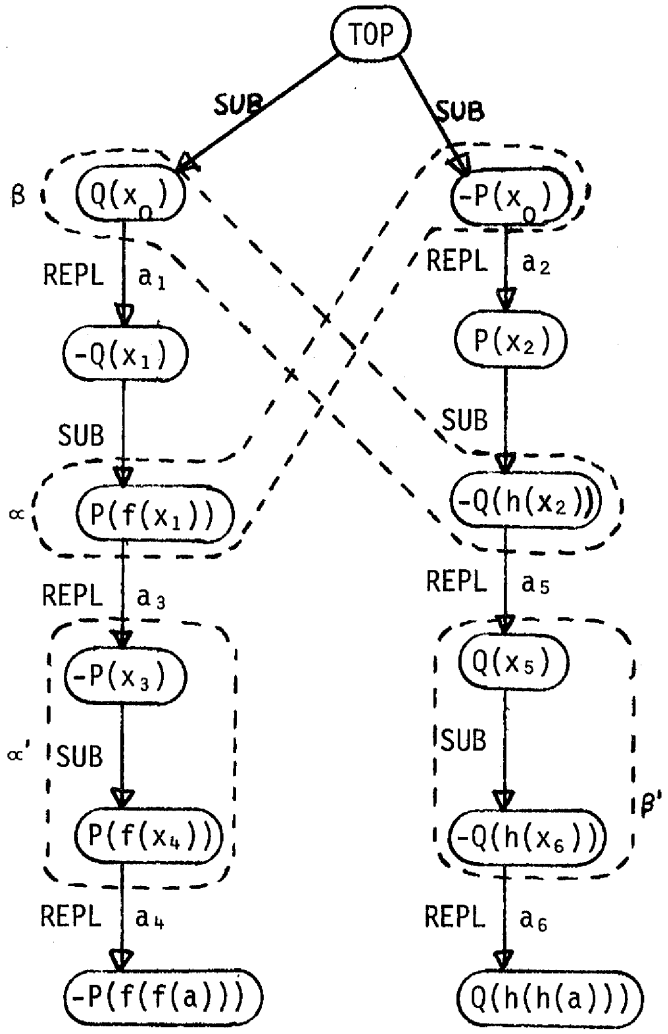
$$Q(h(h(a)))$$

Then figure 5.1.1 illustrates a closed $\mathcal{S}$-graph, and the associated constraint set.

Note that in no other system of linear deduction, would the clauses α and β both be available for use as lemmas, because in such a

system, once -P(x) in the top clause is solved, it is no longer available for use in a lemma; similarly for Q(x). However, one of these subproblems has to be solved first, so that only one of the two lemmas used in the $\mathcal{S}$-graph deduction of figure 5.1.1 is available.

Figure 5.1.1



$\{Q(x_0), Q(x_1)\}$	$\{a_1\}$
$\{P(x_0), P(x_2)\}$	$\{a_2\}$
$\{Q(x_3), Q(x_4)\}$	$\{a_3\}$
$\{P(f(x_1)), P(x_3)\}$	$\{a_3\}$
$\{P(f(x_4)), P(f(f(a)))\}$	$\{a_4\}$
$\{P(x_5), P(x_6)\}$	$\{a_5\}$
$\{Q(h(x_2)), Q(x_5)\}$	$\{a_5\}$
$\{Q(h(x_6)), Q(h(h(a)))\}$	$\{a_6\}$

A closed $\mathcal{S}$-graph and associated constraint for $\mathcal{S}$ of example 5.1. Note that the constraint set is unifiable. Some explanation of the ancestor resolutions is in order. Consider the ancestor clause marked β in the graph. We rename x_0 and x_1 in β as x_3 and x_4 respectively, obtaining β' subject to the constraint set $\{\{Q(x_3), Q(x_4)\}\}$. β' is used in replacement a_5, and the resulting constraint, together with the constraint on β' is added to the constraint set.

5.3 Economy

$\mathcal{S}$-graphs attain an economy of representation which is maximal, consistent with the unrestricted use of lemmas described in section 5.2. This is because one does not know *a priori* whether or not a given literal may be of use in a lemma, so that every literal must be represented at least once.

The sharing of structure in theorem-proving programs has been attempted before. Boyer and Moore in [4], suggested a method for representing resolvents of clauses by a system of pointers to parent clauses, and to the literals resolved upon. In their system, as in ours, each literal is represented only once; however, theirs is strictly a method of representation, and solves none of the problems associated with efficient backtracking, use of lemmas, ordering of subgoals etc. Although clauses are not explicitly created, they exist implicitly, so that in order to perform a resolution, one must make a recursive search through the structure to carry out the necessary unification and (implicit) construction of the resolvent.

Kowalski in [12] presents a deduction system called "connection graphs", which solves many of the inherent problems of linear deduction systems. In connection graphs, however, literals are repeated, substitutions are performed explicitly, and factors are introduced regardless of whether or not they may be required. Also, depending on the order of processing the arcs of the connection graph, clauses which could be used as lemmas in obtaining a short proof are sometimes deleted.

In $\mathcal{S}$-graph deduction, substitutions are never performed: whenever an $\mathcal{S}$-graph is closed, we have a refutation, and need know only that all the constraints are simultaneously unifiable. In this regard, our system

is similar to the higher-order constrained resolution system of Huet [9].

5.4 Backtracking

A problem that has always plagued mechanical theorem-proving systems is that of deciding what to do when the particular line of reasoning currently being pursued, leads to a dead end. The solution usually adopted is most unsatisfactory: namely, when a dead end is encountered, back up to the last point in the deduction where there was a choice of solutions, and try the next solution, assumming that at each choice point, the possible solutions are ordered in some way. This can lead to exhaustive and unnecessary searches in irrelevant areas of the search space.

Example 5.2

Let $\mathcal{S}$ be the set of clauses:

(1) P(x), R(b), R(x)

(2) -P(x), Q(x)

(3) -P(x), H(x)

(4) -Q(x), K(x)

(5) -Q(x), N(x)

(6) -H(x), K(x)

(7) -H(x), N(x)

(8) -K(x), M(x)

(9) -K(x), S(x)

(10) -N(x), M(x)

(11) -N(x), S(x)

(12) -M(x), -B(x)

(13) -S(x), -B(x)

(14) -R(x)

(15) B(a)

Suppose a proof of unsatifiability of this set is attempted using ME-deduction with factoring. To determine the order of choice at branch points of the search space, suppose that:

(i) Rules are ordered thus: contraction, reduction, factoring, extension.

(ii) Input clauses for extension are taken from $\mathcal{S}$ in the above order.

The following deduction is generated, in which A-literals are framed:

(1) P(x), R(b), R(x)

(16) P(b), R(b) Factoring

(17) P(b), [R(b)] Extension with (14)

(18) P(b) Contraction

(19) [P(b)], Q(b) Extension with (2)

(20) [P(b)], [Q(b)], K(b) Extension with (4)

(21) [P(b)], [Q(b)], [K(b)], M(b) Extension with (8)

(22) [P(b)], [Q(b)], [K(b)], [M(b)], -B(b) Extension with (12)

Backtrack to (20)

(23) [P(b)], [Q(b)], [K(b)], S(b) Extension with (9)

. three backtracings occur here

(39) [P(b)], [H(b)], [N(b)], [S(b)], -B(b) Extension with (13)

Backtrack to (1)

(40) P(x), R(b), [R(x)] Extension with (14)

(41) P(x), R(b) Contraction

(42) P(x), [R(b)] Extension with (14)

(43) P(x) Contraction

(44) [P(x)], Q(x) Extension with (2)

(45) [P(x)] , [Q(x)] , K(x) — Extension with (4)

(46) [P(x)] , [Q(x)] , [K(x)] , M(x) — Extension with (8)

(47) [P(x)] , [Q(x)] , [K(x)] , [M(x)] -B(x) — Extension with (12)

(48) [P(a)] , [Q(a)] , [K(a)] , [M(a)] , [-B(a)] — Extension with (15)

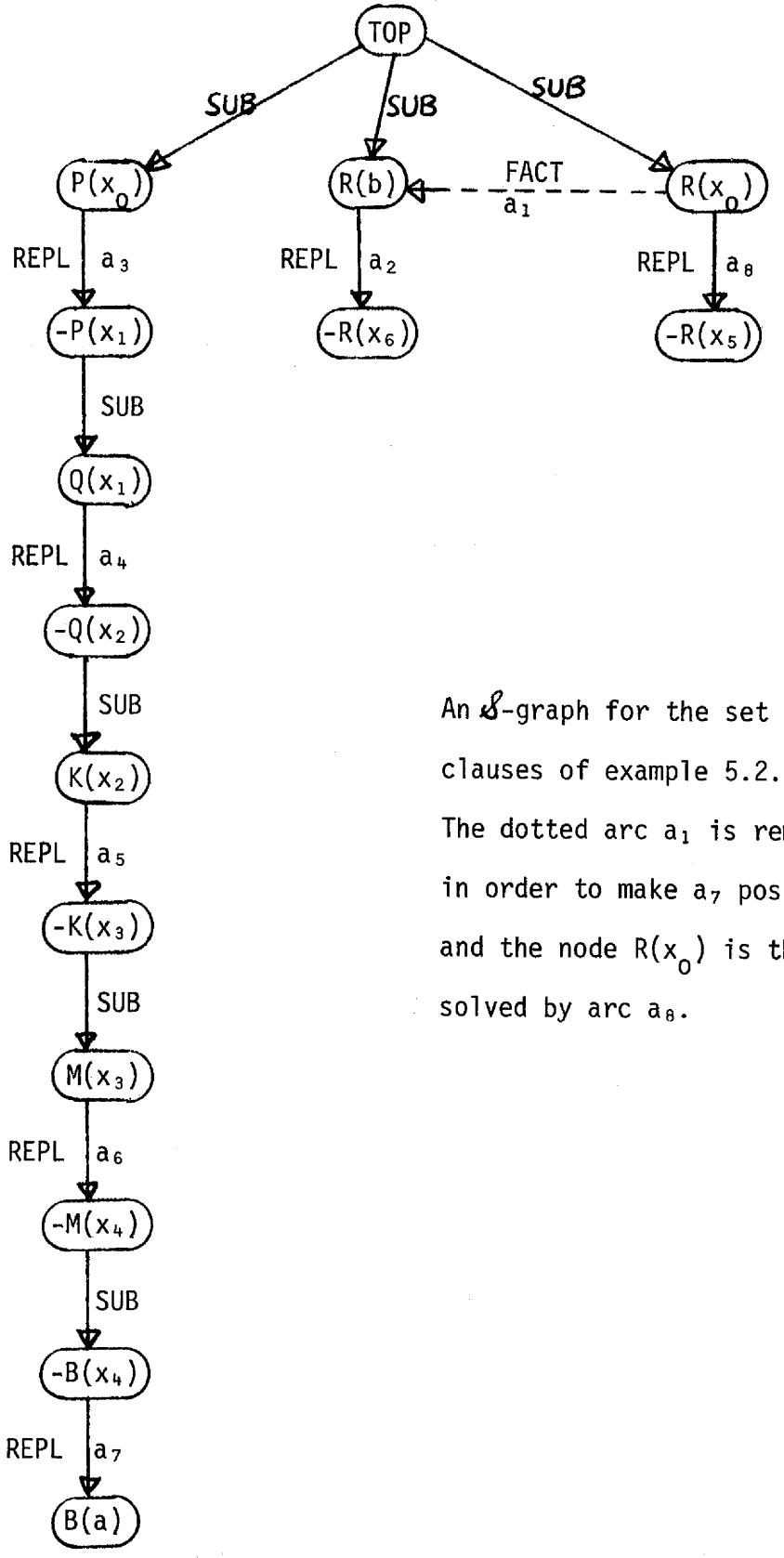
(49)
•
•
•
(53) □ — Contractions

Using the same ordering for alternative solutions, however, our system produces the $\mathcal{S}$-graph of figure 5.2.1, assuming that the same order is chosen for attacking subgoals as in the above ME-deduction.

When the construction of a_7 is attempted, the constraint processing system detects un-unifiability, and indicates that a_1 must be removed to allow construction of a_7 . This having been done, there is only one solution for the remaining subproblem.

$\mathcal{S}$ -graph dedution has yet another advantage over ordinary theorem provers, also related to backtracking. Namely, when it becomes impossible to solve a particular literal, and the source of the un-unifiability has been located, our system discards only the offending piece of proof. Other theorem provers on the other hand, on backtracking to the source of the problem (once it has been located, which, as shown above, involves large amounts of blind search in existing systems), will discard the entire proof from that point on. This is a very wasteful procedure, since some harmless and possibly correct subproofs will be removed, and later must be reconstructed. This is illustrated in the above example 5.2 whenever the ME procedure backtracks: for instance, when it backtracks from clause (39) to clause (1), the entire proof after clause (1) is lost, although it constitutes a perfectly valid solution to the subproblem P(x). Note,

Figure 5.2.1



An $\mathcal{S}$-graph for the set of clauses of example 5.2. The dotted arc a_1 is removed in order to make a_7 possible, and the node $R(x_0)$ is then solved by arc a_8.

however, that the corresponding backtrack in the $\mathcal{S}$-graph deduction preserves the proof of P(x), removing only the offending arc a_7.

REFERENCES

[1] Aho A.V., Hopcroft J.E. and Ullman J.D. The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[2] Baxter L.D. An Efficient Unification Algorithm. Research Report CS-73-23, Department of Computer Science, University of Waterloo (1973).

[3] Baxter L.D. A Practically Linear Unification Algorithm. Research Report CS-76-13, Department of Computer Science, University of Waterloo (1976).

[4] Boyer R.S. and Moore J.S. The Sharing of Structure in Theorem-proving Programs, in Machine Intelligence 7, 101-116, John Wiley and Sons (1972).

[5] Brzozowski J.A. and Yoeli M. Digital Networks, Prentice-Hall (1976).

[6] Colmerauer A., Kanoui H., Paséro R., and Roussel P. Un système de communication homme-machine en francais. Groupe d'Intelligence Artificielle, U.E.R. de Luminy, Marseille (1972).

[7] van Emden M.H. Programming with resolution logic. Research Report CS-75-30, Department of Computer Science, University of Waterloo (1975).

[8] Hewitt C. Planner: a language for proving theorems in robots. Proc. IJCAI, 295-302 (1969).

[9] Huet G.P. Constrained resolution: a complete method for higher order logic. Report 1117, Jennings Computing Centre, Case Western Reserve University (1972).

[10] Kowalski R.A. Predicate logic as a programming language. Proc. IFIP, 569-574 (1974).

[11] Kowalski R.A. Logic for problem-solving DCL Memo 75, Department of Artificial Intelligence, University of Edinburgh (1974).

[12] Kowalski R.A. A proof procedure using connection graphs, J.ACM 22, No. 4, 512-595 (1975).

[13] Loveland D.W. Mechanical theorem proving by model elimination, J.ACM 15, No. 2, 236-251 (1968).

[14] Loveland D.W. A simplified format for the model elimination theorem-proving procedure, J.ACM 16, No. 3, 349-363 (1969).

[15] Loveland D.W. A linear format for resolution, Proc. IRIA Symp. Auto. Demon. Springer-Verlag 147-162 (1970).

[16] Loveland D.W. A unifying view of some linear Herbrand procedures J.ACM 19, No. 2, 366-384 (1972).

[17] Nilsson, N.J. Problem Solving Methods in Artificial Intelligence, McGraw-Hill (1971).

[18] Nilsson N.J. and Fikes R.E. STRIPS: a new approach to the application of theorem proving to problem solving. Tech.note 43, Artificial Intelligence Group, Stanford Research Institute (1970).

[19] Luckham, D. Refinements in resolution theory. Proc. IRIA Symp. Auto. Demon. Springer-Verlag, 163-190 (1970).

[20] Paterson M.S. and Wegman M.N. Linear Unification. Proc. Symp. on Theory of Computing, SIGACT (1976).

[21] Robinson J.A. A machine oriented logic based on the resolution principle. J.ACM 12, No. 1, 23-41 (1965).

[22] Robinson J.A. Computational logic: the unification computation. Machine Intelligence 6, 63-72. American Elsevier (1971).

[23] Rulifson J.F. QA4 programming concepts. Tech. note 60, Artificial Intelligence Group, Stanford Research Institute (1971).

[24] Venturini Zilli M. Complexity of the unification algorithm for first-order expressions, to appear in Calcolo.