Mating Search Without Path Enumeration

Matthew Bishop

April 21, 1999

Submitted in partial fulfillment
of the requirements for
the degree of Doctor of Philosophy

Department of Mathematical Sciences,
Carnegie Mellon University,
Pittsburgh, PA 15213

Contents

Abstract vii

Acknowledgements ix

Chapter 1. Introduction 1
1.1. Background 1
1.2. Objectives 2
1.3. Brief Description 3
1.4. General Principles 4

Chapter 2. The Mating Search Method 7
2.1. Language 7
2.2. Expansion Proofs and Jforms 9
2.3. Matings 11
2.4. Natural Deduction Proofs 13
2.5. Implementations of Mating Search 14
2.6. Other Approaches to Path-Enumeration Search 16
2.7. Summary 17

Chapter 3. Partitions, Fragments and Components 19
3.1. Touching and Blocking 19
3.2. Motivation 22
3.3. Components 24
3.4. A Worked Example 25
3.5. Checking the Completeness of a Mating 29
3.6. Minimality 37

Chapter 4. The Component Search Method 41
4.1. The Search Algorithm 41
4.2. Enumerating Connections and Minimal Components 43
4.3. Soundness and Completeness 44
4.4. The Weighted Search Algorithm 45
4.5. Choosing a Fragment 47
4.6. Pruning out Redundant Matings 48
4.7. Related Work 48

Chapter 5. Unification 51
5.1. Unifiers as Directed Graphs 52

5.2. Operations on Unification Graphs 59
5.3. Extensions of Unification Graphs 66
5.4. Choosing a Variable Ordering 67
5.5. Duplication 68

Chapter 6. Duplication and Set-Variable Substitution 71
6.1. Duplication 71
6.2. Substitutions for Set Variables 77

Chapter 7. Equality Reasoning 79
7.1. Global and Local Equalities 79
7.2. Unification with Equalities 79
7.3. Search Algorithm with Equalities 81
7.4. Remarks on the Unification Method 82
7.5. Parity and Interpretations 84
7.6. Translation to a Natural Deduction Proof 87

Chapter 8. Optimizations and Variant Algorithms 91
8.1. A Parallel Version of the Search Algorithm 91
8.2. Alternative Definitions of a Fragment 93
8.3. Ordering Functions for Fragments 94
8.4. Weighting Functions for Components 95
8.5. Pruning and Reduction Operations 95
8.6. Skolem Nodes and Selection Nodes 97
8.7. Instantiating Definitions 98
8.8. Variable Orderings for Unification Graphs 98
8.9. Alterations to the Unification Algorithm 100
8.10. Extensions of the Equality Algorithm 100

Chapter 9. Results 101
9.1. Basic First-Order Problems 101
9.2. Basic Higher-Order Problems 104
9.3. Previously Intractable Problems 105
9.4. Comparison with Path-Based Searches 112

Chapter 10. Summary and Future Work 113
10.1. Summary 113
10.2. Future Work 113

Appendix A. Flags 115

Appendix B. Structures 127
B.1. Components 127
B.2. Unification 128
B.3. Equality Reasoning 128

Appendix C. Example 131

Bibliography 139

Index 147

Abstract

Mating search is a very general method for automating proof search; it specifies that one must find a complete mating, without specifying the way in which this is to be achieved. It is the foundation of TPS, an automated theorem-proving system for simply-typed lambda-calculus, and has proven very effective in discovering proofs of higher-order theorems. However, previous implementations of mating search have all relied on essentially the same mating search method: enumerating the paths through a matrix of literals. This is a depth-first strategy which is both computationally expensive and unable to avoid blind alleys in the search space; the unification problems which are generated in the higher-order case are difficult, and most of the computational effort in finding a complete mating is actually spent in unification rather than in enumerating different partial matings.

We describe a new breadth-first mating search method, called component search, in which matings are constructed by taking unions from a fixed list of smaller matings, whose unifiers are stored and manipulated as directed graphs. Storing unifiers as graphs also allows us to introduce equality reasoning into higher-order mating search.

Component search is capable of handling much larger search spaces than were previously possible with path-enumeration searches, and has produced fully automatic proofs of a number of interesting theorems which were previously intractable.

Acknowledgements

I am very indebted to Peter Andrews, my thesis advisor, both for his guidance and support during my many years at Carnegie Mellon and for teaching me a great deal about logic and automated reasoning. His guidance and encouragement have been immensely valuable, and his patience when confronted with half-baked solutions to ill-articulated problems appears to be boundless; I am also grateful for his financial support over the past years. I have been helped and encouraged by many discussions with Frank Pfenning, Rick Statman and Ed Clarke, all of whom have taught me a great deal and been extremely generous with their time.

Disorganized, hand-waving descriptions of this research have, over the years, been foisted on Chris Benzmüller, Chad Brown, Michael Kohlhase, Volker Sorge, Thilo Trapp and HongWei Xi; their comments and suggestions have been invaluable.

My parents and my brother have been a constant source of encouragement and support whenever I needed it. Finally, of course, I'm eternally grateful to Ping, who has supported, encouraged, and just plain put up with me throughout the whole process.

This work was supported by the National Science Foundation through grants CCR-9624683 and CCR-9732312, and by Intel Corporation through an equipment grant.

CHAPTER 1

Introduction

1.1. Background

The *mating search method* [Andrews, 1989; Andrews, 1981] came about as a result of reflections [Andrews, 1976] on the logical structure of proofs found by the resolution method [Robinson, 1965]. Unlike the resolution method, mating search does not require that a formula be reduced to clause-normal form; the more compact negation-normal form is used instead. The *connection method* was developed independently [Bibel, 1981; Bibel, 1987] and uses essentially the same ideas; early antecedents of this approach can be found in [Prawitz, 1970; Quine, 1955]. The mating search methods described in this thesis have been implemented in TPS [Nesmith *et al.*, 1996; Andrews *et al.*, 1996], which is an automated theorem-prover for classical type theory (i.e. higher-order logic [Andrews, 1986; Church, 1940]).

The mating search method has proven very effective as a component of a procedure for discovering proofs of higher-order theorems. We shall discuss mating search in more detail in Chapter 2; for now, the salient feature of this method is that it requires consideration of the vertical paths through a two-dimensional matrix of literals. Specifically, a proof is represented by a set of pairs of literals (a *mating*) with a substitution that simultaneously makes each of these pairs complementary, and such that every vertical path through the matrix contains at least one of these pairs (i.e. is *spanned by the mating*; a path which is not spanned is said to be *open*). Such a mating is said to be *complete*.

This characterization of a proof suggests an algorithm for proof search: given such a matrix, begin with the empty mating and an empty substitution. At each step, given a mating M, find the leftmost open path in the matrix, and attempt to span it by adding a new pair of literals (*connection*) to M and verifying that the result is unifiable. If there are no open paths, then M is complete and we have succeeded. If there is an open path but it cannot be spanned, then backtrack by removing the most recently added connection in M. If we attempt to backtrack when M is empty, then fail.

The above method, modulo some modifications to allow for duplication of quantifiers and substitutions for set variables, has been used in every search procedure in TPS. We shall refer to such a search as a *path-enumeration* search. However, mating search is a very general method for automating proof search; note that it specifies only that one must find a complete mating, without specifying the way in which this is to be achieved. It is therefore

possible to define alternative mating search procedures which do not rely on path enumeration. We first discuss the reasons why one might want to define an alternative search method, and then sketch the details of the new procedure.

1.2. Objectives

Determining the leftmost open path in a matrix is computationally quite expensive; it would be more efficient to have a search algorithm which could add connections to a mating without searching for open paths, thus separating the construction of the mating from the check for completeness. Furthermore, path-enumeration search becomes increasingly inefficient for large search spaces, since it is essentially a depth-first search of the space of possible matings. The order in which these matings are enumerated depends critically on the order in which vertical paths are enumerated, and hence upon the arrangement of literals in the matrix. In the worst case, if the very first connection which is added is wrong, we must explore all matings which contain this connection before finally backtracking and beginning again essentially from scratch. To compound the problem, one of the best strategies for duplication of quantifiers in a path-enumeration search says (more or less) "don't backtrack until you've tried duplicating all the quantifiers on the current open path"; this makes it even more difficult to backtrack out of blind alleys in the search.

This problem can be seen in two of the theorems to which TPS has been applied: THM15B (discussed in [Andrews *et al.*, 1996]), and THM126 (discussed in Section 3.4). TPS can prove THM15B quickly using a path-enumeration search, because the first four connections it considers happen to be correct, and form half of the complete mating. If the matrix is rearranged slightly (for example, by applying commutativity of $\wedge$ and $\vee$), the time taken for a proof increases considerably, because the first connections it finds are incorrect and it must search until it is forced to backtrack.

THM126 has a much larger matrix than THM15B; although it is a comparatively trivial theorem (it says that the composition of homomorphisms is a homomorphism), TPS cannot prove it at all using path-enumeration, because no available arrangement of the matrix will allow it to find more than a small fraction of the connections of a complete mating before it makes a significant error. There are enough potential errors available in the search space to ensure that, even after searching for several weeks, it will make no significant progress beyond the few connections it finds in the first few minutes. This point will be illustrated further in Section 3.4.

The tendency of path-enumeration searches to backtrack only as a last resort can be seen clearly in their automatic proofs of a well-known puzzle: suppose you have to cross a river using a boat which can carry only you and one other object. For reasons which are unclear, you have brought along a wolf, a goat and a cabbage; for reasons which are perfectly clear, the wolf can't be left alone with the goat, nor the goat with the cabbage. How do

you get all of this cargo across the river? This problem can be expressed in first-order logic by introducing a notation for the state (which objects are on which bank of the river) and writing down all of the acceptable transitions between states. One formalization of the problem is THM100 (see Section 9.1), which can be solved quickly by most mating search procedures. However, the automatic proofs produced by path-enumeration tend to produce many redundant steps — you take the wolf across the river and then immediately bring it back again, for example. This redundancy is caused by the way in which path-enumeration searches cannot backtrack unless they are absolutely forced to: once you try something, you're stuck with it until you reach the bounds of the search. This is one of the rare cases in which we are not forced to backtrack after a bad choice of connection, because alternative (albeit longer) proofs are available to us.

Another major source of inefficiency in mating search is the higher-order unification algorithm: because it must unify incrementally, after each connection is added, the search spends an enormous amount of time on unification which might better be spent searching for new matings.

Our objective, then, is to devise a breadth-first search strategy which is impervious to rearrangements of the matrix and which does not require backtracking. Such a strategy should store and reuse as much information as possible (in contrast to path-enumeration, which stores almost nothing), and should also avoid the more expensive aspects of path-enumeration search (in particular, incremental unification and the search for open paths) wherever possible.

We will also attempt, insofar as this is possible, to separate decisions affecting the search space itself (such as quantifier duplications and substitutions for set variables) from decisions affecting the search for a mating (such as which connection to consider next).

1.3. Brief Description

In most jforms, there are groups of literals which lie on all the same horizontal paths. For example, a theorem in which one hypothesis asserts the transitivity of some relation R will have a universally-quantified disjunction of three literals $\sim Rxy$, $\sim Ryz$ and Rxz. Using the disjunction heuristic, described in Lemma 3.1.6, we observe that any minimal complete mating which involves one of these literals (i.e. in which one of these literals is the endpoint of a connection) must also involve the other two. We may use this information to extend a partial mating involving one of these literals into a larger partial mating involving all of them. In this case, at least, it is not necessary to enumerate paths in order to decide which literal to mate next.

We describe a search procedure in which the jform is divided into groups of literals in this manner, and in which each partial mating is extended to a set of larger matings as described above. This produces a breadth-first search of the space of partial matings. By assigning weights to partial matings, extending the newly-created matings which have maximal weight and storing

all the others, we may reintroduce some element of depth-first search without requiring much computational effort for backtracking (backtracking will be a matter of choosing the stored matings with the largest weight).

The new search procedure thus avoids path-enumeration wherever possible and reduces the overhead of backtracking. In addition, since the groups of literals will have some variables in common, the number of ways of extending the mating will be constrained by the unifiability of the sets of connections which are added; this constraint will in general be stronger than the equivalent for path-enumeration procedures (which only add a single connection that can be almost anywhere in the jform).

We also describe a somewhat unusual approach to higher-order unification, based on the method of ordered binary decision diagrams [Bryant, 1986]. This method enumerates all the available substitutions (down to a particular depth), for every variable in the jform, before the search procedure begins. The unifiers for each connection are then stored as a directed acyclic graph, and unifiers for sets of connections are computed by merging sets of directed acyclic graphs in an obvious way. Although this introduces a considerable overhead at the beginning of the search, the subsequent computations appear to be much less expensive than the more usual unification algorithm for partial matings. Furthermore, this new algorithm has a natural extension to unification modulo (a bounded number of applications of) some set of equalities, which means that our search procedure can partly avoid the multiple duplications and large unification depths needed to deal with Leibniz-style equalities.

As the description above suggests, the new procedure is a little slower than path-enumeration for most small problems because of the extra computational effort required to start the search. However, it can solve larger problems which are completely intractable for path-enumeration search. In Chapter 9, we discuss a number of these previously intractable theorems (for example, a version of the Church-Rosser theorem, and a version of the Knaster-Tarski fixed-point theorem for lattices).

1.4. General Principles

In addition to the objectives mentioned above, there are several more general principles which should be kept in mind when designing any algorithm. Of course, none of the following is original with the present author, but all are applicable to the algorithms which are presented here.

Modularity: As far as possible, it should be feasible to alter the way in which one part of the algorithm works without having to alter everything else. In this context, one example would be the attempt to keep decisions about the search space entirely separate from decisions about the search; another would be the attempt to keep the overall algorithm as generic and nondeterministic as possible.

Eagerness: If there is something that is extremely likely to need to be computed sooner or later in the search, do it immediately, particularly

if doing so will allow you to constrain the search somehow. In this context, we assume that almost every connection will sooner or later be tried, and so we check them all for unifiability at the beginning of the search; this allows us to constrain the partial matings which will be considered.

Reusability: Store and reuse as much information as possible, particularly if it is likely to be needed again and it required some effort to obtain. In this context, the treatment of unification is almost entirely driven by an attempt to compute and store everything relevant before the search even begins.

CHAPTER 2

The Mating Search Method

2.1. Language

Well-formed formulae (wffs) are the formulae of Church's simple type theory [Church, 1940], a detailed presentation of which can be found in [Andrews, 1986]. Much of the following introductory discussion is taken from [Pfenning, 1987] and [Issar, 1991].

DEFINITION 2.1.1. *The set of types $\mathfrak{T}$ is defined inductively as follows:*

1. $\iota \in \mathfrak{T}$ *is the type of individuals.*
2. $o \in \mathfrak{T}$ *is the type of truth values.*
3. *If $\alpha \in \mathfrak{T}$ and $\beta \in \mathfrak{T}$, then $(\alpha\beta) \in \mathfrak{T}$. The type $(\alpha\beta)$ is the type of functions from elements of type β to elements of type α.*

DEFINITION 2.1.2. *The wffs of our language $\mathfrak{L}$ are defined inductively as follows:*

1. *A variable x_α is a wff of type α.*
2. *A constant k_α is a wff of type α.*
3. *$[A_{(\alpha\beta)}B_\beta]$ is a wff of type α, for any wffs $A_{(\alpha\beta)}$ and B_β.*
4. *$[\lambda x_\beta A_\alpha]$ is a wff of type $(\alpha\beta)$, for any variable x_β and wff A_α.*
5. *$\sim A_o$ is a wff of type o for any wff A_o.*
6. *$A_o \wedge B_o$ is a wff of type o for any wffs A_o and B_o.*
7. *$A_o \vee B_o$ is a wff of type o for any wffs A_o and B_o.*
8. *$\forall x_\alpha A_o$ is a wff of type o for any variable x_α and wff A_o.*
9. *$\exists x_\alpha A_o$ is a wff of type o for any variable x_α and wff A_o.*

DEFINITION 2.1.3. *We add the following defined symbols to our language:*

$$A_o \supset B_o \stackrel{\text{def}}{=} \sim A \vee B$$

$$A_o \equiv B_o \stackrel{\text{def}}{=} [A \supset B] \wedge [B \supset A]$$

We may also define equality using Leibniz' definition:

$$A_\alpha = B_\alpha \stackrel{\text{def}}{=} \forall q_{o\alpha}[qA] \supset [qB]$$

Wffs will be presented in the style of [Andrews, 1986], using the following conventions:

- A dot stands for a left bracket whose mate is as far to the right as is consistent with the pairing of brackets already present.
- Quantifiers and negations have the smallest possible scope.

- Omitted brackets associated with the connectives $\wedge$, $\vee$, $\supset$, and $\equiv$ should be restored in that order, giving the connective the smallest possible scope in each case.
- We may omit the type symbols when they are not relevant, or when they can be inferred from the context. (Our implementation of the language uses a type inference mechanism based on an algorithm by Milner [Milner, 1978].) All wffs in the following are assumed to be well-typed.

Substitution is defined in the usual way, taking care to avoid capture of free variables (we may do this by renaming bound variables appropriately). If Q is a binder ($\forall$, $\exists$ or λ), x a variable and A a wff, we use the informal notation $QxA(x)$ to indicate a wff in which x may occur free in A. We write either $A(t)$ or $[x \mapsto t]A$ for the result of substituting a term t for all of the free occurrences of x in A, renaming the bound variables in A to avoid capture of the free variables in t wherever necessary.

DEFINITION 2.1.4. *β-reduction of the redex $[\lambda x_\beta A_\alpha]B_\beta$ is defined by*

$$[\lambda x_\beta A_\alpha]B_\beta \xrightarrow{\beta} [x \mapsto B]A$$

which we may also write as $[\lambda x_\beta A_\alpha(x)]B_\beta \xrightarrow{\beta} A(B)$. If X is a term which contains a redex $[\lambda x_\beta A_\alpha]B_\beta$ as a subterm, and Y is the result of replacing this redex in X with the term $A(B)$, then we write $X \xrightarrow{\beta} Y$.

DEFINITION 2.1.5. *A is in β-normal form if there is no B such that $A \xrightarrow{\beta} B$. The relation $\overset{\beta}{=}$ is the equivalence closure of (i.e. the smallest equivalence relation containing) the relation $\xrightarrow{\beta}$.*

The following result is well known:

LEMMA 2.1.6. *The calculus above has the strong Church-Rosser property: every wff A in $\mathfrak{L}$ has a unique (up to renaming of bound variables) normal form B such that $A \overset{\beta}{=} B$.*

PROOF. See (for example) [Barendregt, 1984]. □

DEFINITION 2.1.7. *We define $\sim$-reduction as follows:*

1. $\sim\sim A \xrightarrow{\sim} A$
2. $\sim [A \vee B] \xrightarrow{\sim} [\sim A \wedge \sim B]$
3. $\sim [A \wedge B] \xrightarrow{\sim} [\sim A \vee \sim B]$
4. $\sim [\forall x A] \xrightarrow{\sim} [\exists x \ \sim A]$
5. $\sim [\exists x A] \xrightarrow{\sim} [\forall x \ \sim A]$

The relation $\xrightarrow{\sim}$ defines the usual rules for converting a wff into negation normal form. A is said to be in negation normal form if it is in β-normal form and there is no B such that $A \xrightarrow{\sim} B$. We define $\sim$-conversion as the equivalence closure of $\xrightarrow{\sim}$, and write $A \overset{\beta\sim}{=} B$ if A and B are equal up to β-conversion and $\sim$-conversion.

The following is well known:

LEMMA 2.1.8. *The calculus above has the strong Church-Rosser property: every wff A in $\mathfrak{L}$ has a unique (up to renaming of bound variables) negation normal form B such that $A \stackrel{\beta\sim}{=} B$.*

DEFINITION 2.1.9. *A_o is an* atom *if its negation normal form is not of the form $\sim B$, $B \wedge C$, $B \vee C$, $\forall xB$ or $\exists xB$. A wff A_o is a* literal *iff A is either an atom or the negation of an atom.*

DEFINITION 2.1.10. *Let W be a wff in negation normal form. We define the* positive and negative occurrences *of a subwff V of W inductively, as follows:*

- *If $W = V$ then V occurs positively in W.*
- *If $W = \sim W_1$ then V occurs positively (resp. negatively) in W iff V occurs negatively (resp. positively) in W_1.*
- *If $W = W_1 \vee W_2$ or $W = W_1 \wedge W_2$ then V occurs positively (resp. negatively) in W iff V occurs positively (resp. negatively) in either W_1 or W_2.*
- *If $W = \forall xW_1$ of $W = \exists xW_1$ then V occurs positively (resp. negatively) in W iff V occurs positively (resp. negatively) in W_1.*

DEFINITION 2.1.11. *Suppose V is a subwff of W. We say that V is* essentially universal *(resp.* essentially existential*) iff either $V = \forall xV_1$ and V occurs positively (resp. negatively) in W, or $V = \exists xV_1$ and V occurs negatively (resp. positively) in W.*

2.2. Expansion Proofs and Jforms

In [Andrews *et al.*, 1984] it was shown that a sentence is a theorem of elementary type theory if and only if it has a tautologous *development*, where a development is the analogue of a Herbrand expansion of a sentence of first-order logic. Expansion tree proofs (also known as ET-proofs or expansion proofs) represent the original theorem, the tautologous development, and the relation between them in an elegant and concise way. Miller's results about expansion proofs provided a basis for lifting the matings method from first-order to higher-order logic; expansion proofs are described briefly in [Andrews, 1989], and studied in depth in [Miller, 1983; Miller, 1984; Miller, 1987; Pfenning, 1984; Pfenning, 1987; Pfenning and Nesmith, 1990]. Since we do not require it for what follows, a full definition of expansion proofs will not be given here; instead, we provide a brief description of them.

We may represent a wff of first-order logic as a tree whose nodes are connectives, quantifiers or atoms, such that a node is a leaf iff it is an atom. Starting from this familiar idea, suppose we wish to refute some sentence Φ (and hence to prove $\sim\Phi$). Φ is said to be the *shallow formula* of such a tree; we also construct a *deep formula* for the tree which represents the result of instantiating quantifier occurrences in Φ with terms. These instantiation terms are attached as labels to the arcs descending from the corresponding quantifier node. We refer to the nodes corresponding to essentially universal

(respectively, essentially existential) nodes as *expansion nodes* (resp. *selection nodes*); their instantiations are called *expansion terms* (resp. *selection parameters*). The tree may contain other nodes, representing instantiation of definitions, lambda-conversion, or other transformations of the formula.

The above describes a *dual expansion tree*, which is called an *expansion proof* if it satisfies two additional conditions:

1. The deep formula is a contradiction.
2. Selections are done correctly.

The first condition is clearly related to Herbrand's Theorem [Herbrand, 1930], and we discuss it further below. The second condition can be made precise in a variety of ways; one notion of "correctness", taken from [Miller, 1983], is that the imbedding relation on the selection parameters is irreflexive. (The imbedding relation is defined below, in Definition 2.3.7.) One can obtain a similar effect by using Skolemization nodes instead of selection nodes; the details are discussed in [Miller, 1983]. It is interesting to note (although we are getting ahead of ourselves by doing so) that although previous search procedures have generally used Skolemization nodes, selection nodes are usually preferable for the new component search procedure. This will be discussed in Section 8.6.

Expansion proofs abstract away many of the details of concrete deductions (as opposed to sequent-style proof methods [Gentzen, 1935]) while preserving the structure of the formula (as opposed to resolution refutations [Robinson, 1965]), making them in some sense a universal structure for cut-free (or normal) proofs. They have since been adapted for use in automated reasoning about non-classical logics (see, for example, [Wallen, 1990]).

Junctive forms, or *jforms*, were first discussed in the literature in [Issar, 1990], although the search procedures of TPS had been using them for some time. They are derived from the *generalized and/or trees* of [Nilsson, 1980], and are even more abstract than expansion trees.

DEFINITION 2.2.1. *A* jform *is defined inductively as follows:*

- *A literal is a jform.*
- *If $J_1, \ldots, J_n$ are jforms, then $(\vee J_1 \ldots J_n)$ is a jform. We refer to this as a* disjunction node*; the empty disjunction node ($n = 0$) represents $\bot$ (falsehood).*
- *If $J_1, \ldots, J_n$ are jforms, then $(\wedge J_1 \ldots J_n)$ is a jform. We refer to this as a* conjunction node*; the empty conjunction node ($n = 0$) represents $\top$ (truth).*
- *If J is a jform and $x_1, \ldots, x_n$ are variables, then $\forall x_1, \ldots, x_n J$ is a jform. We refer to this as a* universal node *or* expansion node*.*
- *If J is a jform and $x_1, \ldots, x_n$ are variables, then $\exists x_1, \ldots, x_n J$ is a jform. We refer to this as an* existential node *or* selection node*.*

DEFINITION 2.2.2. *A jform is said to be* normal *if:*

1. *There is no node N such that N and the parent of N have the same type, and*

2. *There is no disjunction node* $(\vee J)$ *containing only one disjunct.*
3. *There is no conjunction node* $(\wedge J)$ *containing only one conjunct.*

Issar shows that every expansion tree can be converted into a jform in the obvious way, and that every jform can be converted into an equivalent normal jform.

We define the operations of duplication and deletion (the latter will not be needed for a while):

DEFINITION 2.2.3. *Suppose that* M *is an expansion node in a jform* J *and* ρ *is a renaming substitution for* J*. Let* J' *be the normal form of the jform obtained by replacing* M *in* J *with* $M \wedge \rho(M)$*. We say that* J' *is obtained by* duplication *from* J*. If* J^* *is the result of applying a sequence of such duplications to* J*, we say that* J^* *is an* amplification *of* J*.*

DEFINITION 2.2.4. *If* J *is a normal jform containing a node* N *with parent node* N' *(so* N *is not the root, and* N' *cannot be a literal), then the result of deleting* N *from* J *is the result of normalizing the jform defined as follows:*

1. *If* $N' = (\vee N, J_1, \ldots, J_n)$ *then replace* N' *by* $(\vee J_1, \ldots, J_n)$
2. *If* $N' = (\wedge N, J_1, \ldots, J_n)$ *then replace* N' *by* $(\wedge J_1, \ldots, J_n)$
3. *if* N' *is a universal or existential node then delete* N' *from* J*.*

2.3. Matings

The notion of a *mating* of course predates the expansion proofs and jforms discussed above; in fact, the basic idea of a mating can be found in [Davis, 1963], and the fundamental logical ideas underlying this approach to theorem proving go back to [Herbrand, 1930]. We have delayed the discussion of matings until now since the details of the methods of mating search will rely on these newer concepts.

DEFINITION 2.3.1. *Let* L *be the set of occurrences of literals in some jform* J*. A* mating $\mathfrak{M}$ *is a binary relation on* L *such that there is a substitution* $\sigma_{\mathfrak{M}}$ *with the property that* $\sigma_{\mathfrak{M}}A = {\sim}\sigma_{\mathfrak{M}}B$ *whenever* $A\mathfrak{M}B$*. Such a substitution is called a* unifier *of* $\mathfrak{M}$*.*

In what follows we abuse notation by speaking of "a literal" where we should more properly speak of "an occurrence of a literal". We will also use $\sigma_{\mathfrak{M}}$ as an ambiguous notation for "any unifier of $\mathfrak{M}$".

DEFINITION 2.3.2. *A* vertical path *is a sequence of literals. We define the vertical paths through a jform* J *recursively, as follows:*

- *If* J *is a literal* L*, then the sequence* (L) *is the only vertical path through* J*.*
- *If* J *is a disjunction* $J_1 \vee J_2$ *then every vertical path through* J_1*, and every vertical path through* J_2*, is a vertical path through* J*.*

- *If J is a conjunction $J_1 \wedge J_2$, then for every vertical path P_1 through J_1 and every vertical path P_2 through J_2, the concatenation of P_1 and P_2 is a vertical path through J.*
- *If J is a universal node $\forall \vec{x} J_1$ then every vertical path through J_1 is a vertical path through J.*
- *If J is an existential node $\exists \vec{x} J_1$ then every vertical path through J_1 is a vertical path through J.*

DEFINITION 2.3.3. *A* horizontal path *is a sequence of literals. We define the horizontal paths through a jform J recursively, as follows:*

- *If J is a literal L, then the sequence (L) is the only horizontal path through J.*
- *If J is a conjunction $J_1 \wedge J_2$ then every horizontal path through J_1, and every horizontal path through J_2, is a horizontal path through J.*
- *If J is a disjunction $J_1 \vee J_2$, then for every horizontal path P_1 through J_1 and every horizontal path P_2 through J_2, the concatenation of P_1 and P_2 is a horizontal path through J.*
- *If J is a universal node $\forall \vec{x} J_1$ then every horizontal path through J_1 is a horizontal path through J.*
- *If J is an existential node $\exists \vec{x} J_1$ then every horizontal path through J_1 is a horizontal path through J.*

DEFINITION 2.3.4. *Let $\mathfrak{M}$ be a mating for some jform J. We say that the unordered pair (A, B) is a* connection *iff A and B both lie on some vertical path of J, and either $A \mathfrak{M} B$ or $B \mathfrak{M} A$.*

DEFINITION 2.3.5. *A mating $\mathfrak{M}$* spans *a vertical path P iff P contains a connection of $\mathfrak{M}$.*

DEFINITION 2.3.6. *Given a jform J and a mating $\mathfrak{M}$, we say that $\mathfrak{M}$ is* path-acceptable *(or* p-acceptable*) iff every vertical path through J is spanned by $\mathfrak{M}$.*

Since J may contain selection nodes, we require the following definition from [Miller, 1987]:

DEFINITION 2.3.7. *Let J be a jform and ρ a substitution. Suppose that a and b are selection variables in J. Then we say $a \prec_{(\rho,J)} b$ iff:*

- *a occurs free in $\rho(x)$ for some expansion variable x occurring at an expansion node E in J, and*
- *b is a selection variable occurring at a selection node which is a descendant of E in J.*

The relation $\prec_{(\rho,J)}$ is called the imbedding relation on J induced by ρ. *It is said to be* acyclic *iff there is no sequence of selection variables $a_1, \ldots, a_n$ such that $a_1 \prec_{(\rho,J)} a_2 \prec_{(\rho,J)} \ldots a_n \prec_{(\rho,J)} a_1$.*

Finally, we have the following definition and theorem:

DEFINITION 2.3.8. *A* refutation *for a jform J is an ordered pair $(J^*, \mathfrak{M})$ such that:*

1. *J^* is an amplification of J.*
2. *$\mathfrak{M}$ is a p-acceptable mating for J^*.*
3. *The imbedding relation on J induced by $\sigma_{\mathfrak{M}}$ is acyclic.*
4. *If x is a variable such that $\sigma_{\mathfrak{M}}(x) \neq x$, then x is an expansion variable in J^*.*

THEOREM 2.3.9. *A jform J is unsatisfiable iff there is a refutation for J.*

PROOF. See [Issar, 1991; Miller, 1987]. □

It is possible that a wff will contain no existential quantifiers (for example, if it has been Skolemized), in which case the acyclicity condition becomes vacuously true. There are advantages to using selection nodes instead of Skolemizing, which will be discussed in Section 8.6.

In the light of the theorem above, we may search for refutation proofs by searching for p-acceptable matings in amplifications of the corresponding jforms. Since matings will presumably be constructed incrementally by some search procedure which halts on discovering a p-acceptable mating, it is natural to think of such a mating as being *complete*. This terminology has been in use informally for some time, but seems not to have appeared in the literature; we state it, for the record, here:

NOTATION *A* complete mating *is a p-acceptable mating (see Definition 2.3.6). A* partial mating *is a mating which is not complete.*

Before considering practical methods for searching for a complete mating, we briefly discuss relation between formal proofs in the usual sense (i.e. natural deduction) and the expansion proofs presented above.

2.4. Natural Deduction Proofs

Expansion proofs preserve the logical structure of theorems; nevertheless, they can hardly be described as an intuitively obvious way of presenting proofs, and it is desirable to translate them into a more human-readable form.

A procedure for automatically translating matings into natural deduction proofs in a Gentzen-style calculus was first presented in [Andrews, 1980]; this work was extended to expansion proofs in [Miller, 1984]. An improved method, including methods for handling equality and extensionality, was presented in [Pfenning, 1987; Pfenning and Nesmith, 1990]. The inverse translation, from natural deductions to expansion proofs, was developed in [Pfenning, 1984; Pfenning, 1987]. A central problem for these methods is the fact that no cut-elimination algorithm for type theory has ever been proven correct; [Pfenning, 1987] presents a candidate for such an algorithm, but without a proof of termination. Finding such a proof is intrinsically difficult; in particular, if it were possible to formalize such a proof in analysis

(i.e. type theory with added axioms of extensionality, descriptions and infinity), then one would have a proof from within analysis of the consistency of analysis, contradicting Gödel's Second Incompleteness Theorem. The reader is referred to [Andrews, 1989; Pfenning, 1987] for a discussion of this and related results.

Translation of expansion proofs to natural deduction proofs is important for reasons other than readability; specifically, a natural deduction proof which is produced by translating an automatically-derived expansion proof may be regarded as a certificate for the correctness of that particular instance of our automatic search. This addresses an obvious concern about automatic proofs: how do we know that there are no bugs in our (possibly extremely complicated) search algorithm? If the algorithm returns no certificate, then we can only rely on the skill of the programmer, or on our own ability to check the code. However, if a certificate is returned, in the form of a natural deduction proof, then we can verify the proof using a proof-checking algorithm (of course, we still need to establish the correctness of that algorithm, but at the very least it should be much simpler than any theorem-proving algorithm). Such considerations are important for applications of theorem-proving to areas such as hardware and software verification, and also form the basis of the integration of external reasoning components into the mathematical assistant ΩMEGA [Benzmüller *et al.*, 1997; Benzmüller *et al.*, 1999].

It is important, therefore, that any proofs produced by our new algorithm should also be translatable into natural deduction style. We will return to this point in Chapter 7, when we discuss equality reasoning.

2.5. Implementations of Mating Search

Mating search is the process of constructing a refutation for a jform J. In order to do this, we must find an amplification of J and construct a complete mating $\mathfrak{M}$ and an appropriate unifying substitution $\sigma_{\mathfrak{M}}$ for it. We describe such a search method in the most abstract terms, and then indicate how the existing search procedures in TPS fit into this.

Mating search is done incrementally by beginning with the original jform J and an empty mating $\mathfrak{M}$, and then repeatedly finding an open path through the existing mating, adding a connection which spans this path, and verifying that the new mating is unifiable. The choice of when to duplicate quantifiers is left to the search procedure. Some form of backtracking which removes erroneous connections is also necessary. We omit discussion of substitution for set variables for the time being. The search algorithm should be complete, to ensure that the correct mating will not be overlooked.

The most naive search method in TPS simply enumerates vertical paths beginning with the leftmost, backtracks by removing the most recently-added connection, and duplicates all the outermost quantifiers if it backtracks over the empty mating. This is clearly not a very good method for any but the smallest search spaces.

Path-focused duplication, defined in [Issar, 1991], is more sophisticated than this; in particular, it abandons the level-saturation approach [Chang and Lee, 1973]. Duplication is by need; a quantifier is only duplicated if some path cannot be spanned otherwise, and even then only if it contains some literal which is potentially mateable with a literal on the current path. Quantifier duplication is done implicitly: the duplicated part of the jform is not constructed explicitly, and unifiers involving its literals are, where possible, produced by renaming the substitutions for the original literals rather than by explicit computation. Issar presents a method by which connections may be removed, when necessary, using intelligent backtracking [Cox, 1977; Cox and Pietrzykowski, 1981] with either the maximal unifiable subset method of [Cox, 1977], or the minimal non-unifiable subset method of [Bruynooghe and Pereira, 1984]. Unfortunately the method presented is nondeterministic, and the actual implementation defaults to blind backtracking; implementing an effective backtracking method is non-trivial (see [Issar, 1991] for details and more references to this and other methods for backtracking).

In all of the path-enumeration based mating searches in TPS, higher-order unification is performed after each new connection is added, using a bounded version of the semidecision algorithm of [Huet, 1975]. In contrast to first-order unification, which is decidable in linear time [Paterson and Wegman, 1978] and non-branching (returning a most general unifier if it succeeds), higher-order unification is undecidable [Huet, 1973; Goldfarb, 1981] and branching.

Unification is one of the major problems in designing an effective higher-order search; experiments with TPS have shown that higher-order searches with path-enumeration spend, on average, about 80-90% of their time on unification. The search for open paths is rather computationally expensive, and accounts for a significant proportion of the remaining time, leaving very little time for actually generating matings. This observation is one of the major motivations for the component search method: we attempt to constrain the branching of the unification search by choosing batches of connections which have some variables in common and then unifying them all at once. The results can be stored as unification graphs (Chapter 5) and we describe an algorithm for quickly combining pairs of unification graphs. We also introduce a search algorithm, based on these batches of connections, which avoids the search for open paths in most cases. This has greatly reduced the proportion of the search time spent on unification and path-enumeration, and so has allowed larger search spaces to be considered.

One could, of course, decide to avoid higher-order logic altogether, using the methods of [Boyer *et al.*, 1986] to express higher-order problems in first-order logic; arguments against doing this can be found in [Andrews and Bishop, 1996].

2.6. Other Approaches to Path-Enumeration Search

The search algorithm which we present in Chapter 4 dispenses with path-enumeration for the majority of the search. One might wonder whether there are more efficient ways to approach path-enumeration searches, rather than abandoning them altogether.

An alternative approach to paths in mating search is the *path dissolution* inference rule introduced in [Murray and Rosenthal, 1993]. This rule may be viewed either as a generalization of the matrix reduction rule of Prawitz [Prawitz, 1970], or as a generalization of the method of analytic tableaux [Fitting, 1990; Smullyan, 1968]. Path dissolution repeatedly removes unsatisfiable paths from a wff, and has the appealing property (at least in the propositional case) of being strongly complete: any sequence of path dissolutions applied to a wff must terminate with (in effect) a list of models for the wff; it is therefore an extremely powerful method for reasoning about propositional logic. (By contrast, the analogous strong completeness statement for connection graph resolution is known to be false [Eisinger, 1986].) The algorithm is of course not strongly complete in the first-order case; if it were, we would have a decision procedure for first-order satisfiability. In this case, the path dissolution rule produces an instantiated copy of the original graph (with the instantiation being the most general simultaneous unifier of the links which are dissolved) conjoined with the original graph. In the absence of most general unifiers, it is not clear how to generalize this to the higher-order case in a useful way.

Dissolution is one example of a *reduction operation*; it removes literals from the search space without affecting satisfiability. Reduction operations are particularly useful in removing redundancy from proof searches. An overview of the major sources of redundancy in connection methods is given in [Bibel *et al.*, 1995], and techniques for avoiding redundancy in connection methods are discussed in [Brüning, 1994]. Removing redundancy from the search space is *not* always beneficial, since it can in fact increase the length of the shortest available proof [Baas and Leitsch, 1994]. It can even be useful to *add* redundancy to some search spaces; dual instantiation [Bishop and Andrews, 1998] handles higher-order definitions by doing exactly that. Nonetheless, removing redundancies is a good idea in most cases.

Subsumption [Ramesh *et al.*, 1995; Bol *et al.*, 1991] is perhaps the best-known reduction operation; it is used by many first-order provers. Reduction operations have been widely studied, particularly in relation to tableau-based methods, and very efficient proof procedures have been developed using them (for example, the authors of the Markgraf Karl system [Blasius *et al.*, 1981] credited the strength of their system mostly to its reduction operations). Surveys of reduction stategies can be found in [Eisinger *et al.*, 1991].

Lifting reduction operations to the higher-order case (and, where necessary, to the non-clausal case) would seem to offer an alternative to eliminating path-enumeration altogether. However, these operations become more

expensive to apply in the higher-order case (the branching of the unification search causes problems for many of them), and this would not address the inefficiency of unification itself. However, the existence of these reduction operations in the non-higher-order case means that component search may be less effective than alternative search methods in the first-order case (although, as a breadth-first search, it should still be expected to outperform depth-first searches on certain examples).

2.7. Summary

We summarize some of the limitations of higher-order path-enumeration searches, at least as they are currently implemented in TPS:

- We unify incrementally after each connection is added; this is extremely computationally expensive and accounts for the vast majority of the time spent searching in most cases.
- No attempt is made to give priority to new connections which contain variables that already occur in the existing partial mating. In the fairly common case where a variable only occurs in one disjunction in the search space, at least some such connections will have to be considered, and doing so will significantly constrain the search for a unifier.
- The search is very sensitive to rearrangements of the jform.
- Path enumeration is expensive and inflexible.
- Duplication, even in the path-focused case, is also expensive, and causes combinatorial explosion to the point of intractability after only a few duplications.

Component search avoids (or at least mitigates) all of these problems by:

- Dividing the jform into fragments and combining matings for each fragment to produce a complete mating.
- Doing many duplications and set-variable substitutions at the start of the search, and then limiting the use of the extra copies in an appropriate way.
- Constructing all possible substitutions that might occur in unification, unifying each potential connection before the search even starts, and then using an algorithm similar to the algorithm for binary decision diagrams to combine these unifiers (hence, in particular, no substitution or lambda-normalization need occur after the first step of the search).
- Using the disjunction heuristic, rather than a path-enumerator, to choose the next connections wherever possible.
- Using weightings and search-pruning methods to allow greater control over the search.

CHAPTER 3

Partitions, Fragments and Components

3.1. Touching and Blocking

Suppose we are given a jform J, and let L be the set of literals (strictly, the set of occurrences of literals) in J.

Instead of searching for a complete mating by adding single connections to a partial mating, we would like to add larger sets of connections; this will allow us to constrain the unifying substitutions for a partial mating much more strongly, and also to reach a complete mating in fewer steps. Obviously, we would like to do this in a way that will result in a sound and complete search procedure; the first step towards this goal will be to partition L in some way, and use the elements of the partition to guide the construction and use of the above-mentioned sets of connections.

The following two definitions produce such a partition.

DEFINITION 3.1.1. *Let L be partitioned by some equivalence relation $\sim_J$ on literals. A* fragment *of J is an equivalence class of $\sim_J$.*

By definition, we have that every element of L occurs in exactly one fragment, and this will be sufficient for a large part of the discussion which follows. In practice, however, there are only a few choices for $\sim_J$ which make sense in the context of proof search. For clarity, we will choose a particular definition of $\sim_J$ and assume that this remains fixed throughout the following discussion. We postpone the consideration of alternative choices for $\sim_J$ until Section 8.2

DEFINITION 3.1.2. *Let J be a jform, and let A, B be literals in J. Then $A \sim_J B$ iff all of the following hold:*

1. *Every horizontal path through A also passes through B.*
2. *Every horizontal path through B also passes through A.*
3. *The universal quantifiers above A in J are the same as the universal quantifiers above B in J.*

It is clear that $\sim_J$ is an equivalence relation, and the equivalence classes of this relation will be the fragments of our search. Note that for the jforms we will consider, all of the quantifiers will be universal, since existential quantifiers will have been removed.

EXAMPLE 3.1.3. *We give some examples of the fragments defined by Definition 3.1.2:*

1. *For any jform which is in clause normal form, the fragments are exactly the clauses.*
2. *In the following jform there are four fragments:* $\{A, F\}, \{B\}, \{C\}$ *and* $\{D, E\}$.

$$A \vee \begin{bmatrix} \forall q \\ [B] \end{bmatrix} \vee C \\ D \vee E \end{bmatrix} \vee F$$

3. *A larger example is given in Figure 3.1.*

In order to define a search algorithm which is based on fragments, we will require the concepts of *touching* and *blocking*, which we now define.

DEFINITION 3.1.4. *Given two sets of literals* K *and* L*, we say that* K *is* touched by L *iff* $L \cap K \neq \emptyset$.

For the sake of brevity we will use this definition repeatedly, in multiple senses, by replacing K and L with some objects which define two sets of literals. In particular:

1. A fragment F is touched by L if $L \cap F \neq \emptyset$.
2. A mating $M = \{(l_{2i}, l_{2i+1}) \mid 0 \leq i \leq n\}$ is touched by L if $\{l_i \mid 0 \leq i \leq 2n+1\} \cap L \neq \emptyset$.
3. A vertical path $P = l_0, \ldots, l_n$ is touched by L if $\{l_i \mid 0 \leq i \leq n\} \cap L \neq \emptyset$.
4. A literal l is touched by the mating M if $\{l\}$ is touched by M.

DEFINITION 3.1.5. *Let* J *be a jform, and* L *a set of literals from* J*. A fragment* F *is said to be* blocked by L *if every vertical path in* J *which touches* F *also touches* $F \cap L$.

REMARK In particular, if the fragments are defined as in Definition 3.1.2, then a fragment F is blocked by L iff $F \subseteq L$. This follows since if there were some $l \in F \setminus L$, we could choose a vertical path P such that $P \cap F = \{l\}$, contradicting the definition of blocking.

As with the definition of *touched*, we will feel free to abuse notation and speak of a fragment being blocked by any object which defines a set of literals. To forestall any possible confusion, we remark that in order for a mating to span every vertical path in J which passes through F, it is neither necessary nor sufficient for it to block F.

Note that, by definition, a fragment F which is blocked by L is also touched by L (because there certainly exist vertical paths through J which touch F, and in order for them to touch $F \cap L$ this set must be non-empty, whence F touches L). The converse is not true; if F is touched by L then there may be some vertical path through J which passes through F without passing through any of the literals in $F \cap L$.

The definition of blocking is motivated by the following lemmas:

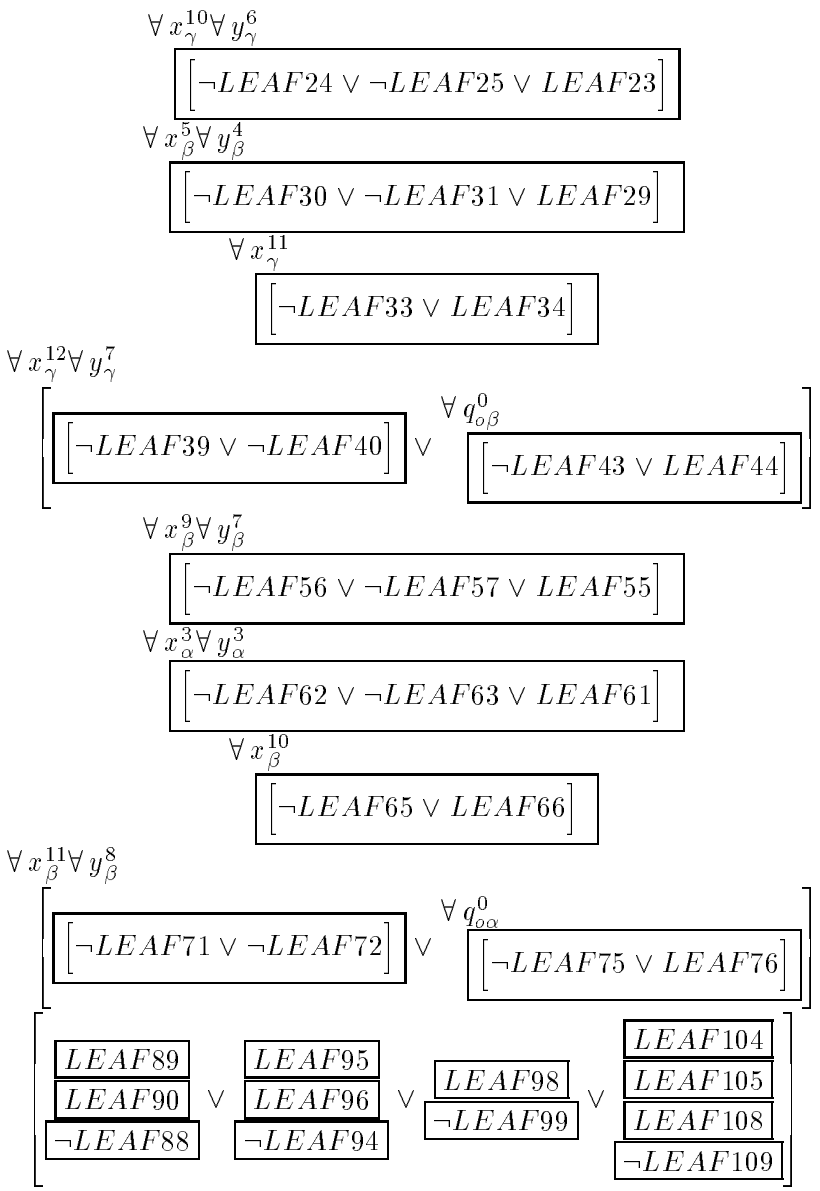


FIGURE 3.1. The fragments of a jform (each outlined box is a single fragment), as given by Definition 3.1.2.

LEMMA 3.1.6. *[The Disjunction Heuristic] If J contains a disjunction $A \vee B$, and M is a complete and minimal mating for J which touches some literal in A, then M must also touch some literal in B.*

PROOF. Suppose $c \in M$ is a connection which touches some literal in A. Since the mating is minimal, there must be some vertical path P, through J, on which c is the only spanning connection (if there were no such path, then $M \setminus \{c\}$ would be a complete mating, contradicting minimality). Let P' be

a variant of this path which is identical to P outside of A, but which passes through B instead of A. Since M is complete, P' must also be spanned by some connection $c' \in M$, and clearly $c' \neq c$ since c cannot lie on P' (because it touches A). If c' does not touch B, then c' lies on P, contradicting the assertion that c is the only connection spanning P. Hence c' must touch B. □

The name "disjunction heuristic" appears to have been first mentioned in the literature by Issar in [Issar, 1991], although the heuristic itself was well-known beforehand — see, for example, [Miller *et al.*, 1982; Hörnig and Bibel, 1982] for similar results.

DEFINITION 3.1.7. *Given a partial mating M, we say that:*

- *A fragment F is* weak in M *if it is touched but not blocked by M.*
- *M is* weak *if there is some fragment F which is weak in M.*
- *M is* strong *if it is not weak (i.e. every fragment which is touched by M is blocked by M).*

LEMMA 3.1.8. *In a jform J with fragments given by the definition of $\sim_J$ above, we have:*

1. *If M is a minimal complete mating, then M is strong.*
2. *Duplicating a quantifier in J will duplicate all of the fragments below that quantifier, and will leave all other fragments unchanged (in particular, it will not break up a fragment).*

PROOF. Suppose M is not strong; then some fragment F is weak in M. By the definition of $\sim_J$, this fragment contains a literal L_1 which is touched by M and a literal L_2 which is not, and both of these literals are children of the same disjunction node in J. Hence, by the disjunction heuristic, M cannot be complete, and so we have (1).

(2) follows immediately from part 3 of the definition of $\sim_J$. □

As we have already remarked, the definition of $\sim_J$ given in Definition 3.1.2 is not the only possible acceptable equivalence relation for defining the fragments of a jform; in fact, almost any relation satisfying Lemma 3.1.8(1) will suffice. Lemma 3.1.8(2) is a useful but not essential property of an acceptable equivalence relation, in the case where quantifiers might be duplicated after the fragments have been defined. We will return to the definition of the equivalence relation $\sim_J$ in Section 8.2.

3.2. Motivation

Lemma 3.1.8(1) is the major motivation for the definition of a search procedure based on fragments, as opposed to the existing procedures based on vertical paths. We now informally outline this new search procedure, before proceeding with a formal definition.

Suppose that we are given a jform J, and that there exists some complete mating for J. Suppose further that we have somehow enumerated the acceptable (i.e. unifiable) connections of J, and also that we have enumerated, for each fragment of J, all of the minimal sets of these connections which block that fragment. (By "minimal set", we mean "such that no proper subset of this set blocks the given fragment", rather than "of minimal cardinality".)

If we are then given a partial mating M, we may attempt to extend it to a larger partial mating by computing the list of fragments $f_1, \ldots, f_n$ which it touches but does not block. By Lemma 3.1.8(1), these fragments must be blocked in any minimal complete mating which extends M. Since we have enumerated the minimal sets of connections which block f_1 (suppose they are $C_1, \ldots, C_k$), we may simply generate the new matings $M \cup C_1, \ldots, M \cup C_k$, discarding any new matings which are not unifiable, or which violate some other bound which has been set by the user. Each of these new matings is clearly strictly larger than M, and hence if M is in fact a subset of a minimal complete mating, we are guaranteed that one of the new matings is a larger subset of the same complete mating. We may then repeat this process with each of the new matings, until we either reach a complete mating or fail.

Unfortunately, it is possible that the list of fragments which M touches but does not block is in fact empty. This is a necessary (Lemma 3.1.8(1) again) but not sufficient condition for completeness of M, so at this point we should check for completeness. If the mating is complete, we can stop; if not, we can always find an open path, and we can certainly enumerate all of the connections $c_1, \ldots, c_k$ which span it, and so we can generate $M \cup \{c_1\}, \ldots, M \cup \{c_k\}$ and continue as before. (In fact, we shall do a little better than this in some cases, but finding and spanning an open path will remain our last resort.)

If we can somehow get this process started, then we have all the makings of a sound and complete search procedure. To start, we may (for example) simply pick a fragment using any heuristic we like, and begin with all the minimal sets which block it. The set of support strategy might be a good heuristic for this purpose, since (as we will see) the starting fragment can be considered to be the goal of the search. If this fragment is touched by any complete mating, we will eventually find a complete mating; if not, then we will eventually fail everywhere, in which case we may choose a new fragment and start again.

Given the above, a new fragment-based search procedure appears to be possible; is it desirable? That is, is it likely to be more efficient than a path-enumeration procedure, at least on some class of problems? We argue that it is, for the following reasons:

- The literals in a fragment are much more likely to have variables in common than are the literals on a path. Thus, mating one of them is likely to constrain the unification substitutions for that variable, and hence the acceptable mates for the others; in an extreme case, there might not even be any acceptable mates for the others, in which case

path-enumeration would head off down a blind alley. By mating all of the literals in a fragment at once, rather than one at a time as they occur on vertical paths, we are more likely to avoid blind alleys, and also more likely to quickly narrow down the range of acceptable unifying substitutions.

- Furthermore, at least in the stages when the search blocks a fragment which is weak in the current mating, we can be reasonably sure that the unifiers which have already been determined will constrain the unifiers for the connections which are about to be added.
- Finding the next open path through a mating is computationally expensive; it requires a depth-first search of the vertical paths through the diagram. In the case where few open paths remain (or none), this is especially difficult (to convince oneself of this, take the complete mating for Figure 3.1 which is given in Section 3.4 and attempt to verify by hand that it really is complete). Finding a fragment which is weak in a mating is, by contrast, trivial. The algorithm outlined above only needs to find an open path on rare occasions, and so should be noticeably faster than path-enumeration.
- Path-enumeration searches are naturally depth-first searches through the space of possible matings, extending a mating by one connection at a time, and requiring backtracking. The fragment-based search outlined above is a naturally breadth-first search, extending each mating by several connections at a time, and requiring no backtracking (except in the case where the search fails everywhere and we must begin again from the empty mating). This again minimizes the effect of blind alleys in the search space.
- Since backtracking is not required, the fragment-based search can more easily preserve the information it generates about partial matings, unifiers, and so on, without the risk of storing a large amount of obsolete information. Furthermore, the absence of backtracking makes the search a natural candidate for OR-parallelism; from the sketch algorithm above, it is already clear (at least in principle; the practical details are another question entirely) how several processors might cooperate on a proof. We return to this idea in Section 8.1.

3.3. Components

Suppose that we have fixed a jform J and a partition $\sim_J$ which defines the fragments of J.

DEFINITION 3.3.1. *A* component *is a partial mating M which blocks at least one fragment. A component M is said to be* minimal for the fragment *F iff M blocks F and no proper subset of M blocks F. A* minimal *component is one which is minimal for some fragment.*

In our implementation, components are Lisp structures with slots for other information, such as the unifiers of the partial mating. All of this

other information is in principle recoverable from the partial mating M, so we may freely identify the component with its partial mating, as above.

The following two definitions are given mostly for the sake of brevity in the description of our algorithm:

REMARK Since a component is merely a particular sort of mating, we allow it to inherit most of the terminology of matings. In particular, we may speak of *weak* and *strong* components and *complete* components. The exception to this general rule is *minimal*; not every minimal mating is a minimal component.

DEFINITION 3.3.2. *If C is a component and $L = \{M_1, \ldots, M_n\}$ is a set of partial matings, we say that the set of components $L' = \{M_i \cup C \mid 1 \leq i \leq n \wedge M_i \cup C$ is unifiable$\}$ is the result of* combining C with L. *In particular, our algorithms will often say "combine C with L", where L is a set of components or of connections (thought of as single-element partial matings), by which we will mean "generate the set L'", as above.*

PROPOSITION 3.3.3. *If C is the union of the components A and B, then:*

1. *The fragments which are touched by C are exactly those which are touched by either A or B.*
2. *Any fragment which is blocked by A or B is also blocked by C.*

PROOF. Trivial. Note that the converse of (2) is false; a fragment F may be blocked in C but be weak (and so, by definition, not blocked) in both A and B. □

3.4. A Worked Example

In this section, we work through an example of the informal search procedure based on fragments which was outlined above, and compare it to the existing search procedures based on path enumeration.

The example which we use is THM126, which states that the composition of homomorphisms is a homomorphism, as follows:

$$\forall h1_{\beta\gamma} \forall h2_{\alpha\beta} \forall s1_{o\gamma} \forall f1_{\gamma\gamma\gamma} \forall s2_{o\beta} \forall f2_{\beta\beta\beta} \forall s3_{o\alpha} \forall f3_{\alpha\alpha\alpha}$$
$$.\mathrm{HOM2}\ h1\ s1\ f1\ s2\ f2 \wedge \mathrm{HOM2}\ h2\ s2\ f2\ s3\ f3$$
$$\supset \mathrm{HOM2}\ [h2 \circ h1]\ s1\ f1\ s3\ f3$$

where

$$\begin{aligned}
\mathrm{HOM2} &= \lambda k_{\alpha\beta} \lambda r_{o\beta} \lambda f_{\beta\beta\beta} \lambda s_{o\alpha} \lambda g_{\alpha\alpha\alpha}. f\text{–CLOSED2}r \wedge g\text{–CLOSED2}s \\
&\quad \wedge \mathrm{MAPS}_{o(o\alpha)(o\beta)(\alpha\beta)} krs \\
&\quad \wedge \forall x_\beta \forall y_\beta . rx \wedge ry \supset k[fxy] = g[kx].ky \\
\text{–CLOSED2} &= \lambda k_{\alpha\alpha\alpha} \lambda u_{o\alpha} \mathrm{MAPS2}kuuu \\
\mathrm{MAPS2} &= \lambda k_{\alpha\beta\gamma} \lambda u_{o\gamma} \lambda v_{o\beta} \lambda z_{o\alpha} \forall x_\gamma \forall y_\beta . ux \wedge vy \supset z.kxy \\
\mathrm{MAPS} &= \lambda k_{\alpha\beta} \lambda u_{o\beta} \lambda v_{o\alpha} \forall x_\beta . ux \supset v.kx
\end{aligned}$$

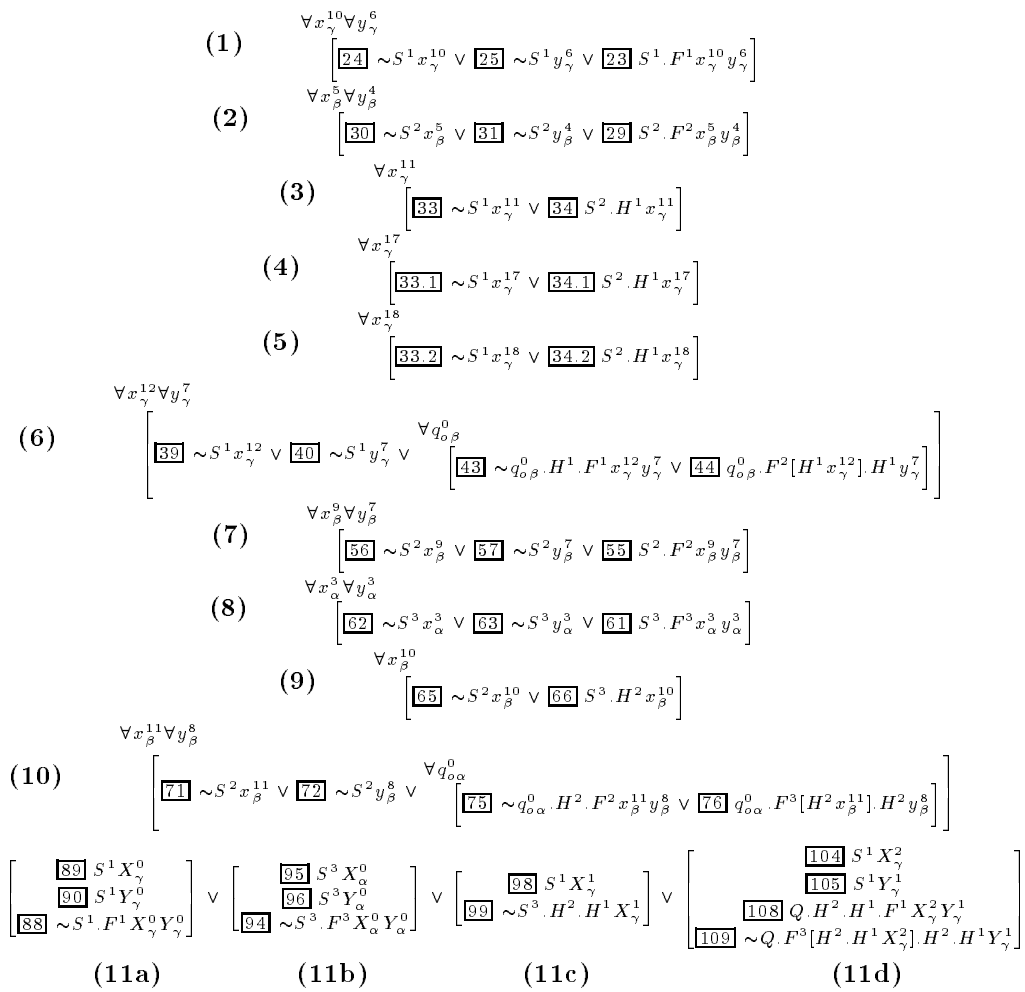


FIGURE 3.2. The jform of THM126, after duplicating quantifiers.

A version of this theorem, formulated in first-order set theory, was the central theorem of [Boyer *et al.*, 1986]; see Section 9.3 for more details.

The jform for this theorem is displayed in Figure 3.2. The two quantifier duplications which are necessary have already been applied to the jform, and so it is possible to find a complete mating without further expansion of the search space.

In plain language, the theorem says the following: suppose that $S^1_{o\gamma}, S^2_{o\beta}$ and $S^3_{o\alpha}$ are sets, that these sets are equipped with (and closed under) the binary functions F^1, F^2 and F^3 respectively, $H^1 : S^1 \longrightarrow S^2$ is a homomorphism with respect to F^1 and F^2 and $H^2 : S^2 \longrightarrow S^3$ is a homomorphism with respect to F^2 and F^3. Then $H^2 \circ H^1 : S^1 \longrightarrow S^3$ is a homomorphism with respect to F^1 and F^3.

We may express this as a diagram, as follows, in which the inner squares are assumed to commute, and we wish to show that the outer rectangle also

commutes. (If X and Y are sets, then $X \times Y$ is their Cartesian product; the function $F \times G$ is defined by $(F \times G)\langle x, y\rangle = \langle Fx, Gy\rangle$.)

$$\begin{array}{ccccc} S^1 \times S^1 & \xrightarrow{H^1 \times H^1} & S^2 \times S^2 & \xrightarrow{H^2 \times H^2} & S^3 \times S^3 \\ \downarrow{\scriptstyle F^1} & & \downarrow{\scriptstyle F^2} & & \downarrow{\scriptstyle F^3} \\ S^1 & \xrightarrow{H^1} & S^2 & \xrightarrow{H^2} & S^3 \end{array}$$

To see how this corresponds to the jform in Figure 3.2, we may translate the disjunctions of that jform into English as follows:

(1) The set $S^1_{o\gamma}$ is closed under the binary operation $F^1_{\gamma\gamma\gamma}$.
(2) The set $S^2_{o\beta}$ is closed under the binary operation $F^2_{\beta\beta\beta}$.
(3) $H^1_{\beta\gamma} : S^1 \longrightarrow S^2$.
(4) Another copy of (3).
(5) Another copy of (3).
(6) If $x \in S^1$ and $y \in S^1$, then $H^1(F^1xy) = F^2(H^1x)(H^1y)$.
(7) The set $S^2_{o\beta}$ is closed under the binary operation $F^2_{\beta\beta\beta}$.
(8) The set $S^3_{o\alpha}$ is closed under the binary operation $F^3_{\alpha\alpha\alpha}$
(9) $H^2_{\alpha\beta} : S^2 \longrightarrow S^3$.
(10) If $x \in S^2$ and $y \in S^2$, then $H^2(F^2xy) = F^3(H^2x)(H^2y)$.
(11) If:
  (a) The set $S^1_{o\gamma}$ is closed under the binary operation $F^1_{\gamma\gamma\gamma}$, and
  (b) The set $S^3_{o\alpha}$ is closed under the binary operation $F^3_{\alpha\alpha\alpha}$, and
  (c) $H^2_{\alpha\beta} \circ H^1_{\beta\gamma} : S^1 \longrightarrow S^3$, and
  (d) $x \in S^1$ and $y \in S^1$,
  then $H^2_{\alpha\beta}(H^1(F^1xy)) = F^3(H^2(H^1x))(H^2(H^1y))$.

There is some unnecessary detail here, which could be removed using the dual instantiation method of [Bishop and Andrews, 1998] — for example, (1) is a negated copy of (11a) — but even using dual instantiation, path-enumeration search cannot find a proof. Component-based search, on the other hand, finds a proof quite quickly. We describe the two methods, as applied to this search space, and attempt to explain why this is the case.

A complete mating for THM126 is the following: {$\boxed{104-39}$, $\boxed{105-40}$, $\boxed{108-43}$, $\boxed{75-44}$, $\boxed{71-34.1}$, $\boxed{72-34.2}$, $\boxed{109-76}$, $\boxed{105-33.2}$, $\boxed{104-33.1}$, $\boxed{95-62}$, $\boxed{96-63}$, $\boxed{94-61}$, $\boxed{65-34}$, $\boxed{99-66}$, $\boxed{98-33}$, $\boxed{89-24}$, $\boxed{90-25}$, $\boxed{88-23}$}. There are other complete matings for this jform (we may permute duplicate copies of the same subjform, for example), but all of the complete matings are essentially of this form.

Naive Path-Enumeration: The search finds the leftmost open path; at the first step, this is the leftmost path in the diagram. The first unifiable pair of literals that it will find on this path (enumerating the literals from the top down) is $\boxed{24-89}$. The next open path is the leftmost path from

[24] to [71], followed by the literals [95],[96],[94]. On this path, the first unifiable connection is [62 – 95]. On the next path (the leftmost path from [24] to [71], then [98] and [99]), the first unifiable connection is [33 – 98] (the connection [24 – 98] is unifiable in isolation, but not when taken together with the existing connection [24 – 89]), and on the path after that we discover [33.1 – 104].

All extensions of the path from [24] to [71] are now spanned, and so we begin to step back along this path, looking for the next open path. This turns out to be the variant of the leftmost path in the jform which runs from [24] to [88] but touches [63] instead of [62]. The connection [63 – 96] is the first to span this path, and the next path will be spanned by [61 – 94]. At this point, we have found six connections, all of which occur in the complete mating.

The next open path, however, is [24], [30], [33], [34.1], [33.2], [39], [56], [62], [65], [71], [104], [105], [108], [109]. The first unifiable connection which spans this is [30 – 34.1]; the other connections on this path are (in the order in which they will be enumerated) [56 – 34.1], [65 – 34.1], [71 – 34.1], [33.2 – 104], [39 – 104], [33.2 – 105], [39 – 105]. The first connection is not correct, but it will be added and the search will continue to enumerate extensions of this mating until it determines that all such extensions are incomplete, and backtracks to this point. Then it will add the next incorrect connection, and do the same again, and so on. These blind alleys in the search space are unimportant for small searches, or when they occur towards the end of a search (if we had found, say, 15 of the 18 connections in the complete mating before we first chose a wrong connection, we would probably backtrack again very soon), but for large searches, or at the beginning of a search, they can be a real problem. Of course, the unification constraints imposed by the rest of the mating are likely to be weakest at the beginning of the search, and so false connections are more likely to occur there. In a path-focused search, where quantifiers are duplicated by need, the problem of blind alleys becomes even more pronounced, since we must evaluate all the matings in *all expansions of* the search space before returning to backtrack. (Of course, path-focused duplication might also avoid some of the missteps of naive path-enumeration; we have used the naive approach here purely because it is easier to describe.)

One approach to this problem is to attempt to force the search to find the correct connections first; the TPS flag ORDER-COMPONENTS can be set to rearrange the jform into an equivalent jform in which (for example) literals with rigid heads have been moved towards the top left of the diagram. This can sometimes be useful, but is not particularly flexible and requires the user to set the flag correctly.

In the implementations of path-enumeration searches in TPS, no flag settings have been found which will allow TPS to find an automatic proof of this theorem. All of the attempts to find such a proof have found between four and six correct connections before entering a blind alley; even allowing

the search to continue for several weeks has not yet produced more than six correct connections.

Component Search: A complete run of the component search procedure on this example is included as Appendix C, from which the figures quoted in the following have been taken. Suppose we have enumerated all of the unifiable connections in the jform (there are 90 of them). We begin at the top left of the jform, again. By Lemma 3.1.6 we know that if $\boxed{24}$ is involved in a mating, so are $\boxed{25}$ and $\boxed{23}$, and so we enumerate all of the ways in which these three literals can simultaneously be mated (there are at most 126). One of these ways is $\boxed{89-24}$, $\boxed{90-25}$, $\boxed{88-23}$. If $\boxed{89}$ has a mate, then so must either $\boxed{98}$ or $\boxed{99}$ (or both), by the same lemma. One of the connections involving $\boxed{98}$ is $\boxed{98-33}$. If we have used $\boxed{33}$, then we must also use $\boxed{34}$; one way to do this is $\boxed{65-34}$; this requires us to use $\boxed{66}$, and so on. Note that at each stage we are constructing the entire tree of ways in which the current mating can be extended, which removes the problem of blind alleys (actually, the whole tree would branch too much if we did this, and so we both prune out some redundancies (see Section 6.1) and use a system of weightings (see Sections 4.4 and 8.4) to decide which parts of the tree should be extended next, but at least now we need not commit to exhausting every possibility in a blind alley before trying something else). Note also that in most cases, the next literal or literals to be used will have some variables in common with the literals already used, and so the unification constraints will help to prevent the tree from branching too much.

The implementation of component search in TPS can find a complete mating for this jform in a little over 4 seconds; the output from the search is presented in Appendix C. If we require TPS to find the quantifier duplications automatically, then the search requires a little more time; see Section 9.3.

3.5. Checking the Completeness of a Mating

The algorithm for component-based search will be presented in Chapter 4; first, we consider how to determine whether a mating is complete.

After each new connection in a path-enumeration search, we are required to find the next open path. This can be computationally very expensive, especially when we are close to completeness and there are few open paths. The algorithm outlined in Section 3.2 has the considerable advantage of being able to avoid searching for open paths under most circumstances, since we have other ways of determining what the algorithm should do next.

In general, our search algorithm will assume that C is a minimal mating[1], and will then use the following as a check for completeness of a component C:

[1]It is possible that C is in fact not minimal; assuming the contrary in the following will not affect the completeness of the algorithm (see Theorem 4.3.2). For more discussion of minimality, see Section 3.6.

1. If C is weak, the mating is not complete (by Lemma 3.1.8). The next step of the algorithm should be to block one fragment which is weak in C.
2. If C is strong, it may still be possible to use Lemma 3.1.6 to determine some list of fragments, at least one of which must be blocked in any complete mating extending C. (For example, if the jform contains $X \vee (Y \wedge Z)$, where X, Y and Z are fragments, and X is blocked, then either Y or Z must also be blocked.) If we can find such a list, we return it, and at the next step the algorithm should combine C with the set of all components which are minimal for any one of this list of fragments. In the remainder of this section, we will discuss one possible way to implement this step.
3. Otherwise, find an open path and return that; at the next step, the algorithm should combine C with the set of (unit sets of) connections which block this path. If there is no open path, then C is a complete mating; return `success`.

The first and third steps are essential; the former distinguishes component-based search from path-enumeration search, and the latter makes sure that component-based search is complete. The intermediate step is non-essential, and may safely be omitted; the search algorithm presented in the next chapter will make use of the results of this step, but it will also work properly if this step is omitted entirely (it will simply mean that one case in the algorithm never occurs).

In general, we may add any other checks that we can devise, and modify the search algorithm to handle the results. The objective of this completeness checker is to avoid computing open paths wherever possible; we have not been able to entirely abandon the search for an open path, for want of an efficient alternative characterization of mating completeness, but the above algorithm already greatly reduces the time spent enumerating paths. Any additional checks which are devised should be added after the first and before the last checks above.

The description of the algorithm for the second step is a little complicated and not particularly interesting, and (since omitting it altogether will not affect the soundness or completeness of the search) the reader may wish to skip ahead to the next chapter. However, the work required to formalize the second step will pay off, both by improving the efficiency of the overall search, and by constructing subjforms that will help in the computation of an open path in the third step (see Lemma 3.5.12).

To make the second step more precise, we introduce the notion of a collapsed jform. Recall that the jform which is used in a search space will be normal (no node will have a child of the same type), and will contain no existential nodes (they will have been removed by introducing either Skolem terms or selection parameters). Hence the remaining nodes will be of four types: conjunctive, disjunctive, universal and literal. Universal nodes have

a single child node, literals have none, and the junctive nodes have a list of child nodes (of length $\neq 1$).

Collapsing a jform will replace each fragment by a single literal. We are given a strong component, and we wish to produce a list of fragments, at least one of which should be blocked. This list will correspond to a partial vertical path through the collapsed jform; by hiding the internal details of the fragments, the construction can be made more efficient and more easily understood.

LEMMA 3.5.1. *Let J be a jform with fragments defined by $\sim_J$. For each fragment F_i, either F_i contains a single literal, or there is some junctive node N_{F_i} such that the literals below some subset C_{F_i} of the children of N_{F_i} are exactly the literals which occur in F_i.*

PROOF. Suppose F_i is a fragment, and that it contains more than one literal. All of the literals in F_i occur on the same horizontal path, and thus below some disjunctive node D. Choose the deepest such node D in the jform. Suppose that C is one of the children of D, and the literals of C are L_C. Then we show that either $L_C \subset F_i$, or $L_C \cap F_i = \emptyset$, which establishes the result, using the following case distinction:

1. If C is a literal, then the result is trivial.
2. C cannot be a disjunction, since it is a child of the disjunction D.
3. If C is a conjunction, then it contains more than one horizontal path. Hence either $F_i \cap L_C = \emptyset$, or $F_i \subset L_C$ (the latter since if $l_1 \in F_i \cap C$ and $l_2 \in F_i \setminus C$ then l_2 lies on some horizontal path which does not contain l_1, contradicting the definition of $\sim_J$). In the first case, we are done. In the second case, F_i must be contained within one of the conjuncts of C. Since F_i is not a single literal, there must be a disjunction D' somewhere below this conjunct such that all of F_i lies below D', contradicting our choice of D.
4. If C is a universal node, the definition of $\sim_J$ means that either $L_C \cap F_i = \emptyset$ or $F_i \subset L_C$ (by a similar argument to the previous case). Again, in the first case we are done. In the second case, the child of C cannot be a disjunction (which would contradict our choice of D), nor a literal (F_i contains more than one literal), and so it must be a conjunction. Then F_i must be contained within one of the conjuncts, contradicting our choice of D as in case (3).

□

DEFINITION 3.5.2 (Collapsed jform). *Let J be a jform with fragments defined by $\sim_J$. Let $\{F_o^i \mid i \in \mathbb{N}\}$ be constants. For each fragment F_i:*

- *If F_i contains only a single literal, replace that literal in the jform with the constant F_o^i.*
- *Otherwise, identify the node N_{F_i} and subset C_{F_i} from Lemma 3.5.1. Replace the subset C_{F_i} in the jform by the literal (constant) F_o^i.*

$$
\begin{array}{c}
F_o^1 \\
F_o^2 \\
F_o^3 \\
\left[F_o^4 \vee F_o^5\right] \\
F_o^6 \\
F_o^7 \\
F_o^8 \\
\left[F_o^9 \vee F_o^{10}\right] \\
\left[\begin{array}{lll}
F_o^{11} & F_o^{14} & F_o^{19} \\
F_o^{12} \vee F_o^{15} \vee F_o^{17} \vee F_o^{20} \\
F_o^{13} & F_o^{16} & F_o^{18} \quad F_o^{21} \\
 & & F_o^{22}
\end{array}\right]
\end{array}
$$

FIGURE 3.3. The collapse of the jform in Figure 3.1.

After this is done, for each universal node $Q = \forall x_1, \ldots, x_n Q_1$ in J, replace Q with Q_1. The collapse *of J is the normalization of the resulting jform, whose literals will be exactly the constants $\{F_o^i \mid F_i$ is a fragment of $J\}$.*

We will attempt to preserve this notation throughout what follows: F_i is a fragment in a jform J, and F^i is the corresponding constant literal in the collapse of J. The collapsed jform is essentially the result of replacing F_i by F^i throughout J, and erasing the quantifiers in J.

The collapse of a jform J preserves the conjunctive and disjunctive structure of J, up to reordering of the children of junctive nodes. For example, the collapse of the jform in Figure 3.1 is given in Figure 3.3. Hence we have the following lemma, which justifies the use of the collapsed jform:

LEMMA 3.5.3. *If $F^{i_1}, \ldots, F^{i_n}$ is a vertical (resp. horizontal) path through the collapse of J, and $P^1, \ldots, P^n$ are vertical (resp. horizontal) paths through the fragments $F_{i_1}, \ldots, F_{i_n}$ in J, then the concatenation of $P^1, \ldots, P^n$ is a vertical (resp. horizontal) path thorough J, up to reordering. Conversely, every vertical (resp. horizontal) path in J can be expressed as the concatenation of such a set of paths thorough fragments.*

PROOF. Obvious; the first assertion follows by the construction of the collapse of J, using Lemma 3.5.1, and the second by induction on the construction of vertical (resp. horizontal) paths in J. □

We now use the collapsed jform and the disjunction heuristic to attempt to construct a list of fragments, at least one of which must be touched by any complete minimal mating extending some given strong component.

DEFINITION 3.5.4. *The* hpath-complement *of a literal F_o^i in a collapsed jform J, written $\overline{F^i[J]}$, is the jform formed by deleting the literal F_o^i and all of the literals which do not occur on some horizontal path through F_o^i, and then normalizing the jform.*

Thus the only literals which survive in the hpath-complement of F^i are those literals which occur on a horizontal path with F^i, excluding F^i itself.

REMARK We can also define the hpath-complement $\overline{l[K]}$ of an arbitrary literal l in a non-collapsed jform K in exactly the same way, although this definition will not be needed in this section.

EXAMPLE 3.5.5. *In Figure 3.3, the hpath-complement of F^5 is (the single literal) F^4. The hpath-complement of F^7 is the empty jform, and the hpath-complement of F^{14} is:*

$$\begin{bmatrix} F^{11} & & F^{19} \\ F^{12} & \vee F^{17} & \vee F^{20} \\ F^{13} & F^{18} & F^{21} \\ & & F^{22} \end{bmatrix}$$

LEMMA 3.5.6. *Let J be a jform and let K be its collapse. Suppose M is a strong component, and F_i is the only fragment blocked by M in J. Then for every vertical path $P = F^{i_1}, \ldots, F^{i_n}$ through $\overline{F^i[K]}$, at least one of the fragments $F_{i_1}, \ldots, F_{i_n}$ must be blocked in J by any complete extension of M.*

PROOF. This is immediate from the definition of $\overline{F^i[K]}$. Note that the assumption that F_i is the only blocked fragment in J means that none of $F_{i_1}, \ldots, F_{i_n}$ is already blocked by M in J, and so by an analogous argument to the disjunction heuristic (Lemma 3.1.6), any vertical path through F_i has a variant through $F_{i_1}, \ldots, F_{i_n}$ which must be blocked. □

Unfortunately, this will not quite be sufficient for our purposes; firstly, the vertical paths which are produced may be longer than necessary (since we did not specify that J was the minimal such jform), and secondly we may miss vertical paths which a more careful application of Lemma 3.1.6 would have discovered.

For example, suppose our collapsed jform is $F^1 \vee F^2 \vee [F^3 \wedge F^4]$, and both F_1 and F^2 are blocked by M. We are clearly required to block either F_3 or F_4 in any complete minimal mating extending M. However, applying the above lemma would produce no paths at all, since any subjform J' containing F_1 either contains F_2 or contains exactly F_1 and nothing else. In the first case, the lemma is not applicable, and in the second case $F^1[K]$ (where K is the collapse of J') is empty.

To remedy this, we will need the following definition:

DEFINITION 3.5.7. *The* hpath-closure *of a literal F^i in a collapsed jform J is the smallest subjform $K \subseteq J$ which contains both F^i and $\overline{F^i[J]}$. We denote this $F^i[J]^*$.*

As we will shortly see (Lemma 3.5.9), the hpath-closure is a closure in the sense that it is the smallest jform containing F^i which is closed under taking horizontal paths (i.e.. which completely contains all of the horizontal paths which touch it).

REMARK Again, we can also define the hpath-closure $l[K]^*$ of an arbitrary literal l in a non-collapsed jform K in exactly the same way.

LEMMA 3.5.8. *For any F^i and J such that F^i occurs in J, the jform $F^i[J]^*$ is either a literal or a disjunction.*

PROOF. Clear: the only other type of node in a collapsed jform is a conjunction, and $F^i[J]^*$ cannot be a conjunction since F^i is a single literal and all the literals of $\overline{F^i[J]}$ occur on some horizontal path with F^i. □

LEMMA 3.5.9. *For any F^i, any horizontal path in a collapsed jform J which touches the hpath-closure $F^i[J]^*$ must remain completely within $F^i[J]^*$.*

PROOF. Suppose not; then there is a horizontal path containing two literals $l_1 \in F^i[J]^*$ and $l_2 \notin F^i[J]^*$. The smallest subjform of J containing both l_1 and l_2 is a disjunction D (because this is how two literals come to be on the same horizontal path).

Furthermore, $F^i \neq l_1$, since if $F^i = l_1$ then $l_2 \in F^i[J]^*$ because l_2 is on a horizontal path with l_1. Hence $F^i[J]^*$ contains at least two literals (F^i and l_1), and so is a disjunction D' by Lemma 3.5.8. Since l_2 is below D and not below D', but l_1 is below both, it must be the case that $D' \subset D$.

So D' is contained in some child node of D. Let l be any literal in any other child node of D, so that in particular $l \notin D'$; then l occurs on some horizontal path through $F^i \in D'$. Hence $l \in \overline{F^i[J]}$, by definition of $\overline{F^i[J]}$, but $\overline{F^i[J]} \subseteq F^i[J]^* = D'$ and so $l \in D'$, which is absurd. □

Suppose now that we are provided with a strong component M, and we wish to return the shortest possible list of fragments, one of which must be blocked in any complete mating which extends M. The collapsed jform, which will not change as long as J and $\sim_j$ remain fixed, is assumed to have been computed and stored.

ALGORITHM 3.5.10. *We now present an algorithm for computing a set S from a jform J and a list of fragments in J which are blocked by some component M. The set S will have the property that any element of S is a list of fragments at least one of which must be blocked by any complete extension of M. Let J' be the collapse of J.*

Let $K = \{F^i[J']^* \mid F_i \in J \wedge F_i$ is blocked by $M\}$. Every element of K contains at least one literal F^i such that the fragment F_i is blocked (it may contain more than one, since it is possible that for some $i \neq j$ we have $F^i[J']^* = F^j[J']^*$).

For each element of K which contains only one blocked fragment, we may apply Lemma 3.5.6 and add the results to S.

For elements $F^i[J']^*$ of K which contain more than one blocked fragment, we know by Lemma 3.5.8 that they must be disjunctions $J'_1 \vee \ldots \vee J'_n$. For each $1 \leq k \leq n$, we proceed by case analysis:

1. *If J'_k contains no literals F^m such that F_m is a blocked fragment, then by the same argument as Lemma 3.5.6 we have that any vertical path*

through J'_k represents a list of fragments, one of which must be blocked by any complete extension of M.

2. *If J'_k contains exactly one literal F^m such that F_m is a blocked fragment, then by Lemma 3.5.6 we have that any vertical path through $\overline{F^m[J'_k]}$ represents a list of fragments, one of which must be blocked by any complete extension of M.*
3. *If J'_k contains more than one literal F^m such that F_m is a blocked fragment, then apply this algorithm recursively to J'_k and the blocked fragments which it contains.*

REMARK Lemmas 3.5.6 and 3.5.3 guarantee that the set S has the desired property. Our completeness checker can then return one of the lists in S, if there is one, using any heuristic to choose this element. A reasonable such heuristic, since the next step in the search will be to attempt to block each of the fragments returned, would be to choose the list for which the sum of the number of minimal components for each fragment on the list is smallest; this will minimize the branching of the search. Given that only one element of S will be returned, there is probably no need to explicitly store or compute all of S; it is sufficient to save the minimal element at each stage.

EXAMPLE 3.5.11. *The above algorithm is somewhat opaque; we provide some examples in the hope of clarifying it. All of these examples will take J to be the following collapsed jform:*

$$\left[\begin{array}{c} \begin{bmatrix} F_o^1 \vee F_o^2 \\ F_o^3 \vee F_o^4 \end{bmatrix} \vee \begin{bmatrix} F_o^5 \vee F_o^6 \\ F_o^7 \end{bmatrix} \vee \begin{bmatrix} F_o^8 \\ F_o^9 \\ F_o^{10} \end{bmatrix} \vee F_o^{11} \\ F_o^{13} \vee F_o^{14} \end{array}\right]$$

We will consider this example with various different sets of blocked fragments:

F_{14} is the only blocked fragment: *First we compute $F^{14}[J]^*$, which is the disjunction $[F^{13} \vee F^{14}]$. Since F_{14} is blocked but F_{13} is not, we apply Lemma 3.5.6, which returns all the paths through F^{13}, namely $\{(F^{13})\}$, as expected.*

F_8 is the only blocked fragment: *We compute $F^8[J]^*$, which is the jform containing all of the fragments F^1–F^{11}. Since this contains only one blocked fragment, we apply Lemma 3.5.6, which, as expected, returns all the paths through* $\begin{bmatrix} F_o^1 \vee F_o^2 \\ F_o^3 \vee F_o^4 \end{bmatrix} \vee \begin{bmatrix} F_o^5 \vee F_o^6 \\ F_o^7 \end{bmatrix} \vee F_o^{11}$

F_5, F_7 and F_{11} are the only blocked fragments: *We compute $F^5[J]^*$, which is the jform containing all of the literals F^1–F^{11}. The hpath-closures of F^7 and F^{11} are also this same jform, so we must consider the disjuncts. The first disjunct (F^1–F^4) and the third disjunct (F^8–F^{10}) contain no F^m such that F_m is blocked, and so we return all the paths through them. The last disjunct contains only F^{11}, and*

F_{11} is blocked; applying case 2 we return no paths (since the hpath-complement of F^{11} in the jform containing only F^{11} is empty). The second disjunct is $K = [[F^5 \vee F^6] \wedge F^7]$, and both F_5 and F_7 are blocked, and so we apply the algorithm recursively. Applying the algorithm to $[F^5 \vee F^6]$ with F_5 blocked, will return $\{(F^6)\}$, as in the first example. Applying it to $[F^7]$ with F_7 blocked will return nothing. The reader may be surprised that we consider F_6 a candidate for blocking, since F_7 is already blocked, but recall that we are attempting to find a minimal *mating, and if F_5 is blocked then so must F_6 be.*

F_1 and F_3 are the only blocked fragments: *As in the previous example, we consider the disjuncts of the jform which contains the fragments F^1–F^{11}. The second, third and fourth disjuncts are untouched, and we will return all paths through them. Applying the algorithm recursively to the first disjunct with F_1 and F_3 blocked will return $\{(F^2)\}$ from the first case and $\{(F^4)\}$ from the second, as expected.*

F_5, F_7, F_{11} and F_{14} are the only blocked fragments: *In this case, the hpath-closures of F^5, F^7 and F^{11} in J are all the same, and will return the same lists of fragments as in the example above in which F_5, F_7 and F_{11} are blocked. The hpath-closure of F^{14} in J was also discussed above.*

REMARK Since the set of lists which is generated depends only on the list of blocked fragments (and the fixed jform), we may store these results, indexed by the list of blocked fragments, to avoid recomputing them.

The above method will return a list of fragments to be blocked if it can find one. It remains to deal with the case in which it cannot find such a list. In this case, we check for completeness of the mating M in J in the normal way, by searching for an open path, and either returning `success` or an open path. However, we may improve the efficiency of the search for an open path as follows:

LEMMA 3.5.12. *Given a jform J and a component M which blocks every fragment that it touches, construct two jforms J_1 and J_2 by deleting nodes from J, such that J_1 contains exactly those fragments F_i such that F_o^i occurs in the hpath-closures of the fragments blocked by M in the collapse of J, and J_2 contains everything else. If there is an open vertical path in J_1, the concatenation of this path and the leftmost vertical path in J_2 is an open vertical path in J. If not, the mating is complete.*

PROOF. By the definition of an hpath-closure, no literals in J_2 are touched, and no literals in J_2 occur on a horizontal path with any literal which is touched. It thus follows that the entire mating M is contained within J_1. Furthermore, since hpath-closures must contain only complete horizontal paths in J (Lemma 3.5.9), every pair of literals ($j_1 \in J_1, j_2 \in J_2$) lie on some vertical path. Hence if the sets of vertical paths in J_1 and J_2 are known, the set of concatenations of all pairs of elements from these sets is exactly the set of vertical paths of J.

Hence every vertical path (in particular, the leftmost) in J_2 is open, and if there is no open path in J_1, the mating must be complete. □

Searching for an open path in J_1 rather than J can significantly improve the performance of the check for an open path, especially when J_1 is small compared to J.

We end this section by summarizing our the results about the completeness checker:

THEOREM 3.5.13. *Given jform J and a strong component M, we perform the following steps to check for completeness:*

1. *Use Algorithm 3.5.10 to determine some list of fragments, at least one of which must be blocked in any complete mating extending M. If such a list is found, return it.*
2. *Otherwise, divide J into J_1 and J_2, as above. If there is an open path in J_1, concatenate it with the leftmost path in J_2 and return it. If not; return* `success`.

CLAIM *This is correct, in the sense that if it returns a list of fragments, then at least one of those fragments must be blocked in any complete mating extending M. If it returns a path, then that path is open and must be spanned in any complete mating extending M. If it returns* `success` *then M is complete.*

PROOF. The first assertion is a consequence of the remarks following Algorithm 3.5.10, using Lemmas 3.5.6 and 3.5.8. The remaining assertions follow from Lemma 3.5.12. □

3.6. Minimality

The above results also give us a more efficient way to test for the minimality of a mating, which we now discuss.

DEFINITION 3.6.1. *A mating M is said to be* minimal *if for every connection $c \in M$ there is some vertical path P_c such that c spans P_c and $M \setminus \{c\}$ does not span P_c.*

We observe that, in the case where M is complete, this definition accords with our notion of a *minimal complete mating*: if M is complete and minimal, then for all $c \in M$ we have that $M \setminus \{c\}$ is not complete.

We also have the following result:

LEMMA 3.6.2. *If $\sim_J$ is defined as in Definition 3.1.2, and M is a minimal component, then M is a minimal mating.*

PROOF. No two literals in any fragment F can lie on the same vertical path, and a component is minimal for F iff it touches every literal of F and no subset of it touches every literal of F. Hence if M is a minimal component for F, then every connection $c = (l_1, l_2)$ in M must span some vertical path which is not spanned by the other connections in M, since without loss of

generality $l_1 \in F$ and no other connection in M can possibly span *any* path through l_1. □

The following lemma has important implications for our search procedure:

LEMMA 3.6.3. *If M_1 is a non-minimal mating and M_2 is any mating, $M_1 \cup M_2$ is not minimal. Furthermore, there exist minimal matings M_1 and M_2 such that $M_1 \cup M_2$ is not minimal.*

PROOF. Trivial. □

This shows that minimal matings are not closed under taking unions. Our search procedure begins with minimal components and takes unions in the hope of finding a minimal complete mating. Given the above lemma, this procedure may construct a non-minimal mating at some point, and this mating will in turn generate more non-minimal matings, without any hope of ever producing a minimal complete mating.

We'd like to suppress these non-minimal matings; the lemma above shows that they can be removed without affecting the completeness of the search. (Note that the completeness proof, Theorem 4.3.2, shows that they can also be allowed to survive without affecting the completeness of the search; all that we need is the existence of an increasing chain of minimal matings, starting with a minimal component and ending with a complete mating.)

Testing for minimality is expensive, and experiments with the component search procedure have shown that very few non-minimal matings are generated (in fact, most searches do not generate any non-minimal matings at all). Furthermore, small matings are very unlikely to be non-minimal, and so the reduction in the search space will not be particularly great — the matings which are pruned out are not near the root of the search tree. These two observations mean that blindly testing matings for minimality usually wastes more time than it saves. However, it may be possible to restrict the testing to certain special cases, in which case the following algorithm might be useful.

ALGORITHM 3.6.4. *To decide whether a mating M already spans all of the vertical paths in J through a connection $(l_1, l_2) \notin M$:*

1. *Construct the jform J_1 from M exactly as in Lemma 3.5.12 (so J_1 will be a union of a minimal set of hpath-closures containing all of M).*
2. *If $l_1 \in J_1$, then delete every literal $l \in J_1$ such that l and l_1 lie on some horizontal path and $l \neq l_1$. Normalize the resulting jform and call it K (if $l_1 \notin J_1$, put $K = J_1$. Note that if $l_2 \in J_1$ then $l_2 \in K$, since l_1 and l_2 lie on some vertical path).*
3. *If $l_2 \in K$, then delete every literal $l \in K$ such that l and l_2 lie on some horizontal path and $l \neq l_2$. Replace K with the normalization of the resulting jform (if $l_2 \notin K$ then leave K unchanged).*

4. *For each literal $l_i \in K$ touched by M, if the hpath-closure $l_i[K]^*$ contains an untouched path (i.e. a path $(l_{i_1}, \ldots, l_{i_k})$ such that none of the literals $(l_{i_1}, \ldots, l_{i_k})$ is touched by M), then delete every literal $l \in J$ in $l_i[K]^*$. Replace K with the normalization of the resulting jform.*
5. *If K is empty (every literal has been deleted), then return* `false`.
6. *Let M' be $M \restriction_K$, which is to say $\{(c_1, c_2) \in M \mid c_1 \in K \wedge c_2 \in K\}$.*
7. *Return* `true` *if M' is a complete mating for K, and* `false` *otherwise.*

LEMMA 3.6.5. *The above algorithm will return* `true` *iff M spans all paths which are spanned by (l_1, l_2).*

PROOF. It is clearly enough to restrict the jform to just those paths which contain l_1 and l_2, and to check whether the restriction of M to this new jform is complete.

In step 1, we restrict the jform to J_1, for exactly the same reasons as in Lemma 3.5.12: if there is an open path through (l_1, l_2) then it must consist of an open path in J_1 concatenated with some open path in J_2 (all paths in J_2 are open, but we must choose an open path in J_2 passing through $\{l_1, l_2\} \cap J$ if this set is non-empty).

So there is an open path in J passing through (l_1, l_2) iff there is an open path in J_1 passing through whichever of $\{l_1, l_2\}$ occurs in J_1.

In steps 2 and 3, we discard all literals which do not occur on vertical paths through l_1 and l_2.

In step 4, we discard any closure having an untouched path. This is reasonable, since if P is such an untouched path, then any path P' through (l_1, l_2) has a variant which passes through P and is otherwise the same as P'. Since no literal in P is even touched by the mating, the variants of P' can only be spanned if they are spanned outside of this closure, in which case P' is also spanned outside of the closure.

If nothing is left, then there are clearly open paths through (l_1, l_2) (either M was empty, or we found a set of untouched paths allowing us to discard every hpath-closure, in which case the concatenation of these paths is open). So it is correct to return `false` at this point.

Otherwise, by the preceding argument, the existence of an open path through (l_1, l_2) in J (meaning that we ought to return `false`) is equivalent to the existence of an open path in K (meaning that M' is not complete for K). Thus the final step is correct. □

To apply this algorithm during the search, we must take each new mating M, and for each $c \in M$, decide whether $M \setminus \{c\}$ spans all of the paths which are spanned by c. So testing a single mating of n connections will require n calls to the algorithm. As previously remarked, this is likely to be too expensive to be of any practical use.

CHAPTER 4

The Component Search Method

4.1. The Search Algorithm

Before beginning the search, we modify the search space by duplicating quantifiers and applying substitutions for set variables, within the bounds that have been set by the user. The resulting jform J is then intended to remain fixed for the rest of the search, although we do include (purely for the proof of completeness) a step in the algorithm which returns to amplify the jform again if all else fails. The algorithm is thus in fact just another iterative deepening or level-saturation proof procedure, although it avoids most of the problems usually associated with such an approach by applying many quantifier duplications at once and controlling the resulting redundancy by using methods which will be described in Chapter 6. Nonetheless, we remark that this search algorithm is designed specifically to work well with the unification methods described in Chapter 5, and that it effectively accepts a modest reduction in the efficiency of the search for a mating in exchange for a substantial improvement in the efficiency of unification. For higher-order searches this is a good trade-off to make (since unification accounts for the vast majority of the time spent searching for a proof), but for first-order or propositional searches (where unification is not a major problem, and optimal search methods are highly desirable) it may be less desirable. The algorithm which we describe will still work in these cases but other, potentially better, methods for first-order searches are available, as discussed in Sections 2.6 and 4.7.

We search for a complete mating in J by constructing first $\sim_J$, then the fragments F, then the minimal components and finally taking unions until we reach a complete mating. We will decide which components to take unions of by using the completeness checker described in Section 3.5.

ALGORITHM 4.1.1. *The basic search algorithm is as follows:*

Suppose we are given a jform J. We will create a list of connections in J; from this we construct a list of minimal components for the fragments of J. These minimal components will be divided into two lists: *current*, the list of components which are being worked on, and *stored*, the list of all other minimal components. Recall that the term *combine* has a specific meaning, given in Definition 3.3.2.

1. Fix all of the search bounds (unification depth, number of quantifier duplications and so on). Amplify J, by applying substitutions for set

variables, duplicating outermost quantifiers, and (optionally) hiding equalities that can be handled by the unification algorithm, to produce a new jform J'.

2. Check each potential connection in J' for unifiability. If it is not unifiable, it will never be considered again; if it is unifiable, then store it as an *acceptable connection*.
3. Divide J' into fragments using the $\sim_J$ relation defined in Definition 3.1.2. For each fragment, use the connections discovered in the previous step to enumerate all of the minimal components which block that fragment.
4. Choose one fragment f, and divide the minimal components into two lists: *current*, the list of all components blocking f, and *stored*, the list of all other components. Verify the unifiability of (at least) the components in *current*, discarding any non-unifiable components. If no components in *current* are unifiable, this fragment cannot be blocked; delete all minimal components which touch it[1], choose a new fragment and repeat this entire step.
5. For each element c of *current*:
   (a) If c is weak, choose a fragment which is weak in c and combine c with the set of all the *stored* components which block this fragment.
   (b) Otherwise, if c is strong, check for completeness. If it's complete, we're done; return c. If not, then the completeness check will return either a list of untouched fragments, one of which must be blocked, or an open path. If it returns a list of fragments, then combine c with the set of all the *stored* components which block any of these fragments. If it returns an open path, combine c with the set of all the acceptable connections on this path.
6. If some new components were produced in step 5, set *current* to be the list of components produced, and return to step 5.
7. Otherwise, no complete mating can involve the fragment chosen in step 4. Delete all components touching this fragment, and return to step 4 to choose another starting fragment (unless no components remain, in which case either fail or return to step 1 and expand J' further).

The reader will observe a great deal of nondeterminism in this algorithm. In particular, we do not specify any of the details of step 1, nor how to choose a fragment in steps 4 and 5 (though the set of support strategy is an obvious candidate for step 4; see Section 4.5), nor how the completeness check works, nor even how potential unifiability is to be determined. The completeness check has already been discussed (see Theorem 3.5.13), and all of the other issues will be discussed later. The intention is that J' should

[1]By Lemma 3.1.8, if we were ever to touch f then we would have to block it, and we have just discovered that it cannot be blocked, so we are justified in deleting all components which touch it.

be a large enough amplification of J to contain a complete mating, so that we will never return to step 1 again; the option to return is given so that we can obtain a completeness result. If we do return to step 1, we may still be able to preserve some of the information which was discovered on the previous iteration, depending on what exactly is changed in step 1 of the new iteration; this is discussed in Section 5.5 and Section 10.2.

In step 3 of the search, we may discard any minimal component C which blocks only a single-literal fragment A and touches any multiple-literal fragment B. In this case, C contains a single connection, and by Lemma 3.1.8, any complete mating involving C must also block B. Hence C will either not be a part of any complete mating, or will be subsumed by some minimal component blocking B.

4.2. Enumerating Connections and Minimal Components

Enumerating connections is a trivial matter of taking every pair of literals, eliminating those which do not lie on any vertical path, and then attempting to unify the rest. The number of connections will generally be quite small, and since allowing a non-unifiable connection to slip by will greatly increase the number of minimal components, we do not allow the user to delay this particular unification check.

Given the definition of $\sim_J$ in Definition 3.1.2, in which all of the literals in a fragment lie on a single horizontal path, it is comparatively trivial to enumerate the minimal components. We can easily find all of the acceptable connections which block a particular literal, and the minimal components are simply all of the ways to choose one such connection for each literal in the fragment.

For definitions of $\sim_J$ which do not have this property (see Section 8.2), we first enumerate the horizontal paths through the fragment and then do exactly the same thing for each horizontal path.

In either case, we must then check the resulting component for unifiability. At the very least, we check that the connections in the component are pairwise unifiable. Pairwise unifiability of connections is stored in a lookup table, since this is a quick way to eliminate many of the larger non-unifiable components without having to attempt to unify them completely. Depending on the parameters which have been set by the user, we may also check that the entire component is unifiable, or even explicitly compute its unifiers; in general, we delay this step until the component is about to be added to the *current* list, or is one of the *stored* components chosen in step 5, since it is computationally expensive, and there may be many minimal components that are never needed by the search.

We now turn to the soundness and completeness of this algorithm. For the remainder of this chapter, we assume that unification contains no equality reasoning, and hence that no equalities are hidden by step 1. We also assume, for the sake of clarity in the proof of completeness, that if we reach step 7 and no components remain, we will choose to fail.

4.3. Soundness and Completeness

THEOREM 4.3.1 (Soundness). *The above algorithm is sound.*

PROOF. If the algorithm terminates without failing (which can only happen at step 5), then the result is a complete and unifiable mating. Hence the procedure is trivially sound, by previous results about the matings method. □

THEOREM 4.3.2 (Restricted Completeness). *The above algorithm is complete, in the sense that it will find a complete mating in J' if there is one to be found within the given search bounds.*

PROOF. First, we observe that if a component c occurs in *current*, then at step 5 it will be replaced by a (possibly empty) set of new unifiable components, each of which represents a proper superset of the connections of c. This is clear, since step 5 will always either block a previously unblocked fragment, or add a connection to an open path. Hence it is not possible to repeat step 5 forever; either the components will grow until one eventually becomes complete (there being only a finite number of acceptable connections), or they will be discarded.

Suppose the possible complete matings in J' (within the current search bounds) are enumerated as $\{M_i \mid 0 \leq i < n\}$. We assume that there is a complete mating, and hence $n > 0$. Suppose also that the completeness check is known to be correct. Let f be the fragment chosen in step 4. By Lemma 3.1.8, we have that either f is blocked by some M_i or f is not touched by any M_i. In the latter case, no component blocking f can possibly be extended to a complete mating, and by the reasoning above we deduce that step 5 must eventually fail on all elements of *current*, and so we will proceed to step 7, and return to choose a new starting fragment. We cannot ever choose the same starting fragment twice, since all components touching a failed fragment are deleted as soon as the fragment fails, and we may only choose a starting fragment for which we have minimal components.

Hence (since each M_i must block some fragment) we will eventually manage to choose a fragment in step 4 which is blocked by some M_i. Since the minimal components enumerate all possible unifiable ways of blocking this fragment within the current search bounds, at least one of them must be a subset of some M_i, and will be added to *current*.

Now we show that if *current* contains a component c which is a subset of some mating M_i, then after one iteration it will terminate if $c = M_i$, and otherwise will contain a new component c' such that $c \subset c' \subseteq M_i$. Since M_i is finite, this is sufficient to complete the proof.

The former case is trivial. For the latter case, if some fragment is weak in c then, by Lemma 3.1.8, any complete mating extending c must block that fragment. Moreover, some minimal component d in *stored* must represent the way in which that fragment is blocked in the mating M_i (since, again, the minimal components enumerate all possible ways of blocking a

fragment). So the component $c' = c \cup d \subseteq M_i$ will be generated in step 5 and added to *current*. If the completeness check (which we proved correct in Theorem 3.5.13) gives us a list of unblocked fragments, at least one of which must be blocked, then we know that one of these fragments is blocked in M_i, whence by the same argument we produce c' as required. Finally, if the completeness check provides us with an open path, then this path must be spanned in M_i by some acceptable connection, and we have enumerated all possible acceptable connections, so once again we must produce c' as required. □

Note: Since there are bounds on the depth of unification, the number of quantifier duplications, and so on, and since we have chosen to fail at step 7 rather than return to step 1, the algorithm is not complete in the more general sense. However, if we suppose that we always opt to return to step 1, and that all the search bounds are relaxed each time we pass step 1, then we have the following:

THEOREM 4.3.3 (Completeness). *If there is a complete mating in some amplification of J, then the above algorithm will terminate with a complete mating.*

PROOF. (Sketch) Let J^* be the smallest amplification of J in which a complete mating can be found. By the remarks above, if J' contains no complete mating then the algorithm will eventually reach step 7, return to step 1, and J' will be replaced by some J''. Similarly, J'' will be replaced by $J^{(3)}$, and so on. If we have relaxed the search bounds in an appropriate way, eventually we will reach some step where $J^{(n)}$ contains a copy of J^* and there is a complete mating within the search bounds (i.e. the unification depth is as large as necessary, etc.). Then, by the restricted completeness theorem above, a mating will be found. □

4.4. The Weighted Search Algorithm

The algorithm above is a strictly breadth-first search; given a partial mating, it will consider all extensions of that mating during the next step of the procedure. We may weight the components in order to give preference to extensions having particular properties, as follows.

Suppose we have a measure function m from components to integers, such that components with higher measure are to be preferred to those with lower measure. The measure is assumed to depend only on the component itself, and not on other features of the search. Each time a component C is constructed, we label it with the measure $m(C)$.

ALGORITHM 4.4.1. *The search algorithm with weighted components is as follows:*

Suppose we are given a jform J.

1. Fix all of the search bounds (unification depth, number of quantifier duplications and so on). Amplify J, by applying substitutions for set variables, duplicating outermost quantifiers, and (optionally) hiding equalities that can be handled by the unification algorithm, to produce a new jform J'.
2. Check each potential connection in J' for unifiability. If it is not unifiable, it will never be considered again; if it is unifiable, then store it as an *acceptable connection*).
3. Divide J' into fragments using the $\sim_J$ relation defined in Definition 3.1.2. For each fragment, use the connections discovered in the previous step to enumerate all of the minimal components which block that fragment.
4. Choose one fragment f. Let k be the maximal measure of the components which block f. Divide the minimal components into three lists: *current*, the list of all components blocking f and having measure k, *pending*, the list of all components blocking f and having measure less than k, and *stored*, the list of all other components. Verify the unifiability of (at least) the components in *current*, discarding any non-unifiable components. If *current* becomes empty but there are components remaining in *pending*, compute the new value of k and move components from *pending* to *current* as appropriate. If both *current* and *pending* become empty, this fragment cannot be blocked; delete all minimal components which touch it, choose a new fragment and repeat this entire step.
5. For each element c of *current*:
   (a) If c is weak, choose some fragment which is weak in c. Combine each of the *stored* components which blocks this fragment with c, to produce a list of new components. Discard any elements of this list which are not unifiable.
   (b) Otherwise, if c is strong, check for completeness. If it's complete, we're done; return c. If not, then the completeness check will return either a list of untouched fragments, one of which must be blocked, or an open path. If it returns a list of fragments, then combine c with each of the stored components which blocks any of these fragments. If it returns an open path, combine c with each of the acceptable connections on this path.
6. If some new components were produced in step 5, set *current* to be the list of components produced. Compute the maximal weight k of components in *current*, and move all components in *current* which have weight less than k to the list *pending*. Return to step 5.
7. Otherwise, if *current* is empty but *pending* is not, move all the components in *pending* of maximal weight to the list *current*, and return to step 5.
8. Otherwise, if both *current* and *pending* are empty, no complete mating can involve the fragment chosen in step 4. Delete all components

touching this fragment, and return to step 4 to choose another starting fragment (unless no components remain, in which case either fail or return to step 1 and expand J' further).

Note that if m is a constant function, then *pending* will remain empty throughout, and this algorithm will be equivalent to Algorithm 4.1.1. We introduce a method for pruning the branching of the search in general (and at step 5(b) in particular) in Section 8.5.

In step 6, we chose to only look at the maximal weight of the (newly-generated) components in *current*. There may be components in *pending* which have equal or greater weight, and these could also be considered by modifying the algorithm in the obvious way.

Weighting the components helps to prevent combinatorial explosion of the number of components in the early stages of the search; it does so at the expense of reintroducing a small amount of depth-first search and backtracking.

The soundness and completeness of Algorithm 4.4.1 follow exactly as for Algorithm 4.1.1.

There are many possible choices for m (see Section 8.4 and the discussion of the flag MS98-MEASURE in Appendix A), but experimentation has shown that the very simple measure in which $m(C)$ is the number of fragments blocked by C is effective in a wide variety of searches.

4.5. Choosing a Fragment

At a number of points in the search algorithms, a fragment is nondeterministically chosen. Obviously, the choice of fragment can dramatically affect the progress of the search, and so we briefly consider how best to choose a fragment. More information about choosing a fragment is given in Section 8.3.

During the innermost loop of the algorithm, we must select one fragment which is weak in c. In many cases the choice will be forced, since there will only be one such fragment; if there is more than one then it seems reasonable to select the one which will cause the least branching in the search. We know the number of minimal components blocking each fragment, and in principle we will choose the fragment having the fewest minimal components. In practice, we need to take account of the fact that some fragments are duplicate copies of other fragments, and so we will create a list of fragments in which all of the original copies occur before all of the first duplicates, which occur before all of the second duplicates, and so on. Within this constraint, we then sort the list so that fragments having fewer minimal components occur earlier in the list; our choice of unblocked fragment will then be whichever occurs earliest in the list.

Before the innermost loop, we are required to choose a starting fragment. The same considerations about minimizing branching apply to this choice as well, but since this fragment will occur in all of the components generated by the innermost loop (until we either fail or reach a complete mating) now

we must also consider the likelihood of any particular fragment being a part of a complete mating. In particular, we will avoid (if possible) choosing any fragment which lies below a set-variable substitution, since this would initially restrict our search to only those matings involving that substitution. As previously remarked, choosing a starting fragment using the set of support strategy is a good heuristic to try here, and has the useful property of making the search more goal-directed than was possible with path enumeration.

4.6. Pruning out Redundant Matings

Since we may have many duplicate copies of the same quantifier, it is possible to create essentially the same mating in many ways, by permuting these duplicate copies. Of course, we would like to prune away these redundant copies of the same mating, and we will address this problem in Chapter 6.

4.7. Related Work

This algorithm shares some ideas with other proof methods in the literature, which we now discuss.

The hyperlinking method of [Lee and Plaisted, 1992; Lee and Plaisted, 1989] is a complete strategy for the first-order predicate calculus; it forms the basis of the CLIN theorem prover [Lee, 1990]. It is similar in spirit to the linked conjunct procedure of [Gilmore, 1960], although it avoids the blind substitution of the latter (and hence also avoids the combinatorial explosion of the search space caused by the latter's level-saturation search of the Herbrand universe). Hyperlinking works for first-order problems in clause-normal form, as follows: given a clause $C = \{L_1, \ldots, L_n\}$ in a connection graph, a hyperlink is a set of connections $\{(L_1, M_1), \ldots, (L_n, M_n)\}$ such that for some substitution Θ the literals $\Theta(L_i)$ and $\Theta(M_i)$ are complementary for each i. For each C, all such hyperlinks are computed. For each hyperlink, the most general substitution Θ is generated, and $\Theta(C)$ is added to the set of clauses. The procedure then checks for unsatisfiability of the resulting clause set using a modified Davis-Putnam procedure [Davis and Putnam, 1960], and repeats the above as necessary.

The hyperlinking procedure requires the problem to be in clause-normal form, and more importantly it relies on the existence of a most general unifier. Our minimal components are rather similar to hyperlinks; however, the search procedure itself is not very similar, and cannot be made so, because higher-order unification does not usually produce most general unifiers.

Hyperlinking in first-order logic also allows one to bypass the problem of quantifier duplication, and a similar approach is taken by [Quigley, 1998], in which the number of duplications for a quantified variable is governed by the number of potential substitutions for it on all the vertical paths through the jform. Such an approach would almost certainly be disastrous in a higher-order problem; there would be too many incomparable substitutions.

The hyper-tableau method of [Baumgartner *et al.*, 1996] is another related approach, similar both to hyper-linking, and to the disconnection method [Billon, 1996]. Hyper-tableaux are analytical (no new clauses are added, except for substitution instances), and the substitution instances are generated locally to a particular branch (as opposed to resolution, in which new clauses are global; conceptually, this is somewhat similar to path-focused duplication[Issar, 1991]). Hyper-tableaux were originally required to blindly guess ground instantiations for some clauses, but this requirement has since been replaced by a unification-driven method [Baumgartner, 1997] which relies on the existence of a most general unifier.

CHAPTER 5

Unification

This chapter deals with the problem of higher-order unification for the search algorithm presented above. First-order unification, which is of course decidable in linear time [Paterson and Wegman, 1978], is handled differently; unification of partial matings is done in the same manner as for every other search procedure.

In the path-enumeration based mating searches in TPS, pre-unification is performed incrementally after each new connection is added. This has the advantage of potentially making unification less expensive (in that the new connection may already be unified, or nearly unified, by the partial substitution already generated). However, it also means that the same connection may be unified many times, in many contexts, and that unification is required to backtrack when a connection is discarded. Lastly, very little of the information generated by unification can usefully be stored, since it generally refers only to the immediate situation, which will not recur during the search. (Some of the path-enumeration searches do test a connection for unifiability on its own, and then ignore all connections which are known not to be unifiable, but this is the only information about unification which they can usefully store.)

Given the undecidability of higher-order unification [Huet, 1973; Goldfarb, 1981], we must have bounds on unification during the search. Initially, these bounds were on the depth of the entire unification tree; however, it has been discovered that a uniform bound on the complexity of the substitution for each free variable is a much more efficient approach. It is generally not possible to increase these bounds during the search, since doing so would require reconsidering almost every connection and partial mating which had previously been discarded on grounds of non-unifiability.

If we accept, then, that there will be fixed bounds on the unification search, and that these will be bounds on the complexity of the substitution for each free variable, then it seems pointless to generate unifiers incrementally as new connections are added. Since we know the constants which occur in the search space[1], and we know the types of all the free variables, we can generate all of the substitutions for a given free variable before we even begin the search. We can then apply these substitutions as we check

[1] In addition to the constants of the search space, we will also need at most one new constant at each base type, if the search space does not contain a constant at that type, to complete the unification of any remaining flex-flex pairs.

the unifiability of each connection and record which (if any) substitutions unify the connection. Unification of partial matings would then be reduced to combining these records in an appropriate way. Notice that the effort required to compute these substitutions will not increase very fast as the size of the search space increases: a large problem in (for example) group theory will probably not have many more types of free variables than a small problem in group theory, and the constants in both cases will be e, $*$, $^{-1}$, and any required skolem constants.

5.1. Unifiers as Directed Graphs

We begin by constructing all of the possible substitutions for each variable type, up to a fixed depth, enumerating them, and storing them in a hash-table. This construction uses a standard version of Huet's algorithm [Huet, 1975; Snyder and Gallier, 1989], in which flex-flex pairs are unified (rather than being ignored). Using this algorithm, we unify $\langle y_\tau, y_\tau \rangle$ for each τ which is the type of some free variable in the jform. The substitutions thus enumerated will be just the ground substitutions down to a particular depth bound.

Any other method of constructing these substitutions would do as well, if it could enumerate all of the substitutions that would be given by Huet's algorithm down to a given bound. Our existing implementation of Huet's algorithm introduces negations where necessary, as described in [Andrews *et al.*, 1996], and so does this new implementation. Connectives and quantifiers will normally be introduced by set-variable substitutions, and will only be used in the unification substitutions if they occur within some literal in the search space.

We now have a finite set S of ground substitutions, and we fix an ordering $S = t_1, t_2, \ldots, t_m$. We also fix a total ordering $v_1, \ldots, v_n$ on the variables of the search space. For each type τ, let $S_\tau \subseteq S$ be the set of substitution terms of type τ, and let $n_\tau = |S_\tau|$.

We shall often refer to a substitution by its number: we can write $\{x_\tau \mapsto 17\}$ rather than $\{x_\tau \mapsto t_{17}\}$ (both of which will be taken to imply that $t_{17} \in S_\tau$).

These substitutions are used to determine unifiability of a connection in the initial stages of the search. The set of successful unifiers of each connection is stored as a directed acyclic graph, called a *unification graph*. Unification graphs strongly resemble the ordered binary decision diagrams of [Bryant, 1986]; the major differences are that unification graphs are "levelized" (all nodes at the same depth are labelled with the same variable), that a node may have many out-arcs rather than just two, that one of the two terminal nodes is omitted, and that we introduce a small amount of nondeterminism into the arc labelling. Each of these points will be explained in more detail in what follows.

To motivate our discussion of unification graphs, we first consider a decision tree for substitutions.

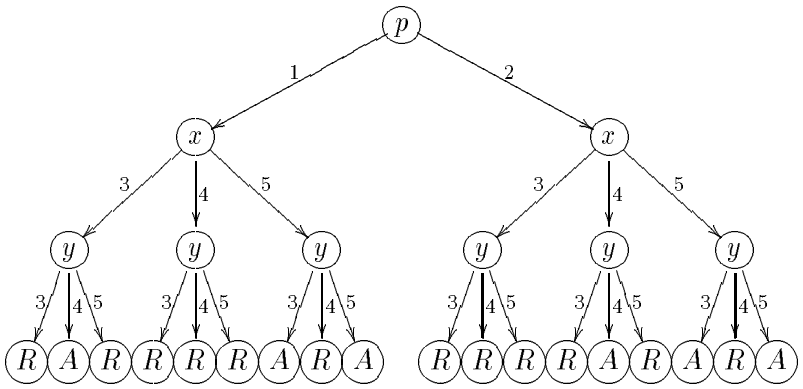


FIGURE 5.1. A decision tree for substitutions

DEFINITION 5.1.1. *A* decision tree for substitutions *is a rooted, directed tree having the following properties:*

- *The root node is labelled* v_1.
- *For* $i < n$, *every node labelled* v_i *is nonterminal, and its children are all labelled* v_{i+1}.
- *Nodes labelled* v_n *are also nonterminal, and their children are terminal nodes labelled either* ACCEPT *or* REJECT.
- *If* v_i *has type* τ, *then for each substitution* $s \in S_\tau$, *each node labelled* v_i *has an out-arc labelled* s.
- *There are no other nodes or arcs besides those specified above.*

The intention of this definition, of course, is that such a tree should store all of our knowledge about some particular unification problem. A path from the root to a terminal node of this tree should represent a substitution in the obvious manner: at each node, the choice of substitution for the variable at that node is represented by the choice of out-arc to be followed. Eventually, we will arrive at a terminal node labelled either ACCEPT or REJECT, which will tell us whether the substitution is acceptable or not. By the definition, every possible substitution up to our depth bounds is represented in the tree.

EXAMPLE 5.1.2. *Suppose there are only three variables* $p_{o\alpha}$, x_α *and* y_α *in our search space, and that* $S_{o\alpha} = \{t_1, t_2\}$ *and* $S_\alpha = \{t_3, t_4, t_5\}$. *We write* A *for* ACCEPT *and* R *for* REJECT. *Then Figure 5.1 is a decision tree for substitutions.*

DEFINITION 5.1.3. *Given a decision tree* T, *if the set ofsubstitutions which it accepts (i.e. the paths which lead to a node labelled* ACCEPT*) is exactly* Σ, *we say that* T *is* a decision tree for the substitutions Σ.

Figure 5.1 is thus a decision tree for the substitutions $\{\{p \mapsto t_1, x \mapsto t_3, y \mapsto t_4\}, \{p \mapsto t_1, x \mapsto t_5, y \mapsto t_3\}, \{p \mapsto t_1, x \mapsto t_5, y \mapsto t_5\}, \{p \mapsto t_2, x \mapsto t_4, y \mapsto t_4\}, \{p \mapsto t_2, x \mapsto t_5, y \mapsto t_3\}, \{p \mapsto t_2, x \mapsto t_5, y \mapsto t_5\}\}$.

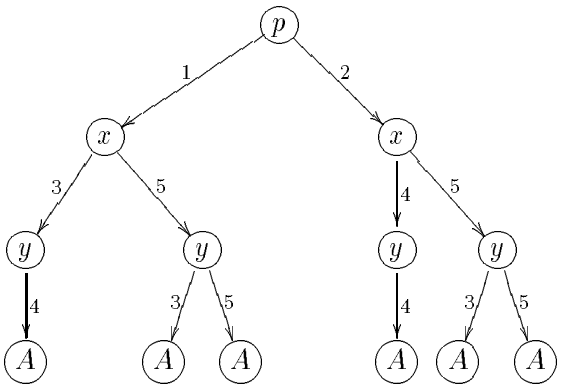


FIGURE 5.2. The result of applying *Prune* to the tree in Figure 5.1.

We are generally not interested in the substitutions which fail to unify a particular set of disagreement pairs, and so we may remove all of the paths in the tree which reach a node labelled REJECT. We recursively prune all such paths in a decision tree as follows:

ALGORITHM 5.1.4 (*Prune*).

1. *Remove all terminal nodes which are not labelled* ACCEPT, *and all in-arcs at those nodes.*
2. *Repeat the previous step until every terminal node in the tree is labelled* ACCEPT.

EXAMPLE 5.1.5. *The result of applying* Prune *to the tree of Figure 5.1 is, as expected, the tree in Figure 5.2.*

Furthermore, the tree may contain many isomorphic subtrees. For example, if one variable v_i of type τ is not used in the disagreement pairs, then all nodes labelled v_i will have n_τ children, and all of them will be isomorphic (since our choice of substitution for v is irrelevant). We recursively remove isomorphic subtrees as follows:

ALGORITHM 5.1.6 (*Reduce*).

Remove duplicate terminals: *Eliminate all but one of the terminal* (ACCEPT) *nodes, and redirect all arcs to the eliminated nodes to the remaining one.*

Remove duplicate nonterminals: *Let N and N' be nonterminal nodes. If there is a bijection f between the out-arcs of N and those of N', such that f preserves both the labels on the out-arcs and the child nodes to which those arcs lead, then eliminate N' and redirect all of its in-arcs to N.*

Repeat: *Repeat the previous step until no such bijections can be found.*

Applying *Reduce* does not change the set of substitutions which are represented by the graph, since arcs can only be redirected from one subgraph to an isomorphic subgraph.

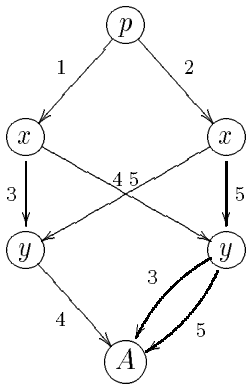


FIGURE 5.3. The result of applying *Reduce* to Figure 5.2.

This is similar to the variant of the linear-time *Reduce* algorithm[Bryant, 1986] which is used to produce levelized ordered binary decision diagrams ("levelized" because all of the nodes at depth k are labelled with the variable v_k, which happens because we have omitted the step of *Reduce* which removes nodes whose out-arcs all point to the same child node).

EXAMPLE 5.1.7. *The result of applying* Reduce *to the tree of Figure 5.2 is the graph in Figure 5.3. It is clear that this graph represents the same set of substitutions as Figure 5.2.*

REMARK The resulting graph is a canonical representation of a given set of substitutions. It can also be viewed as a deterministic finite automaton, by identifying the unifying substitutions with the strings of n integers which represent them; this is a finite language, and hence regular. Therefore there is a minimal deterministic finite automaton which accepts this language; this is essentially the graph which we have produced. Moreover, given two languages L_1 and L_2 with automata M_1 and M_2, the accepting states of $L_1 \cap L_2$ are exactly the accepting states of the product automaton. This suggests a way to compute the intersection of two sets of unifiers given the graphs which correspond to them.

However, we will not use graphs in this form to represent our unifiers; we have one further generalization to make, which we now attempt to motivate. Unification graphs will be used to represent the unifiers of individual connections in the search space, and we wish to intersect them (as was suggested in the remarks above) to represent the unifiers of larger sets of connections. Single connections, however, generally contain very few of the variables in the search space, and so there will be many nodes for which the out-arcs represent all of some S_τ, and all of these arcs lead to the same child node.

Since S_τ may be very large, this has unfortunate implications both for the space required to store the graphs, and for the time taken to intersect them. We avoid this problem by introducing an extra arc label, 0 (recall that

our existing arc labels are 1 to m). An out-arc labelled 0 from a node whose variable is of type τ will represent "any substitution in S_τ", and is treated as a substitution which is compatible with any of the substitutions in S_τ (of course, the substitutions in S_τ are pairwise incompatible, since they are all ground).

The introduction of 0 arcs makes the graphs much more concise, but unfortunately makes the graphs non-canonical (since any 0 arc can be replaced with n_τ arcs, for the appropriate τ, in the obvious manner). This makes it possible that we will fail to notice that two subgraphs represent identical substitutions; however this will not happen very often, for reasons that will become clear later.

From the standpoint of automata, 0 arcs also introduce nondeterminism, since 0 need not be the only out-arc of a node. To illustrate this point, suppose that we are constructing a graph for the unifiers of $\langle p_{o\iota} x_\iota, Q_{o\iota} A_\iota \rangle$ in a search space whose variables are p and x, and whose constants are $Q_{o\iota}$ and $A_\iota, B_\iota, \ldots, Z_\iota$. For x we have the obvious substitutions, numbered 1 to 26, and for p we have found the substitutions $\lambda x.QA$ (27) and $\lambda x.Qx$ (28). Our unifiers will therefore be $\{p \mapsto 27\}$ and $\{p \mapsto 28, x \mapsto 1\}$. If our variable ordering puts p before x, then the unification graph will be the (entirely deterministic) graph of Figure 5.4(a). However, if we order the variables with x first, then either both out-arcs of the root node accept the substitution 1 (Figure 5.4(b)) and the graph is nondeterministic, or we avoid using 0-labelled arcs (Figure 5.4(c)) and graph is deterministic but wasteful of space.

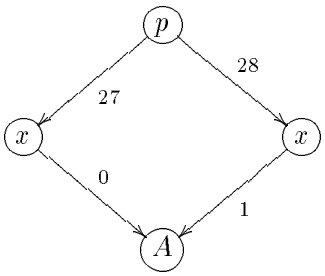
Experimentally, it appears that in almost all cases a reasonable choice of variable order will eliminate any nondeterminism, although we must still construct our algorithms to deal correctly with the case in which 0 is not the only out-arc.

Recall that the variables $v_1, \ldots, v_n$ and the substitutions S_τ are assumed to have been enumerated; we are now able to define a unification graph.

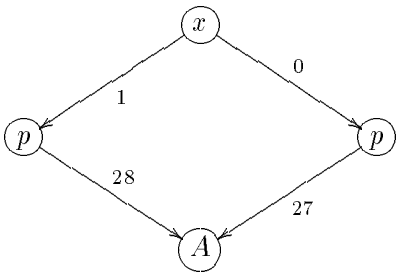
DEFINITION 5.1.8. *A* unification graph *U is a rooted directed acyclic graph with labelled nodes and arcs, satisfying the following conditions:*

1. *The root node of U is labelled v_1.*
2. *For $1 \leq i < n$, the child nodes of a node labelled v_i are labelled v_{i+1}.*
3. *The child nodes of a node labelled v_n are labelled* ACCEPT.
4. *Nodes labelled with variables are nonterminal; nodes labelled* ACCEPT *are terminal.*
5. *If a node N has label v_i, the labels of its out-arcs form a nonempty subset of $S_\tau \cup \{0\}$, where τ is the type of v_i. In particular, no two of these arcs have the same label.*
6. *No two subgraphs of U are isomorphic.*

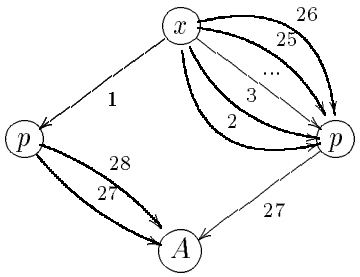
Clearly, applying the algorithms *Prune* and *Reduce* to a decision tree for substitutions will produce a unification graph U (which contains no 0-labelled



(a) $p < x$



(b) x<p ,
nondeterministic



(c) x<p ,
deterministic

FIGURE 5.4. Nondeterminism and 0-labelled Arcs

arcs; we shall introduce 0-labelled arcs in the implementation described below).

DEFINITION 5.1.9. *A* path *through a unification graph will be understood as meaning a path, in the ordinary graph-theoretic sense, which begins at the root node and ends at the* ACCEPT *node.*

Since the variable ordering is fixed, we now have a way to write every substitution σ as a list of n integers $(k_1, \ldots, k_n)$: for each i, if $\sigma(v_i) = v_i$ then put $k_i = 0$, otherwise we must have $\sigma(v_i) = t_j$ for some j, and so put $k_i = j$.

Hence we may identify a path with the list of substitutions on it. For example, the graph in Figure 5.3 contains the path $1, 3, 4$ (the leftmost path), corresponding to the substitution $(1, 3, 4)$ (i.e. $\{p \mapsto t_1, x \mapsto t_3, y \mapsto t_4\}$).

REMARK A path which contains 0-labelled arcs does not, technically, represent a ground substitution. Rather than introduce yet more notation, we shall continue to refer to substitutions σ as *ground* if they have the property that, for all variables x in our ordering, $\sigma(x)$ is either ground or

equal to x. This is reasonable, since σ is ground in our sense iff there is some partition of our variables V such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $\sigma \restriction_{V_1}$ is ground in the usual sense and $\sigma \restriction_{V_2}$ is the identity on V_2.

5.1.1. Implementation. The above definition of a unification graph is conceptually correct, but it is highly non-optimal as a method for actually constructing unification graphs. To create an efficient algorithm, we observe that for any variable x_τ and any set of ground substitutions Σ, we have:

$$\Sigma = \{\sigma \in \Sigma \mid \sigma(x) = x\} \cup \bigcup_{\{t_i \in S_\tau\}} \{\sigma \in \Sigma \mid \sigma(x) = t_i\}$$

This property is analogous to the *Shannon expansion* which is used for constructing OBDDs, and which states that if f is a Boolean function and x is a propositional variable, then $f = \bar{x} \cdot f|_{x \mapsto 0} + x \cdot f|_{x \mapsto 1}$.

NOTATION *We make no distinction in the following between a node N and the subgraph which has N as its root node; this is justified by the obvious bijection between the two.*

Our implementation constructs unification graphs as follows: begin with a global hashtable which will store all of the subgraphs of all of the unification graphs which are ever created. Into this hashtable, we insert a graph containing the single terminal node ACCEPT. Now suppose Σ is a set of ground unifiers for some set of disagreement pairs.

ALGORITHM 5.1.10. *To construct a unification graph for the substitutions Σ over the variables $x_1, \ldots, x_k$:*

1. *Let $s_1, \ldots, s_{n_\tau}$ enumerate S_τ, where τ is the type of x_1.*
2. *For all $1 \leq i \leq n_\tau$, let Σ_i be $\{\sigma \in \Sigma \mid \sigma(x_1) = s_i\}$.*
3. *Let Σ_0 be $\{\sigma \in \Sigma \mid \sigma(x_1) = x_1\}$.*
4. *If $k \neq 1$, then for all $0 \leq i \leq n_\tau$ such that $\Sigma_i \neq \emptyset$, construct the unification graph U_i for the substitutions Σ_i over $x_2, \ldots, x_k$.*
5. *If $k = 1$, then for all $0 \leq i \leq n_\tau$ such that $\Sigma_i \neq \emptyset$, let U_i be the node* ACCEPT *(retrieved from the hashtable — i.e. the same node in all cases).*
6. *If the hashtable contains a nonterminal node N whose out-arcs are labelled with $\{i \mid \Sigma_i \neq \emptyset\}$ and in which each arc labelled i leads to the subgraph U_i, then return that N, otherwise construct such an N, insert it into the hashtable and then return N.*

This algorithm constructs a unification graph in a depth-first manner, by repeatedly partitioning the set of substitutions. Every path through the resulting unification graph represents a ground unifier from Σ. Because the algorithm is depth-first, it will never construct two isomorphic nodes; all of the unification trees that are constructed will share structure wherever possible. Furthermore, the test for isomorphism only needs to check the labels on the out-arcs and the child nodes to which they lead, and need only check them for equality rather than isomorphism. In essence, we are constructing the pruned decision tree with 0-labelled arcs already in place,

and relying on the hashtable to implement the *Reduce* algorithm not only within a single tree, but over all the currently-defined trees.

In fact, our implementation uses an array of hashtables rather than a single hashtable; this array is indexed by the variable at the root node of the graph and by the number of out-arcs of the topmost node. There are n choices for the former (since the ACCEPT node is in fact implemented as a symbol, rather than a node, because it does not need to store any extra information), and $m' = 1 + \max\{n_\tau\}_\tau$ choices for the latter, and so we use an $n \times m'$ array of hashtables. This reduces the number of comparisons that must be made in order to determine whether a graph is already in the hashtable. We also generate a key for each graph (consisting of a pair of easily-computed numbers, representing the number of arcs in the graph and the number of paths through the graph), which again reduces the number of comparisons necessary for each new graph.

To reduce the space requirements somewhat, we do not label any of the nonterminal nodes with their variables. A node at depth k will always refer to the kth variable in our enumeration, and all of the functions we define on unification graphs can easily be made to keep track of the depth of the current node, so this represents no loss of information. The out-arcs of a node are stored as a list, in increasing order (this is one of the advantages of representing substitutions by natural numbers); this also reduces the amount of work required to compare two nodes.

5.2. Operations on Unification Graphs

We fix a variable ordering for the free variables of the search. For operations requiring more than one unification graph, we may therefore assume without further comment that all the unification graphs supplied will be defined with respect to the same variable ordering. Recall that the "substitution" 0 denotes "no substitution is made for this variable", and that all other substitutions are ground (and, hence, that every non-zero substitution is more general than 0, and any two non-zero substitutions are incompatible).

DEFINITION 5.2.1. *Two substitutions $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_n)$ are* compatible *iff for each i, either $a_i = b_i$ or $a_i = 0$ or $b_i = 0$. We write this as $\vec{a} \parallel \vec{b}$.*

Clearly, if the two substitutions are understood to be substitutions for the same list of variables (which will always be the case in any situation where we apply this definition), this is the same as the normal definition of compatibility of a pair of substitutions.

DEFINITION 5.2.2. *A unification graph U is said to* contain *a substitution $\sigma = (k_1, \ldots, k_n)$ if there is a path through U whose arcs are labelled $k_1, \ldots, k_n$ (in that order). We write this as $\sigma \in U$.*

DEFINITION 5.2.3. *A unification graph U is said to* accept *a substitution σ iff for some $\rho \in U$ we have $\sigma \parallel \rho$.*

Trivially, then, a unification graph must accept any substitution which it contains.

DEFINITION 5.2.4. *Two unification graphs U_1 and U_2 are said to be* compatible *if there are substitutions $\sigma \in U_1$ and $\rho \in U_2$ such that $\sigma \parallel \rho$.*

DEFINITION 5.2.5. *Let $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_n)$ be substitutions. Then $\vec{a}$ is* at least as general as *$\vec{b}$ iff for each i, either $a_i = b_i$ or $a_i = 0$. We write this as $\vec{a} \leq \vec{b}$.*

DEFINITION 5.2.6. *Let $\vec{a} = (a_1, \ldots, a_n)$ and $\vec{b} = (b_1, \ldots, b_n)$ be such that $\vec{a} \parallel \vec{b}$. The* merge *of $\vec{a}$ and $\vec{b}$ is given by $\mathrm{mrg}(\vec{a}, \vec{b}) = (c_1, \ldots, c_n)$ in which $c_i = a_i$ if $a_i \neq 0$, and $c_i = b_i$ otherwise.*

Clearly, in the above definition, $\vec{c}$ has the property that $\vec{a} \leq \vec{c}$ and $\vec{b} \leq \vec{c}$. Furthermore, for any substitution $\vec{d}$ having this property, we have $\vec{c} \leq \vec{d}$, and so $\vec{c}$ is the *most general common instance* of $\vec{a}$ and $\vec{b}$. (The condition that $\vec{a} \parallel \vec{b}$ is equivalent to asserting that such a $\vec{c}$ exists.)

Note that if $\vec{a}$ is a ground unifier for X and $\vec{b}$ is a ground unifier for Y, and these substitutions are compatible, then $\mathrm{mrg}(\vec{a}, \vec{b})$ is a ground unifier for $X \cup Y$.

Again, we will construct our algorithm by analogy with the methods of OBDDs. Bryant introduces a uniform algorithm *Apply* for computing all two-argument logical operations on Boolean functions, which proceeds as follows: let $\star$ be such a logical operation, f and f' two Boolean functions, and x a propositional variable. Then, using the Shannon expansion, we have $f \star f' = \bar{x} \cdot (f|_{x \mapsto 0} \star f'|_{x \mapsto 0}) + x \cdot (f|_{x \mapsto 1} \star f'|_{x \mapsto 1})$, and we may use this to construct the OBDD for $f \star f'$ from the OBDDs for f and f'.

For our purposes, we are only really interested in taking intersections of graphs. Suppose we have two graphs that contain no 0 arcs, so that the substitutions for each variable are compatible iff they are identical. Let Σ and Γ be the two sets of ground substitutions represented by these graphs, and let x be a variable. If we make the following definition:

$$\Delta|_{x \mapsto s} =_{\mathrm{def}} \{\delta_x \mid \delta \in \Delta \wedge \delta(x) = s\}$$

where

$$\delta_x(y) = \begin{cases} \delta(y) & \text{if } y \neq x \\ y & \text{if } y = x \end{cases}$$

then we have the analogous result:

$$\Sigma \cap \Gamma = \bigcup_{\{t_i \in S_\tau\}} \{\{x \mapsto t_i\} \circ \rho \mid \rho \in (\Sigma|_{\{x \mapsto t_i\}} \cap \Gamma|_{\{x \mapsto t_i\}})\}$$

We need to generalize this a little to account for the existence of 0 arcs, but this is the basis for our merging algorithm. The point of defining such an algorithm is, of course, that by attaching unification graphs to each of the connections in the search algorithm, we may compute the unifiers of a

component by simply merging the unifiers of the connections or components from which it was formed. This method gives a great reduction in unification time in larger searches, and since all existing search methods also use a fixed bound on the depth of unification, it does not affect the completeness of this search as compared to other methods already in use.

ALGORITHM 5.2.7 (*Merge*). *The* merge *of two unification graphs whose roots are U and V is defined as* Reduce *applied to the result of the following algorithm:*

1. *Suppose U has children $U_1, \ldots, U_k$, reached by out-arcs labelled $u_1, \ldots, u_k$, and V has children $V_1, \ldots, V_l$, reached by out-arcs labelled $v_1, \ldots, v_l$. Let $C = \emptyset$.*
2. *If $U = V$, return U. Otherwise:*
3. *For each pair (i, j) where $1 \leq i \leq k$ and $1 \leq j \leq l$:*
   (a) *If $u_i = v_j$ then let N be the merge of U_i and V_j. If N is a unification graph, add the pair (u_i, N) to C.*
   (b) *If $u_i = 0$ then let N be the merge of U_i and V_j. If N is a unification graph, add the pair (v_j, N) to C.*
   (c) *If $v_j = 0$ then let N be the merge of U_i and V_j. If N is a unification graph, add the pair (u_i, N) to C.*
4. *If C contains two entries (u, N) and (u, N'), replace C with* Collapse(C) *(defined below). (Note that since C is a set, this implies $N \neq N'$.)*
5. *If C is empty, return* FAIL.
6. *Otherwise, create a node M, labelled with the same variable as U and V. The children of M are given by the pairs of elements in C: for each $(u, N) \in C$, add an out-arc labelled u pointing to the node N. Return M.*

ALGORITHM 5.2.8 (*Collapse*). *The* collapse *of a set of pairs of arcs and nodes $C = \{(u_i, N_i) \mid 1 \leq i \leq p\}$ (as in the above definition) is defined as follows:*

1. *If all the u_i are distinct, return C.*
2. *Otherwise, there is some u such that $\{(u, N), (u, N')\} \in C$. Let D be the set of all (v, M) such that either N or N' has an out-arc labelled v pointing to a node M. Let D' be* Collapse(D).
3. *Create a node N'' labelled with the same variable as N and N', whose children are given by the pairs of elements in D', as in* Merge. *Reduce N''.*
4. *Let C' be the result of replacing (u, N) and (u, N') in C by (u, N''). Return* Collapse(C').

Before proving that this algorithm is correct, we present an example. We begin with two unification graphs, displayed in Figure 5.5; to simplify the presentation, we give names to the nodes, and label the nodes with these names rather than with the names of variables (which are irrelevant to the merging process anyway). The ACCEPT nodes are marked with an asterisk.

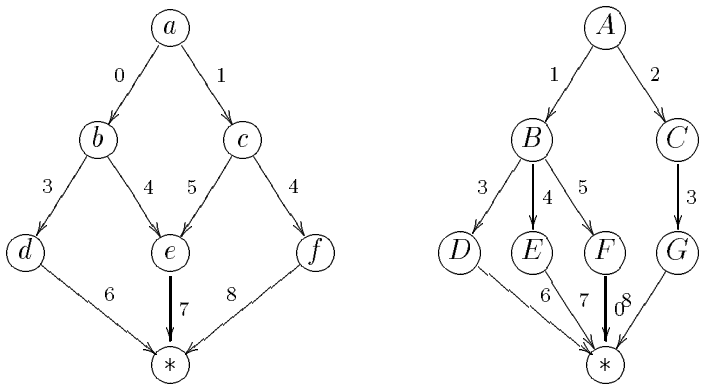


FIGURE 5.5. Two unification graphs to be merged.

In order to merge these two graphs, we call $Merge(a, A)$. The only pairs of out-arcs from these nodes which are compatible are $(0,1)$, $(1,1)$, and $(0,2)$ and so we must compute $Merge(b, B)$, $Merge(c, B)$ and $Merge(b, C)$.

First we compute $Merge(b, B)$. The only compatible pairs of out-arcs from these nodes are $(3,3)$ and $(4,4)$, and so we must compute $Merge(d, D)$ and $Merge(e, E)$. Note that d and D are in fact the same unification graph, as are e and E, hence $Merge(d, D)$ is d and $Merge(e, E)$ is e. So we may finish the computation of $Merge(b, B)$; we need not call *Collapse*, and so we try to create a new node with out-arcs 3 and 4 leading to nodes d and e. This node is in fact identical to b, and so $Merge(b, B)$ is b.

Second, we compute $Merge(c, B)$. The only compatible pairs of out-arcs from these nodes are $(4,4)$ and $(5,5)$, and so we compute $Merge(f, E)$ and $Merge(e, F)$. Nodes f and E are incompatible, so their merge is FAIL, which is not a unification graph. Nodes e and F are compatible, and $Merge(e, F)$ is clearly e. Again, we need not call *Collapse*, and $Merge(c, B)$ is a new node with a single out-arc 5 leading to the node e; we construct this node and call it g.

We must also compute $Merge(b, C)$. The only compatible pair of out-arcs is $(3,3)$, leading to the nodes d and G. Clearly $Merge(d, G)$ is FAIL, since 6 and 8 are incompatible, and so we generate no child nodes, and $Merge(b, C)$ is also FAIL.

Now we can finish the computation of $Merge(a, A)$. Since $b \neq g$, the pairwise merging of nodes has returned a set $C = \{(1, b), (1, g)\}$ and we must call *Collapse*. Now b has two out-arcs labelled 3 and 4, leading to d and e, and g has one out-arc 5 leading to e, and since all of these arcs have distinct labels *Collapse* does not need to recursively call itself, and simply constructs and returns a new node h with three out-arcs labelled 3,4 and 5, leading to d, e and e respectively. Finally, *Merge* constructs a node i with one out-arc labelled 1, leading to h. Hence $Merge(a, A)$ is the graph in

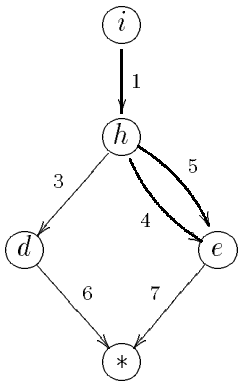


FIGURE 5.6. The result of merging the graphs in Figure 5.5.

Figure 5.6. The reader may verify that the substitutions contained in i are exactly $\{\mathrm{mrg}(\sigma,\rho) \mid \sigma \in a \wedge \rho \in A \wedge \sigma \parallel \rho\}$.

LEMMA 5.2.9. *Suppose M is a node having out-arcs labelled $u_1,\ldots,u_p$ leading to nodes $N_1,\ldots,N_p$. Let C be $\{(u_i,N_i) \mid 1 \leq i \leq p\}$, and let C' =*Collapse(C)*. Then we have the following:*

1. *If $(v,N) \in C'$ then N is a unification graph.*
2. *If M' is defined to be the unification graph whose children are given by the elements of C', as in the last step of the merging algorithm, then $\sigma \in M$ iff $\sigma \in M'$.*

PROOF. We show this by double induction, firstly on the depth of the nodes N (all the nodes in the list must be of the same depth), and secondly on the number of pairs $\{(u,N),(u,N')\} \in C$.

Suppose all the N are of depth 1. If there are no pairs to collapse, the result is immediate. Otherwise, we will pick some pair $(u,N),(u,N')$ such that both N and N' have some number of out-arcs all leading to ACCEPT. So D will be a set, containing (v,ACCEPT) for each v which is the label of an out-arc of N or N'. Then $D' = D$ immediately, so N'' contains all the substitutions of N and N' (and nothing else), and is clearly a unification graph (since D is a set). So we have reduced the number of pairs to be collapsed, and the result follows by induction.

Suppose now that the N are of greater depth. If there are no pairs to collapse, we are done. Otherwise, we will pick some pair $(u,N),(u,N')$ from C, and generate the new list D. By the induction hypothesis, we know that the substitutions represented by D' =*Collapse*(D) are exactly those represented by D, and that if $(v,M) \in D'$ then M is a unification graph. By inspection of the definition of *Collapse*, we have that D' does not contain any pairs $\{(v,M),(v,M')\}$. Hence *Reduce*(N'') is a unification graph, and clearly (u,N'') represents exactly the same substitutions as $\{(u,N),(u,N')\}$. We have collapsed one pair, and so by the induction hypothesis we are done. □

NOTATION *If* $L = (l_1, \ldots, l_n)$ *is a list, then* $l_0 \frown L$ *is the list* $(l_0, l_1, \ldots, l_n)$.

THEOREM 5.2.10. *For any two unification graphs* U *and* V*:*

1. *If* U *and* V *are incompatible then* Merge(U, V) *is* FAIL.
2. *If* U *and* V *are compatible, then* Merge(U, V) *is a unification graph, and* $\{\sigma \mid \sigma \in \text{Merge}(U, V)\} = \{\text{mrg}(\nu, \tau) \mid \nu \in U \wedge \tau \in V \wedge \nu \parallel \tau\}$

PROOF. By induction on the depth of U and V; we use the notation of the above algorithms:

A graph of depth 1 has a single root with a number of out-arcs leading to the node ACCEPT. In particular, the merge of any pair of child nodes must be the node ACCEPT, and the conclusion follows immediately since C cannot contain any pairs $\{(u, N), (u, N')\}$.

For other graphs, if U and V are incompatible then suppose for some i and j we have $u_i = v_j$ or $u_i = 0$ or $v_j = 0$. (If this is not the case, then clearly we will have C empty and return FAIL, from which the result follows.) If U_i and V_j were compatible, then by definition we could find a pair of compatible substitutions $\sigma \in U_i$ and $\rho \in V_j$, and then $u_i \frown \sigma$ would be compatible with $v_j \frown \rho$, making U compatible with V. Hence U_i and V_j cannot be compatible, and by the induction hypothesis we have *Merge*(U_i, V_j)=FAIL, which is not a unification graph. Hence C will remain empty, and *Merge*(U, V)=FAIL as required.

If U and V are compatible, suppose their root variable is x. If $\nu \in U$ and $\tau \in V$ are compatible, then write $\nu = (\nu_1, \ldots, \nu_p)$ and $\tau = (\tau_1, \ldots, \tau_p)$. By definition of compatibility, $\nu_1 = \tau_1$ or $\nu_1 = 0$ or $\tau_1 = 0$, and so we will attempt to merge the appropriate pair U_i and V_j. Since U_i contains the restriction $(\nu_2, \ldots, \nu_p)$ of ν, and V_j the corresponding restriction of τ, they are compatible, and by the induction hypothesis *Merge*(U_i, V_j) is a unification graph containing $\text{mrg}((\nu_2, \ldots, \nu_p), (\tau_2, \ldots, \tau_p))$. So a pair (u, N) will be added to C. By Lemma 5.2.9, the *Collapse* function has no effect on the set of substitutions which are represented in C, and the second elements of the pairs which are returned by the *Collapse* function are always unification graphs. It is clear that C is nonempty, and that *Collapse* cannot make C empty, and so we will reach the last step of the *Merge* algorithm and create the new node M. By the above argument, M contains all of the substitutions creating by merging elements of U and V, and cannot contain any other substitutions. All of the immediate subgraphs of M are unification graphs; by inspection of *Collapse* we see that the out-arcs of M cannot contain any duplicates. Hence M satisfies all of the requirements of a unification graph except perhaps for not containing isomorphic subtrees, and so *Reduce*(M) is a unification graph having the required properties. □

REMARK Although we have claimed that intersection is the only operation on subgraphs which we are interested in, the algorithm *Collapse* in fact computes the union N'' of two subgraphs N and N', in order to remove duplicate out-arcs with the same labels while preserving all the substitutions

of N and N'. However, the author is unable to think of any other uses for the union of two graphs.

5.2.1. Implementation. The above algorithm seems somewhat inefficient, for four reasons:

1. The same pair of nodes may be reached by many paths, and so may be merged many times.
2. We call *Reduce* after almost every call to *Merge* or *Collapse*.
3. Given two nodes, one with n out-arcs and one with m out-arcs, we must examine nm pairs of arcs in order to take the first step in merging them.
4. In order to decide whether to call *Collapse*, we must inspect a (possibly very large) list of new children; the alternative is to call *Collapse* every time, which would be even less efficient.

The first problem, which is by far the most serious, can be solved by establishing a hashtable which stores the past results of merging pairs of nodes (in fact, it need only store pointers to the main hashtable which contains all of the known subgraphs). The contents of this hashtable may or may not be preserved between top-level calls to *Merge*; this is a trade-off between speed and memory (storing old results may make things faster, but may also simply fill the memory with irrelevant data). Accordingly, we allow the user to control how often the contents of this hashtable are discarded, although the contents will always be preserved for the entire course of a top-level call to *Merge*.

The second problem can be dealt with quite simply; at the two points where the algorithm says "create a node", we assume that it first checks the hashtable of existing nodes and, if the node to be created already exists, returns the existing node. It is clear that this will allow us to omit the calls to *Reduce*, for the same reasons as in the construction of the original unification graphs — principally because this algorithm constructs nodes depth-first.

For the third problem, recall that out-arcs are assumed to be arranged in increasing order. The only way in which two arcs can be compatible is if they have identical labels, or one of them is labelled 0. If a 0-labelled arc exists, it will be the first out-arc, and so will be easily found by the algorithm. Suppose U has n out-arcs and V has m out-arcs. We may then match the out-arcs as follows:

1. If U has a 0-labelled out-arc, match it with all of the out-arcs of V.
2. If V has a 0-labelled out-arc, match it with all of the non-zero out-arcs of U.
3. Match all of the non-zero out-arcs of U with all of the non-zero out-arcs of V, using the obvious algorithm on lists (compare the first elements; if they are the same then merge them, otherwise discard the smaller one and repeat).

This completes the task in at most $n + m + \max(n, m)$ recursive calls to *Merge*. In the case where neither U nor V has both 0-labelled and non-0-labelled out-arcs (which is overwhelmingly more common than the other case), it will take only $\max(n, m)$ calls to *Merge*.

Finally, we have the calls to *Collapse*. Note that the condition which requires a call to *Collapse* can only be satisfied if we have generated two out-arcs with the same label, and this can only happen if either U or V has both 0 and non-0 out-arcs. Again, this case is quite rare, and can easily be tested for, and so in the majority of cases we may ignore the check for calling *Collapse*.

5.3. Extensions of Unification Graphs

There is one further extension of this idea which will prove useful in some circumstances, which is to allow a single node of the graph to represent several variables, and to label its out-arcs with lists of substitutions. This will have two major advantages over the previous description: it will save space in the case where large parts of the jform are redundant (for example, if we have made many duplications but will only use a few of them), and in some circumstances it will provide a straightforward way of reusing unification graphs after expanding the jform (see Section 5.5).

We first begin by redefining the concept of a variable ordering; instead of a list of variables such that the nth arc on a path is labelled with a substitution for the nth element of the list, we require a list of *lists* of variables, such that the nth arc is labelled with a *list* of substitutions for the nth element of the list.

DEFINITION 5.3.1. *A* variable ordering *for a jform J is a list of lists of variables satisfying the following requirements:*

1. *Each element of the list is itself a list, whose elements are free variables of J.*
2. *Each free variable in J occurs in exactly one of elements of the list, and occurs there exactly once.*

EXAMPLE 5.3.2. *Suppose that the free variables of J are $\{x, x', x'', y, y', p\}$. Then the following are all valid variable orderings:*

1. $((x), (x'), (x''), (y), (y'), (p))$
2. $((x, x', x''), (y, y'), (p))$
3. $((p, y), (y'), (x'', x), (x'))$

We may now construct decision trees for substitutions as before, except that now the nodes at depth n will be labelled with the nth list from our variable ordering, and their out-arcs will be labelled with lists of integers representing substitutions. The algorithms *Prune* and *Reduce* have obvious generalizations to this case, and we may introduce the "substitution" 0 exactly as before (now, of course, it will occur as an element in the list which labels an arc). We may recover essentially the same unification graphs as in

the previous sections, by choosing a variable ordering in which each interior list has length 1 (as in Example 5.3.2(1)), and so this really is a generalization of the previous definition. It seems unnecessary to repeat the entire construction again, and so we jump immediately to the algorithm for constructing unification graphs with the new labelling:

ALGORITHM 5.3.3. *To construct a unification graph for the substitutions Σ over the variable ordering $l_1, \ldots, l_k$, where $l_i = (x_i^1, \ldots, x_i^{n_i})$ is a list of variables for each i:*

1. *Let $\tau_1, \ldots, \tau_{n_1}$ be the types of $x_1^1, \ldots, x_1^{n_1}$.*
2. *Let $\Sigma = \{(m_1, \ldots, m_{n_1}) \in \mathbb{N}^{n_1} \mid \forall i (1 \leq i \leq n_1) \Rightarrow 0 \leq m_i \leq n_{\tau_{n_1}}\}$ enumerate all possible lists of substitution terms for these variables, including zeros for variables which do not occur in a substitution.*
3. *For all $i \in \Sigma$, let Σ_i be $\{\sigma \in \Sigma \mid (\sigma(x_1^1), \ldots, \sigma(x_1^{n_1})) = i\}$.*
4. *If $k \neq 1$, then for all $1 \leq i \leq n_\tau$ such that $\Sigma_i \neq \emptyset$, construct the unification graph U_i for the substitutions Σ_i over $x_2, \ldots, x_k$.*
5. *If $k = 1$, then for all $1 \leq i \leq n_\tau$ such that $\Sigma_i \neq \emptyset$, let U_i be the node* ACCEPT *(retrieved from the hashtable — i.e. the same node in all cases).*
6. *If the hashtable contains a nonterminal node N whose out-arcs are labelled with $\{i \mid \Sigma_i \neq \emptyset\}$ and in which each arc labelled i leads to the subgraph U_i, then return that N, otherwise construct such an N, insert it into the hashtable and then return N.*

Exactly the same implementation considerations apply to this algorithm as to Algorithm 5.1.10. The definitions of compatibility and merging are also exactly analogous to the previous versions, and are omitted.

EXAMPLE 5.3.4. *Figure 5.7 shows a unification graph with this new labelling. Suppose that the variable ordering is $((p, p'), (x, x'), (y))$; then the substitutions represented by this graph are*

1. $\{p \mapsto 1, p' \mapsto 2, x \mapsto 3, y \mapsto 6\}$
2. $\{p \mapsto 1, p' \mapsto 2, x \mapsto 4, x' \mapsto 4, y \mapsto 7\}$
3. $\{p \mapsto 1, p' \mapsto 2, x \mapsto 5, x' \mapsto 4, y \mapsto 7\}$

5.4. Choosing a Variable Ordering

The choice of variable ordering for an ordered binary decision diagram critically affects the size of the resulting graph. It is reasonable to expect that, despite the differences between unification graphs and OBDDs, the choice of variable ordering will also affect the efficiency of our unification graph algorithms.

The standard choice of variable ordering for our implementation of component search is to partition the variables into sets such that x and y occur in the same set iff x is a duplicate copy of y or y is a duplicate copy of x. (Example 5.3.2(2) shows such an ordering, if we assume that a variable v' is only ever produced by duplicating a variable v.) These sets will form the

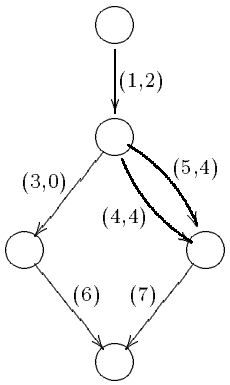


FIGURE 5.7. A Unification Graph with List-Labelled Arcs.

interior lists of our ordering, and the main list will be produced by ordering the interior lists so that L_1 occurs before L_2 if, on a depth-first traversal of the jform, some element of L_1 occurs before all elements of L_2.

This ordering is efficient in most cases, probably because it satisfies the same general idea as is satisfied by most good variable orderings for OBDDs — namely, that variables which are close to each other in the formula should also be close to each other in the ordering. However, other variable orderings are certainly possible, and we discuss some of them in Section 8.8.

5.5. Duplication

Suppose that the interior lists of our variable ordering have been chosen as described above (i.e. that each list contains exactly the original copy of a variable and all of its duplicates). Suppose also that our jform uses Skolem terms rather than selection parameters.

If a quantifier is duplicated during the search (note that, in our algorithm, this will happen very rarely if at all; duplication normally occurs before the search starts), the unification graphs already in existence can easily be reused. We merely add the newly-generated variables to the appropriate elements of the variable ordering, and then relabel the arcs of all the existing graphs by adding an extra 0 in the appropriate position in the substitutions for those elements.

Furthermore, we can easily compute the unification graphs for most of the new potential connections by generating a relabelled copy of an existing unification graph. Specifically, if $C_1 = (L_1, L_2)$ is a known connection, and $C_1' = (L_1', L_2')$ is a new connection in which L_1' and L_2' have been obtained by quantifier duplication from L_1 and L_2, then the ground unifiers of C_1' can be obtained by renaming variables in the ground unifiers of C_1. In each element $x \mapsto x'$ of this variable renaming, the variables x and x' will be duplicate copies of each other, and hence will occur within the same element of the variable ordering. Because of this, the new unification graph can be

obtained by merely permuting some of the lists which label the arcs of the old graph in the obvious way.

This will not solve all of the new unification problems (since, for example, (L_1, L_1') may be a valid new connection), but it will significantly reduce the amount of work required to accommodate a quantifier duplication.

Note that the above assumes that the only new symbols introduced by a quantifier duplication are new free variables. This is true if Skolemization is being used, but may not be true if selection nodes [Miller, 1983] are used instead; in this case, new constants (and hence new potential unifiers even for old connections) may also be introduced. and the presence of Miller's "acyclicity check" means that the relabelling of old unifiers may no longer produce valid new unifiers. We discuss Skolemization and selection nodes, and introduce methods for pruning unification graphs when selection nodes are used, in Section 8.6.

CHAPTER 6

Duplication and Set-Variable Substitution

6.1. Duplication

Previous search procedures in TPS have had two possible approaches to duplication: older procedures waited until all matings had been tried and then duplicated all of the outermost quantifiers, while the newer procedures used path-focused duplication [Issar, 1991] in which the quantifiers on a particular vertical path would be duplicated if there was no other connection available on that path. The former is clearly not very efficient, especially for larger jforms in which it might take some time to try all possible matings. The latter, although more effective in many cases, is not useful for searches in which more than a couple of duplications are needed, or in which the vertical paths contain many quantifiers, because having added a connection to a partial mating, it cannot backtrack over it until all possible extensions of that mating have been explored (and there will be an enormous number of such extensions if we may duplicate many quantifiers on each path).

We will attempt to preserve the "duplication by need" aspects of path-focused duplication, but to implement them in the context of a component-based search which, being essentially breadth-first, may not be as vulnerable to combinatorial explosion as was path-focused duplication.

Recall that one of the basic strategies which inform the design of this search procedure is that decisions about the search space should be separated from decisions about the search itself. This is done partly to increase the modularity of the search (and hence to allow for future changes to be made more easily), and partly to ensure that saved information will remain relevant throughout the search. In preceding chapters, we have seen that even duplicating a single quantifier can require us to revisit all of the unification graphs, and to return to the level of individual connections and begin to rebuild the components for a new search. Accordingly, we will attempt to perform all of the duplications that may be necessary (i.e. that fall within the bounds set by the user) before we begin to search for a mating. It may appear that, as a strategy for duplication, this is no better than the worst of the former strategies, but in the context of a component-based search we can control the way in which the new duplications are allowed to appear in a mating, and hence recover most if not all of the benefits of duplication by need.

There are two problems with expanding the jform before the search begins. One problem is that this will increase the size of the expansion tree,

and also of the jform, and will require the enumeration of more possible connections and more unification graphs. If new variable types or new constants are introduced, it may also increase the number of possible substitutions that must be enumerated. This problem is minor: since these numbers and sizes are small in comparison to (for example) the number of components that will be enumerated, the extra demands on memory will be small in comparison to the demands made by the main part of the search. The extra time required to enumerate the new connections and unifiers is harder to quantify, but again seems not to be a major cause of inefficiency — possibly because many of the same connections and unifiers would eventually be produced by any other search method, perhaps many times over.

The more pressing problem is the explosion in the size of the search space. Doubling the number of literals will increase the number of connections from at most $\binom{n}{2}$ to at most $\binom{2n}{2}$, but will increase the number of partial matings by vastly more. Accordingly, once we finish enumerating connections and begin to enumerate minimal components and then larger components, we run the risk of a combinatorial explosion.

Fortunately, we can evade this risk, at least partially, by distinguishing the original copy of a fragment from the first duplication, and that from the second duplication, and so on, and by requiring (essentially) that a component must touch the first k copies before it is allowed to touch the $(k+1)$st copy. In this way, we recover the idea of "duplication by need". We will also allow the user to set bounds for the search that restrict the total number of duplications that may be touched by a component, by setting the flag MS98-NUM-OF-DUPS (see Appendix A).

In contrast to some of the path-enumeration procedures (in particular, path-focused duplication [Issar, 1991]), we will only ever duplicate the outermost quantifiers of the jform. This may produce a larger search space than necessary, which is why path-focused duplication attempts to duplicate innermost quantifiers first, but in the context of component search it is justifiable for several reasons. Firstly, since we do not amplify the jform during the search, and we have no a priori way to determine which quantifiers must be duplicated, we must duplicate the outermost quantifiers in order to guarantee completeness. It would be possible to duplicate *all* quantifiers, but this would of course make the problem of combinatorial explosion even worse. Secondly, our method for eliminating redundancy in the search, which we present below, only works if the jforms below two copies of the same quantifier are identical up to renaming of bound variables; this will not be the case if we can duplicate both an outermost quantifier and some other quantifier which lies below it in the jform. A more complex method of identifying redundancy could probably be devised, but at some cost to the efficiency of the search.

So we begin the search by making a fixed number of duplications of every outermost quantifier (this number is given by the flag NUM-OF-DUPS; see Appendix A). During the enumeration of individual connections, we use

the methods from the previous chapter to construct the unification graphs of any new connection which is a renaming of an existing connection. If MS98-NUM-OF-DUPS is set, we also reject any connection that would violate this bound. (For example, if the total number of duplications is limited to four, then we cannot possibly mate two literals which each occur in the third copy of their respective fragments, since in order to use this connection we would have had to touch the first two copies of each fragment, and hence any mating involving this connection would use at least six duplications.)

When components or connections are combined, we check the number of duplications used by the result (comparing it to the flag MS98-NUM-OF-DUPS), discarding the result immediately if it uses too many duplications.

We now discuss the requirement that earlier duplications of a fragment must be touched before later duplications. Taken literally, this will add an unnecessary incompleteness to the search, since there may be several fragments below each duplicated quantifier, and mating one of them may prevent our mating any of the others. For example, consider the theorem $\forall x_\iota[[A_{o\iota}x \vee C_{o\iota}x] \wedge B_{o\iota}x] \supset [AY_\iota \vee CY] \wedge BZ_\iota$

When translated to a jform, this produces the following:

$$\forall x_\iota^0 \left[\begin{bmatrix} A_{o\iota}x_\iota^0 \vee C_{o\iota}x_\iota^0 \\ B_{o\iota}x_\iota^0 \end{bmatrix}\right] \left[\begin{matrix} \sim A_{o\iota}Y_\iota \\ \sim C_{o\iota}Y_\iota \end{matrix} \vee \sim B_{o\iota}Z_\iota\right]$$

There is no complete mating for this jform, since x^0 cannot be unified with both Y and Z. After one duplication, we get

$$\forall x_\iota^0 \left[\begin{bmatrix} A_{o\iota}x_\iota^0 \vee C_{o\iota}x_\iota^0 \\ B_{o\iota}x_\iota^0 \end{bmatrix}\right] \forall x_\iota^1 \left[\begin{bmatrix} A_{o\iota}x_\iota^1 \vee C_{o\iota}x_\iota^1 \\ B_{o\iota}x_\iota^1 \end{bmatrix}\right] \left[\begin{matrix} \sim A_{o\iota}Y_\iota \\ \sim C_{o\iota}Y_\iota \end{matrix} \vee \sim B_{o\iota}Z_\iota\right]$$

for which a complete mating can easily be found. However, if we begin by mating the fragment containing $A_{o\iota}x_\iota^0$ and $C_{o\iota}x_\iota^0$ to the two fragments containing $\sim A_{o\iota}Y_\iota$ and $\sim C_{o\iota}Y_\iota$, our next step (which will complete the mating) will have to use the fragment containing $B_{o\iota}x_\iota^1$, which is of course a duplicate copy of the fragment containing $B_{o\iota}x_\iota^0$, which is currently untouched. Under the constraint given above, this is disallowed, and hence we would fail to find the mating.

Clearly, then, we must use the somewhat weaker constraint that a duplicate fragment created by duplicating some variable x may only be touched

if, for every earlier duplicate copy of x, some fragment below that earlier duplicated quantifier has been touched.

DEFINITION 6.1.1. *Let $\simeq$ be the equivalence relation on the outermost universal nodes of the jform J which is defined by taking the reflexive, transitive and symmetric closure of the relation "is a duplicate copy of".*

DEFINITION 6.1.2. *Let ρ be a permutation of the outermost universal nodes of J. We say that ρ is a* relabelling *of J if for every outermost universal node $q \in J$ we have $\rho(q) \simeq q$.*

REMARK In the case that only outermost quantifiers have been duplicated, a relabelling ρ induces a permutation σ on the nodes of the jform J (since the structure of the jforms below q and $\rho(q)$ are identical). By abuse of notation, we refer to this permutation as a relabelling. This permutation obviously preserves the types of nodes, so that (in particular) we have L is a literal iff $\sigma(L)$ is a literal.

We assume for the remainder of this chapter that only outermost quantifiers have been duplicated, and so the above remark holds. Suppose further that we have chosen to use Skolemization rather than selection nodes, and so duplication of a quantified formula $\forall q.C(q)$ produces a variable-renaming of the original formula, viz. $\forall q'.C(q')$. (If selection nodes are used, we may be required to rename some constants in C as well; see [Issar, 1991] for details.)

We also suppose, without loss of generality, that no variable occurs both bound and free in J, and no veriable is bound twice in J.

LEMMA 6.1.3. *A relabelling σ of J induces a variable renaming τ_σ of the universally quantified variables of J, such that if the literal L represents a formula Φ, then the literal $\sigma(L)$ represents a formula $\tau_\sigma(\Phi)$.*

PROOF. Obvious; for any universally-quantified variable $x \in J$, suppose x is the kth variable quantified by some universal node $q \in J$. Let $\tau_\sigma(x)$ be the kth variable quantified by $\sigma(q)$; by previous remarks, τ_σ is well-defined and is a permutation of the universally-quantified variables of J. Then τ_σ clearly has the required property. □

DEFINITION 6.1.4. *We will say that two matings M_1 and M_2 are* equivalent *if there is some relabelling σ such that $\sigma(M_1) = M_2$.*

LEMMA 6.1.5. *If M_1 and M_2 are equivalent by some relabelling σ then:*

1. *M_2 and M_1 are equivalent by a relabelling σ^{-1}.*
2. *ν is a unifier of M_1 iff $\tau_\sigma \circ \nu \circ \tau_\sigma^{-1}$ is a unifier of M_2.*
3. *M_1 is complete iff M_2 is complete.*

PROOF. (1) is obvious, and σ^{-1} is well-defined since σ is a permutation.

(2): Suppose ν is a unifier of M_1. Then, by Lemma 6.1.3, we have $M_2 = \{(\tau_\sigma(L_1), \tau_\sigma(L_2)) \mid (L_1, L_2) \in M_1\}$. Hence $\nu \circ \tau_s igma^{-1}$ is a unifier of M_2, and applying τ_σ to the result will merely return any remaining free variables which were renamed by τ_σ^{-1} to their original values. The converse follows from part (1), using the trivial observation that $\tau_{(\sigma^{-1})} = (\tau_\sigma)^{-1}$.

$$\left[\begin{array}{c} \forall x_\iota^0 \\ \left[\begin{array}{c}\boxed{4}\ A_{o\iota}x_\iota^0\\ \boxed{5}\ B_{o\iota}x_\iota^0\end{array}\right] \\ \forall x_\iota^1 \\ \left[\begin{array}{c}\boxed{4.1}\ A_{o\iota}x_\iota^1\\ \boxed{5.1}\ B_{o\iota}x_\iota^1\end{array}\right] \\ \\ \boxed{6}\ C_{oi}X_i \\ \left[\boxed{8}\ \sim A_{o\iota}X_\iota \vee \boxed{9}\ \sim B_{o\iota}X_\iota \vee \boxed{7}\ \sim C_{o\iota}X_\iota\right] \end{array}\right]$$

FIGURE 6.1. A jform which has two pairs of equivalent complete matings

(3): If P were an open path through M_1, then $\sigma(P)$ would clearly be an open path through M_2; the converse is analogous. □

Next, we fix some ordering on the copies of each quantifier, by numbering them from 0th (the original copy) to kth.

DEFINITION 6.1.6. *A component C is said to be* elementary *if, whenever C touches some fragment F_k, if F_k lies below the nth duplicate copy of some variable x (for $n > 0$), then C also touches some fragment below each of the 0th, ...,$(n-1)$st copies of x.*

This amounts to saying that a component is elementary iff it touches an initial segment of the copies of each quantifier.

LEMMA 6.1.7. *Every component is equivalent to an elementary component. Furthermore, if C is elementary and $C \cup D$ is not, then $C \cup D$ is equivalent to an elementary component by some relabelling which fixes C.*

PROOF. Obvious; if C is not elementary, the required relabelling need only move those literals which cause the component to be non-elementary. Specifically, if C uses the 0th through ith copies of some quantifier, and then k other copies not including the $(i+1)$st, then relabel these k copies to become the $(i+1)$st through $(i+k)$th copies. □

EXAMPLE 6.1.8. *Let J be the jform of Figure 6.1. J has four complete matings:*

1. $\{\boxed{7-6}, \boxed{4-8}, \boxed{5-9}\}$
2. $\{\boxed{7-6}, \boxed{4.1-8}, \boxed{5-9}\}$
3. $\{\boxed{7-6}, \boxed{4-8}, \boxed{5.1-9}\}$
4. $\{\boxed{7-6}, \boxed{4.1-8}, \boxed{5.1-9}\}$

Matings (1), (2) and (3) are elementary; mating (4) is not. Matings (1) and (4) are equivalent, as are matings (2) and (3) — in both cases via the relabelling which exchanges $\forall x_i^0$ and $\forall x_i^1$ (which is the only non-identity relabelling possible in this jform). It is clear that there are only two essentially

distinct matings here: the mates for $\boxed{8}$ *and* $\boxed{9}$ *can be either below the same quantifier or below different quantifiers. In fact, if we had taken* k *duplicate copies of the quantifier* $\forall x_i^0$*, producing literals* $\boxed{4.1}$*, ...,*$\boxed{4.k}$ *and* $\boxed{5.1}$*, ...,*$\boxed{5.k}$*, then we would have* $(k+1)^2$ *possible complete matings, each of which would be equivalent to either mating (1) or mating (2) above.*

The example above strongly suggests that eliminating equivalent matings from the algorithm is a good idea, particularly when the number of duplications k is large.

THEOREM 6.1.9. *Suppose the algorithms presented in Chapter 4 are restricted so that, when components are being combined, only elementary components may be created. Then the algorithms are still sound and complete.*

PROOF. Soundness is trivial; a complete mating is still a complete mating. Intuitively, completeness is also somewhat trivial: the only difference between the ith and jth copies of a fragment are the renamings which were chosen for x and all of the variables below it. If no fragments below the ith copy have been touched, then no substitutions have been specified for any of these variables, and we may replace all the connections touching a fragment below the jth copy with the equivalent connections to the fragment below the ith copy, and this will produce an equivalent mating; hence, we may always assume that our connections touch an initial segment of the available duplicates. (This amounts, more or less, to the assertion that multiple copies of the same hypothesis may be used in any order, since they are all equivalent.)

We now sketch a more formal proof.

Let $C_0, C_1, \ldots, C_n$ be a chain of components created by our algorithm. C_0 is minimal, and C_n is a complete mating. We create a new chain of components satisfying the condition, by induction on the first point at which the component is non-elementary. This chain will end in a complete mating by Lemma 6.1.5. Our inductive hypothesis is that if the first non-elementary component is C_i, then there is an equivalent chain in which the first $(i+1)$ components are elementary. Since the chain is finite, this will prove the theorem, and we may thus restrict our algorithm to consider only chains of elementary components.

Suppose, then, that C_0 is not elementary. By Lemma 6.1.7, there is a relabelling σ such that $\sigma(C_0)$ is elementary. Since, by our algorithm, C_0 was chosen as a minimal component for some original (non-duplicate) fragment F, we may assume that $\sigma(F) = F$. Then $\sigma(C_0)$ is a minimal component blocking F, and $\sigma(C_0), \sigma(C_1), \ldots, \sigma(C_n)$ is an equivalent chain which does not violate the condition at the first step.

Now suppose that $C_0, \ldots, C_i$ are elementary but C_{i+1} is not. Then by Lemma 6.1.7 there is a relabelling σ such that $\sigma(C_{i+1})$ is elementary, and $\sigma(C_i) = C_i$. In particular, if C_i touches but does not block some fragment F, then F remains fixed under σ, and so if C_{i+1} was produced by merging C_i with some minimal component D for F, then $\sigma(C_{i+1})$ can be

produced by merging C_i with $\sigma(D)$, which is also minimal for F. Hence $\sigma(C_0), \sigma(C_1), \ldots, \sigma(C_n)$ is an equivalent chain (note that $\sigma(C_j) = C_j$ for all $j \leq i$) which is elementary at least up to and including step $(i+1)$, and by induction we are done. □

THEOREM 6.1.10. *Suppose the algorithms presented in Chapter 4 are restricted so that when components are being combined, a component C will not be created if an equivalent component D is already known. Then the algorithms are still sound and complete.*

PROOF. Obvious. It is clear that C can be extended to a minimal complete mating iff D can be extended to a minimal complete mating, by Lemma 6.1.5. □

The result of Theorem 6.1.9 is easily implemented, since checking for non-elementary components is straightforward. This also partly implements Theorem 6.1.10, in the case where C is elementary and D is not. However, it is possible for two elementary components to be equivalent, and this is a much harder case to detect; we do, however, check for cases of the form $C = B \cup F_1$, $D = B \cup F_2$, where B is some component and F_1, F_2 are minimal components for some fragment (these are easily found since the relabelling must fix B). We discuss this further in Section 8.5.

EXAMPLE 6.1.11. *Returning to Example 6.1.8, we observe that Theorem 6.1.9 will eliminate mating (4) from consideration, while preserving mating (1). If matings (2) and (3) are considered as minimal components for the fragment containing* 7, 8 *and* 9, *then our partial implementation of Theorem 6.1.10 described in the previous paragraph will eliminate one of these matings and preserve the other.*

REMARK Although we have developed this pruning algorithm under the assumption of Skolemization rather than selection nodes, Theorem 6.1.9 and Theorem 6.1.10 are equally applicable to the case in which selection nodes are used. This is clear, since the choice of selection nodes or Skolemization does not affect the arrangement of the literals in the jform, nor the unifiability of a partial mating. See [Miller, 1983] for details.

6.2. Substitutions for Set Variables

We handle substitutions for set variables in much the same way as we handle duplications. All of the substitutions that are to be tried are applied simultaneously at the beginning of the search, and to avoid a combinatorial explosion, the number of set-variable substitutions that can be touched by any particular component is restricted, usually to one or two.

In theory, this approach is beneficial in that it doesn't require us to spend a fixed amount of time considering each potential but incorrect substitution before we reach the correct one. In practice, there may be an extremely large number of such substitutions, even within the bounds set by the user. Since

each substitution introduces new variables (often with types not previously the type of any free variable), and the substitutions may also generate new constants (see Section 8.6 for an explanation of this and a discussion of how we can prune the unification graph in this case), this may increase the space of possible unifiers to the point where it becomes intractable. Therefore we may have to accept a compromise in which substitutions are considered in batches (rather than individually) for a fixed time before the search is restarted with a new batch of substitutions. Both of these possibilities are implemented in TPS, and are controlled by the flag MS98-INIT.

The actual methods used for generating and applying substitutions for set variables in component search are exactly those used by the other searches in TPS, and are described in [Bishop and Andrews, 1998; Andrews *et al.*, 1996]. Alternative methods are presented in [Benzmüller, 1998; Bledsoe and Feng, 1993; Bailin and Barker-Plummer, 1993], but it is not clear how to integrate these methods into mating search.

CHAPTER 7

Equality Reasoning

Since the unification of individual connections is completed before the mating search begins, we can now define a convenient way to unify modulo (some bounded number of applications of) an arbitrary set of equalities. This helps us to avoid the difficulties of instantiating Leibniz variables (the q's in $\forall q.qA \supset qB$, which generally require large substitutions corresponding to a context into which A and B are substituted). Considering such Leibniz equalities as unification constraints for particular connections can also dramatically reduce both the size of the search space and the number of duplications required, as we shall now explain.

7.1. Global and Local Equalities

DEFINITION 7.1.1. *Given a jform J, we will classify all of the equalities which occur positively in J as either* global *or* local. *Suppose $A = B$ is an equality which occurs positively in J as $\forall q.[\sim qA] \vee qB$. If every vertical path through J passes through either the literal $\sim qA$ or the literal qB, we say that $A = B$ is* global. *If $A = B$ is not global, but every vertical path through a literal L passes through either the literal $\sim qA$ or the literal qB, we say that the equality is* local to L.

When we are performing a search with unification modulo equalities, we will remove all of the global equalities from the jform and restrict the search to whatever literals remain. We can do this because of the way in which the unification routine is applied, as we now explain.

7.2. Unification with Equalities

Our method of unification with equalities is extremely simple-minded: given a connection (L_1, L_2), compute the corresponding disagreement pair $\langle s, t \rangle$ in the normal way, for each ground unification substitution σ take the two ground terms σs and σt and attempt to find a common rewrite using (first-order or ground instances of) the global equalities and the equalities which are local to both L_1 and L_2. If a common rewrite can be found within the search bounds that have been given, we will accept (L_1, L_2) as a valid connection, and σ as a substitution which unifies it. In fact, we will implicitly have a chain of connections which begins at L_1 and passes through some number of equalities before eventually reaching L_2, but, by our definitions of global and local equalities, this chain of connections will span every path

through the pair L_1, L_2. When the mating is complete and we attempt to translate it to a natural deduction proof, this chain of connections will have to be made explicit in order for the translation to work, but for the purposes of finding a complete mating we may regard it as simply a single connection.

DEFINITION 7.2.1. *A* rewrite tree *for a term t is a tree in which each node is labelled with a wff. The root is labelled with t, and in the children of a node labelled s are nodes representing the collection of all the one-step rewrites of s. The arcs from a node s to a node t are labelled with the rewrite rule r_i that is used in rewriting s to t.*

DEFINITION 7.2.2. *A* discrimination tree *[McCune, 1992] is a tree in which each node is labelled with a symbol, and any path from the root to a leaf represents the sequence of symbols in the printed representation of a term.*

REMARK We use the printed representation of a term as produced by TPS, including parentheses, rather than an enumeration of its parse tree, in order to distinguish between higher-order terms that differ only in the arrangement of their parentheses (otherwise, for example, the path $\lambda - x_\alpha - y_{o\alpha} - z_\alpha$ could represent either $\lambda x_\alpha [y_{o\alpha} z_\alpha]$ or $[\lambda x_\alpha . y_{o\alpha}] z_\alpha$). Terms are converted to $\alpha\beta\eta$-normal form before being inserted into the tree; we α-normalize by choosing some new sequence of variables $v_1, v_2, \ldots$ and replacing the first bound variable on the path with v_1, the second with v_2, and so on.

REMARK Discrimination trees are only one of a large class of approaches to term-indexing which have been widely used in theorem-proving, particularly in the first-order case. See [Graf, 1995] for a survey of term-indexing methods. In our case, we only need to insert and retrieve single terms, and so we choose discrimination trees because they are simple and deterministic, and the operation of inserting a new term is particularly efficient. If operations such as deletion or retrieving sets of unifiable terms were required, then some other method (for example, substitution-tree indexing) would be called for.

Given two terms s and t, we will simultaneously construct a rewrite tree and a discrimination tree for each. The discrimination tree will contain all of the terms in the rewrite tree, and each leaf will be labelled with the node in the rewrite tree to which it corresponds. Given a new term to be added to the rewrite tree for s (resp. t), we first inspect the discrimination tree for s (resp. t), and discard the term if it is already present in the tree. Then we inspect the discrimination tree for t (resp. s); if the new term is present in this tree, then we have successfully rewritten s to t (recall that all of our rewrites are bidirectional), and we may extract from the two rewrite trees the list of rewrites required to do this.

Ordinarily, we will in fact be attempting to make s and t contradictory. If there is no rewrite rule which can remove or introduce a top-level negation (as is normally the case), then we may simply fail if both terms are negated,

or if neither is, and otherwise attempt to rewrite $\sim s$ to t, after removing double negations. If there is such a rewrite rule, then we generate rewrite trees for s and t, and remember to insert a negation when we are inspecting the tree for successful rewrites.

We now state the procedure for the search with equalities.

7.3. Search Algorithm with Equalities

ALGORITHM 7.3.1. *The equality procedure works as follows:*

1. *Before the mating search begins, the jform is searched for positive instances of Leibniz equalities. For each of these equalities, a rewrite rule is defined, and this is marked as* global *or* local *according to the definition above. The literals corresponding to global equalities are marked as dead for the duration of the mating search.*
2. *If any rewrite rule is higher-order (i.e. contains variables of function type), we reduce it to a set of first-order (or possibly ground) rewrites by applying all available substitutions for its higher-order variables. For convenience, the resulting rewrites are labelled with the substitutions which produced them and the name of the original rewrite rule.*
3. *The mating search then begins as before, with a change in the step where connections are computed: a single connection cannot fail during unification. Instead, we generate all possible pairs $\langle s,t\rangle$ of ground instances of those literals and compute the set of* active rewrite rules, *which contains all of the global equalities plus the local equalities of this connection (or, if there are higher-order rewrite rules, the appropriate substitution instances of these equalities).*
4. *For each pair of terms $\langle s,t\rangle$ and set of active rewrite rules, we will be given a bound on the length of any chain of rewrites, and a unification-style bound on the size of any term that can be substituted for a new variable (for example, the group theory rule $x.x^{-1} = e$, if applied from right to left, will introduce the new variable x, which must be substituted for). We search for a rewrite proof of $s = t$, by generating a rewrite tree and a discrimination tree for each of s and t, as above. If rewriting succeeds, we store the chain of rewrites in a hashtable, indexed by the pair of literals and the substitution, and then treat the connection exactly as if it were unified by that substitution. If it fails, the connection is not unifiable and is ignored.*
5. *The search then continues, using the connections and their unification graphs to build up minimal components and then to combine these, exactly as in Algorithm 4.1.1.*

Note that once the unification graphs have been computed for individual connections, Algorithm 4.1.1 does all subsequent unification by applying *Merge* to unification graphs. In particular, the algorithm never verifies that the substitutions represented by the graphs really are unifiers of the connections and components to which they are associated. This is why we can treat

a "connection" which is justified by a rewrite proof in exactly the same way as a normal connection.

7.4. Remarks on the Unification Method

As remarked above, attempting to find a common rewrite of s and t by simply generating all possible rewrites is a somewhat simple-minded approach, as is our method of dealing with higher-order rewrites by making ground substitutions for the higher-order variables until they become first-order.

In the remainder of this section, we consider some possible alternative methods for searching for a proof of $s = t$ modulo an arbitrary set of equalities (equivalently, a set of bidirectional rewrite rules). Both rewriting and equality reasoning have been the subject of considerable research, and there is extensive literature on both; the following discussion is in no way comprehensive.

Special Unification Methods. Many methods have been developed for equational unification in particular theories. Surveys of results of this type can be found in [Kryvyi, 1998; Jouannaud and Kirchner, 1991; Siekmann, 1989; Siekmann, 1984]. In a similar vein, the method of theory resolution [Stickel, 1985] allows the incorporation of decision procedures for particular theories into a general resolution framework. Since we require a method which is applicable to an arbitrary set of equations, we do not discuss this further.

Completion, Paramodulation and Narrowing. Given a set of equations E, a completion strategy attempts to construct an equivalent set R of rewrite rules by orienting equations according to some reduction ordering $\prec$ and by analysing the critical pairs of the equations on E. An algorithm for doing this is presented in [Knuth and Bendix, 1970]; the objective is to produce a convergent (i.e. confluent and terminating) set R of rewrite rules. An introduction to completion (and to rewriting in general), with the definitions of all of the undefined terms above, is provided in [Dershowitz, 1993]. A fair strategy will either fail, converge on a decision procedure for E, or generate an infinite set of rewrite rules R [Huet, 1981]. There are unfailing completion methods [Bachmair *et al.*, 1986; Bachmair *et al.*, 1987] which eliminate the possibility of failure but not of non-termination. Non-termination can be handled, for certain classes of equations, by special methods such as associative commutative completion [Bachmair, 1991].

Paramodulation [Robinson and Wos, 1969] is an inference rule for equations; a calculus for higher-order paramodulation is given in [Benzmüller, 1998]; the same paper also extends the equality reasoning of [Digricoli, 1979] to the higher-order case. Narrowing [Slagle, 1974; Lankford, 1975] is similar to paramodulation, but uses rewrite rules instead of equations. An overview of first-order narrowing is given in [Middeldorp and Hamoen,

1994]; the higher-order case is discussed in [Prehofer, 1997]. The first E-unification algorithm using narrowing was presented in [Fay, 1979]; approaches to higher-order E-unification are discussed in [Snyder, 1990; Snyder, 1991; Nipkow and Qian, 1991; Nipkow, 1993; Müller and Weber, 1994]. In higher-order E-unification, extensionality is a particular problem; a method for dealing with extensional unification is presented in [Benzmüller, 1998; Benzmüller and Kohlhase, 1998].

Because these methods are not guaranteed to terminate on arbitrary sets of equalities, it would seem that they cannot easily be used by our approach unless they are integrated with the search procedure itself, rather than being a part of the preprocessing on individual connections.

Equational Matings and Rigid E-Unification. There are extensions of the mating search method which are designed to deal directly with equality reasoning. Two such methods (actually, one method with two sets of terminology) are presented in [Bibel, 1987], in which complementary literals are replaced by eq-connections instantiated by eq-unifiers, and [Gallier *et al.*, 1990], in which complementary literals are replaced by mated sets and rigid E-unifiers. Unfortunately, the problem of finding a mated set with a rigid E-unifier is difficult, even in the first-order case. A path containing m equations and n negated equations will have $n.2^m$ mated sets, and the check for rigid E-unifiability of a mated set is NP-complete [Gallier *et al.*, 1992]. Finally, discovering the substitution for the whole matrix requires the solution of the *simultaneous* rigid E-unification problem, which is now known to be undecidable in general even for the first-order case [Degtyarev and Voronkov, 1995b], although a number of special cases are known to be decidable [Voronkov, 1998; Degtyarev *et al.*, 1997; Gurevich and Voronkov, 1997].

A more tractable method for mating search with equality is proposed in [Degtyarev and Voronkov, 1997; Degtyarev and Voronkov, 1995a]. This method, known as equality elimination, separates equality reasoning and extension (i.e. quantifier duplication) procedures. The procedure of [Degtyarev and Voronkov, 1997] is sound and complete for first-order logic with equality, and relies on an incomplete but terminating algorithm for rigid E-unification.

Although we have presented matings methods as if they were entirely distinct from rewrite-based methods, this is of course not the case. In particular, the algorithm for rigid E-unification is based on a rule for rigid basic superposition, a variant of the rigid superposition rule [Bachmair *et al.*, 1992], which is itself a restricted form of paramodulation.

Our Method. In a typical TPS proof using equality reasoning, there will be many pairs of terms to be tested for rewrite-equivalence. Quite possibly, there will also be many different sets of equalities being used, meaning that we require a procedure that can take a set of equalities and two terms s and t, and very quickly either discover a rewrite proof of $s = t$ or terminate with

failure. In the interests of preserving the user's sanity, this procedure should also be reasonably predictable — for example, it should be guaranteed to discover all proofs in which the chain of rewrites (using the original equalities) is shorter than a given length. It is also desirable to produce rewrite proofs which can easily be translated into natural deduction style, or into partial matings.

Unfortunately, all of the alternative approaches seemed to generate more problems: they only worked for certain sorts of equalities, or they were not guaranteed to terminate on an arbitrary set of equalities (most completion procedures), or they would terminate eventually, but leaving the user no control over the complexity of the chain of rewrites (unfailing completion). Because of these problems we have retained the approach outlined above, although with some optimizations which will be described in the next section.

The ideal solution would probably be to use an external system which specializes in equational reasoning, such as EQP [McCune, 1997a]. Since TPS and EQP are now both integrated with the ΩMEGA mathematical assistant [Huang *et al.*, 1994; Benzmüller *et al.*, 1997; Benzmüller *et al.*, 1999], this should not be unreasonably difficult to implement. We discuss this further in Section 10.2.

7.5. Parity and Interpretations

The last step of the equality procedure contains one unfortunate feature: it is necessary to attempt to rewrite every possible pair of ground instances of the original terms. This makes unification rather more arduous than one would hope, since pairs of terms that cannot be rewritten to each other must still have rewrite trees generated for them, to the maximum possible depth, before we can say that they are definitely not rewritable. In practice, this is the major problem with our rewriting method — it can spend considerable amounts of time attempting to rewrite terms that (to the user's eyes) clearly do not have a common rewrite. For example, given the equalities defining $+$ in Peano Arithmetic ($\forall x.x+0 = x$ and $\forall x \forall y.x+S(y) = S(x+y)$), it is clearly futile to attempt to rewrite 0 to $S(0)$. There are at least two reasonable ways to justify this insight: either because under our usual interpretation of the symbols we know that these terms correspond to 0 and 1, and $0 \neq 1$, or because each equality preserves the number of occurrences of the symbol S in a term, and 0 has one fewer S than does $S(0)$. We now attempt to formalize these ideas and develop them further.

We can improve the efficiency of this part of the procedure by introducing some methods for pruning out obviously hopeless pairs of terms before we attempt to rewrite them. These methods fall into three classes, corresponding to considerations about top-level rewrites, parity and interpretations.

7.5.1. Top-Level Rewrites. Ordinarily, two rigid literals can only be mated if one is negated and the other is not; in particular, two ground literals can only be mated if one is the negation of the other. Because of the

possibility that a rewrite rule will introduce a negation, we are obliged by the algorithm above to consider pairs of literals even when both are negated (or neither is).

However, it is a trivial matter to inspect the rewrite rules to see whether any of them does in fact introduce a negation; if none does, then we can immediately discard a large number of potential connections without attempting to rewrite them. In most cases, rewrite rules do not introduce negations, and so this saves a lot of time in practice.

An immediate generalization of this idea is to check, before attempting to rewrite s to t, whether any top-level rewrite is possible for either term. If s and t have different head constants and no top-level rewrites are possible for either, then no amount of rewriting will make s equal to t, and we may discard this pair. If they begin with the same head constant but no top-level rewrites are possible then we may instead attempt to rewrite, pairwise, the arguments of this constant; this will generally be much faster than rewriting the entire term.

7.5.2. Parity. By inspection of the rewrite rules, we may discover that any rewrite proof will conserve either the number of occurrences of some constant symbol, or at least the parity (odd or even) of this number.

Examples: In the following, suppose x, y and z are variables, and all other symbols are constants.

- $x + y = y + x$ preserves the number of occurrences of all constants.
- $x + E = x$ preserves the number of occurrences of all constants except E and $+$.
- $[Fx[Fyz]] = [F[Fxy]z]$ preserves the number of occurrences of all constants.
- $x+[Sy] = S[x+y]$ preserves the number of occurrences of all constants.
- $x * [Ix] = E$ preserves the parity of all constants except E, I and $*$.

For example, under the "usual" axioms for groups, the parity of any symbol except the binary operator, inverse or identity will remain fixed during rewriting. Because of this, we can discard any pairs of terms in which some symbol (apart from the three listed) has odd parity in one term and even parity in the other.

Many standard axiomatizations of mathematical objects preserve parity in this way, and these cases can easily be detected automatically; in these cases, at least, we can improve the performance of the rewriting algorithm by considering the parity of the constant symbols.

7.5.3. Interpretation. It is possible that we can find an interpretation for the constant symbols of the rewrite rules, and that we have a way to quickly check whether this interpretation is valid (compare theory resolution [Stickel, 1985], which can also be regarded as adding a semantic component to the existing reasoning about syntactic properties of the search space).

For example, suppose that we have the following rewrite rules, where x, y and z are variables:

1. $Jxx = x$
2. $Mxx = x$
3. $J[Jxy]z = Jx[Jyz]$
4. $M[Mxy]z = Mx[Myz]$
5. $Jxy = Jyx$
6. $Mxy = Myx$
7. $M[Jxy]y = y$
8. $J[Mxy]y = y$

We observe that, interpreting M as $\wedge$ and J as $\vee$, and taking the variables to range over arbitrary Boolean expressions, all of these rewrite rules are valid. Hence, if these are the only rewrite rules available, then there can only be a rewrite proof of $s = t$ if s and t are equivalent under this interpretation.

We can confirm the validity of this interpretation by first replacing J and M with $\vee$ and $\wedge$, replacing the variables with new Boolean constants (regardless of their original types) and replacing equality with equivalence. Then each rewrite rule becomes a ground wff which we may test for validity using the TPS function `valid-p`.

In general, we may attempt to find an interpretation by using TPS's list of known Boolean functions and constants (binary functions `and`, `or`, `implies`, `equiv`, unary functions `not`, `id`, and constants $\top$, $\bot$), assigning them to function or constant symbols of the same arity, replacing variables with new constant symbols, and calling `valid-p`.

Interpretations are a sufficiently useful tool that it is worth attempting to find other possible interpretations for rewrite rules which could be checked from within TPS or Lisp. One possibility is $\mathbb{Z}$: we have plenty of candidates for binary functions (e.g. `*`, `+`, `-`, `<`, `>`, `max`, `min`), unary functions (e.g. `-`, `id`) and constants (e.g. `0`, `1`). Unfortunately, we have no equivalent for `valid-p`; the domains of these functions are infinite, and so rewrite rules involving variables would become universally-quantified theorems about integers (or rationals, or whatever our domain might be). Proving these theorems, especially in TPS, is non-trivial.

There are a number of possible ways around this problem, none of which is really satisfactory:

1. Recursively call TPS to prove the required theorems.
2. Build a database of known results for typical rewrite rules (such as associativity) into TPS.
3. Use another program to generate and/or check the interpretations.
4. Choose domains for which decision procedures exist, and implement those procedures in TPS.
5. Cheat (i.e. find a heuristic that sometimes works, and pretend it always works).

Option 1 is far too computationally expensive to even consider. Option 2 is possible, but would only work for a small subset of the possible rewrite rules that might occur in a theorem. Option 3 is more plausible, but it would

require significant work to integrate TPS and another theorem-prover (there are no programs known to this author which could generate infinite models of this sort). Some work has been done [Benzmüller *et al.*, 1999] to integrate TPS with ΩMEGA, but this integration is currently one-way (ΩMEGA can call TPS, but not vice-versa). Option 4 — adding decision procedures to TPS — would be a very useful development, with many more applications than simply pruning rewrite trees; it should probably be done in the most general way possible, and so represents a significant research project by itself. One such decision procedure (`valid-p`, above) is already implemented in TPS. Other candidates for decision procedures include Presburger arithmetic [Presburger, 1930] and the decidable fragment of the real numbers described in [Tarski, 1951]. A comparison of decision procedures for Presburger arithmetic can be found in [Janičić *et al.*, 1997], and an extensive survey of theorem proving in the real numbers is provided in [Harrison, 1996]. We could, of course, do the integration mentioned in option 3 and then use decision procedures that have been implemented in other systems, rather than implementing them again.

This leaves us with option 5, cheating. We implement interpretations from $\mathbb{Z}$ by simply substituting a few possible values for the variables, and checking the equalities under those substitutions. If the equalities are valid under these test values, then we accept the interpretation. It is clear that this will never reject a correct interpretation, but it might well accept an incorrect interpretation. We attempt to minimize the chances of this happening (for example, by choosing large random numbers as our test values), but we cannot eliminate the possibility; this is why the user is allowed to turn this feature off in our implementation. See the description of MS98-REWRITE-MODEL in Appendix A.

Under an incorrect interpretation, we may reject pairs s, t that are in fact rewrite-equivalent, and hence fail to find a proof of this. Hence the search procedure with interpretation of rewrite rules is incomplete. We emphasize that, even with potentially false interpretations, the search procedure is *not* unsound: a complete mating is still a complete mating. Also, note that Boolean interpretations do not have this incompleteness, since they are guaranteed to be correct.

7.6. Translation to a Natural Deduction Proof

At the end of the mating search, we will have a mating in which some of the "connections" are not really connections, in the sense that their alleged unifying substitution does not make them contradictory. We then use the stored chain of rewrites for the connection, plus the substitution for the complete mating, to replace that connection with a set of connections through duplicated copies of the equalities, in the obvious way. (A simple example: if we have mated the ground terms FAA and $\sim FBA$, modulo a rewrite $A = B$, we instead mate the literal FAA to the $\sim qA$ literal of the equality, and mate the qB literal of the equality to $\sim FBA$. Since this equality is guaranteed to

occur on all the paths through FAA and $\sim FBA$ — otherwise we wouldn't have used it — these two new connections still span all the paths that were spanned by the old connection.) While we do this, we can also generate the appropriate substitutions for the Leibniz variables in those equalities; they will all be of the form $\lambda w.C$, where this is in lambda-normal form and the term C contains exactly one instance of w.

This new set of connections will span all of the paths that were spanned by the original connection, and so the mating will still be complete; furthermore, it will be unifiable in the ordinary sense, and hence can now be merged and translated to a natural deduction proof in the normal way.

We now describe an algorithm for replacing a single "connection" justified by a sequence of rewriting steps with a set of unifiable connections.

NOTATION *If X is a term containing an occurrence of the term A, we write $X \stackrel{A=B}{\longrightarrow} Y$ to denote that Y is the result of replacing a single occurrence of A in X with B.*

REMARK It is immediate from the above definition that if $X \stackrel{A=B}{\longrightarrow} Y$, then we must have $X \stackrel{\beta}{=} [\lambda x.C(x)]A$ and $Y \stackrel{\beta}{=} [\lambda x.C(x)]B$ for some term $C(x)$ which contains exactly one free occurrence of x.

ALGORITHM 7.6.1. *Suppose that we have introduced a connection between two ground literals which is justified by a sequence of equality rewrite steps. Without loss of generality, we may assume that the two literals are of the form L and $\sim K$, and that the sequence of rewrites is of the form $L = A_0 \stackrel{B_0=B_0'}{\longrightarrow} A_1 \stackrel{B_1=B_1'}{\longrightarrow} \ldots \stackrel{B_i=B_i'}{\longrightarrow} A_{i+1} = K$, where each of the equalities in this sequence is either global or local to both L and $\sim K$. Also without loss of generality, we may assume that each of the equalities $B_j = B_j'$ is represented in the jform as $\forall q.\ \sim qB_j \vee qB_j'$ (the other case, $\forall q.\ \sim qB_j' \vee qB_j$, is exactly analogous, except that the substitution for q will have to introduce a negation which will be removed by the double negation rule in the literal $\sim qB_j'$).*

We introduce new connections as follows:

1. *If $i = 1$, then we take a duplicate copy of $\forall q.\ \sim qB_0 \vee qB_0'$, mate L to $\sim qB_0$ and $\sim K$ to qB_0'. These connections are obviously unifiable, by the remark above, and they clearly span any path through L and K (because $B_0 = B_0'$ occurs on all the vertical paths through L and K).*
2. *If $i > 1$, then we take a duplicate copy of $\forall q.\ \sim qB_0 \vee qB_0'$ and mate L to $\sim qB_0$. By the remark above, this induces a substitution for q such that $qB_0' \stackrel{\beta}{=} A_1$. All paths through L and K pass through either $\sim qB_0$ or qB_0', because $B_0 = B_0'$ occurs on all the vertical paths through L and K, and all such paths through $\sim qB_0$ are now spanned. We now recursively call this algorithm to construct a set of connections corresponding to the sequence of rewrites $qB_0' = A_1 \stackrel{B_1=B_1'}{\longrightarrow} \ldots \stackrel{B_i=B_i'}{\longrightarrow} A_{i+1} = K$ which span all the paths through L, K and qB_0'.*

REMARK By its construction, the above algorithm clearly terminates and produces a set of connections which are unifiable and span all of the paths through L and $\sim K$, as required. We may thus replace all of the connections which are justified by equality reasoning with true connections, and use the methods in [Pfenning, 1987] to translate the proof into natural deduction style.

Note that the complete mating may contain many more duplications than were produced by the mating search; in a sense, the Leibniz quantifiers no longer count towards the bound on the number of quantifiers allowed in a mating. As a result, we can generate huge matings (large enough that we have had to modify the procedure for merging expansion proofs [Pfenning, 1987], which was spending hours attempting to minimize the mating) without the usual problems of quantifier duplication.

CHAPTER 8

Optimizations and Variant Algorithms

In this chapter we describe and motivate some possible modifications to the search and unification algorithms presented previously.

8.1. A Parallel Version of the Search Algorithm

ALGORITHM 8.1.1. *The breadth-first character of the search procedure presented as Algorithm 4.1.1 allows the main loop of the algorithm to be divided among several independent processors, each of which only needs to know the acceptable connections and minimal components, as follows:*

Suppose we are given a jform J and n connected computers.

1. Fix all of the search bounds (unification depth, number of quantifier duplications and so on). Amplify J, by applying substitutions for set variables, duplicating outermost quantifiers, and (optionally) hiding equalities that can be handled by the unification algorithm, to produce a new jform J'.
2. Check each potential connection in J' for unifiability. If it is not unifiable, it will never be considered again; if it is unifiable, then store it as an *acceptable connection*. Make the acceptable connections globally visible.
3. Divide J' into fragments using the $\sim_J$ relation defined in Definition 3.1.2. For each fragment, find a computer which isn't doing anything (if necessary, wait for one to become available) and tell it to enumerate all of the minimal components which block that fragment. Collect up the resulting minimal components.
4. Choose one fragment f, and divide the minimal components into two lists: *current*, the list of all components blocking f, and *stored*, the list of all other components. Verify the unifiability of (at least) the components in *current*, discarding any non-unifiable components. If no components in *current* are unifiable, this fragment cannot be blocked; delete all minimal components which touch it, choose a new fragment and repeat this entire step.
5. Make the *stored* list globally visible. Partition the *current* list into at most n pieces, and send these pieces to different computers. Each computer then does the following:
   (a) For each element c of (this computer's piece of) *current*:

(i) If c is weak, choose a fragment which is weak in c. Combine each of the *stored* components which blocks this fragment with c, to produce a list of new components. Discard any elements of this list which are not unifiable.

(ii) Otherwise, if c is strong, check for completeness. If it's complete, we're done; return c and tell all the other computers to stop searching. If not, then the completeness check will return either a list of untouched fragments, one of which must be blocked, or an open path. If it returns a list of fragments, then combine c with each of the stored components which blocks any of these fragments. If it returns an open path, combine c with each of the acceptable connections on this path.

(b) If some new components were produced in step 5, set the local version of *current* to be the list of components produced, and return to step 5.

(c) Otherwise, all of the minimal components in the original local version of *current* (if there were any) have failed. Find another computer which is still working on the problem, get it to send half of its version of *current* to this computer, and return to step 5. If no other computer is still working on the problem, then all of the minimal components chosen in step 4 have failed, so return `failure`.

6. If we reached this step with all of the computers returning failure, then no complete mating can involve the fragment chosen in step 4. Delete all components touching this fragment, and return to step 4 to choose another starting fragment (unless no components remain, in which case either fail or return to step 1 and expand J' further).

This is a very sketchy algorithm, and the details of the communications between the various processors would need to be specified in much more detail for an actual implementation, as would the process of making something "globally visible" (which might, for example, mean either a blackboard architecture in which every processor can retrieve the information it needs, or actually keeping a local copy of the information on each processor).

Nonetheless, the basis for the algorithm is clear; not much information needs to be shared once the minimal components have been computed, and since component search works by incrementally adding minimal components to an existing mating, the various processors would not duplicate each others' work provided they started with different minimal components for the same fragment. The latter isn't strictly true, because the step in which open paths are spanned by adding single connections can (when MAX-MATES is not 1, or we have an unusual definition of $\sim_J$) result in a mating which is a superset of *two* different minimal components. However, this is not a common situation, and can easily be avoided completely by using a blackboard, visible to all processors, to store a list of those components which have already been tried.

The various improvements to the original algorithm, such the methods of controlling redundancy and the introduction of weightings for components, are also applicable to the parallel algorithm.

It is less obvious how (or whether) the path-enumeration searches in TPS could usefully be parallelized. There is a considerable body of research on parallel theorem-proving techniques, mostly based on resolution methods. Surveys of these techniques, and classifications of existing parallel theorem-provers according to their uses of parallelism, can be found in [Schumann, 1990; Kurfeß, 1990]. Parallel methods for first-order connection graphs are discussed in [Vagin and Salapina, 1998], and methods for first-order temporal logic using tableaux are discussed in [Johnson, 1994].

8.2. Alternative Definitions of a Fragment

The definition in $\sim_J$ given in Definition 3.1.2 is not the only possible equivalence relation which can be used to define fragments. In the remarks following Lemma 3.1.8 we observed that the crucial property for any usable alternative definition of $\sim_J$ is that any fragment touched by some minimal mating M must be blocked by M. In this section, to avoid confusion, $\sim_J$ will always refer to the equivalence relation given in Definition 3.1.2.

DEFINITION 8.2.1. *Let R and R' be relations. R is a* refinement *of R' (written $R \sqsubseteq R'$) iff for all x and y, $xRy \Rightarrow xR'y$.*

It will turn out that for all of our alternative equivalence relations $\sim'_J$, we have either $\sim'_J \sqsubseteq \sim_J$ or $\sim_J \sqsubseteq \sim'_J$. This is perhaps not too surprising, since our main objectives in defining such relations will be to deal with search spaces in which the fragments are either too small (there are jforms in which each fragment defined by $\sim_J$ contains exactly one literal, making the search behave almost like path-enumeration) or too large (producing unreasonable numbers of minimal components for each fragment).

The first, and most obvious, possibility for $\sim'_J$ is to remove the third condition of Definition 3.1.2, so that $A \sim'_J B$ iff every horizontal path through either A or B passes through both A and B. This allows fragments to span universal nodes in the jform, and so $\sim_J \sqsubseteq \sim'_J$. The example in Appendix C uses the fragments defined by $\sim'_J$; our implementation uses a flag BREAK-AT-QUANTIFIERS to determine whether fragments are allowed to span universal nodes or not. This is only a minor variation in the definition, and does not seem to make a great difference in most search spaces.

Another possibility is to define $\sim'_J$ to allow fragments containing more than one horizontal path. In this case, we fix a maximum number of horizontal paths k, and define the fragments by traversing the jform and finding the shallowest nodes N such that the subjform below N contains fewer than k horizontal paths. The sets of literals below these nodes form the fragments of the jform. In this case, again, $\sim_J \sqsubseteq \sim'_J$. To find the minimal components for fragments under this definition, we first enumerate the horizontal paths

through each fragment and then take all combinations of acceptable connections spanning these paths. In our implementation, the value of k can be set using the flag HPATH-THRESHOLD.

We could also define $\sim'_J \sqsubseteq \sim_J$ by the simple brute-force method of fixing some number k and breaking up any fragment of $\sim_J$ containing more than k literals into two or more parts. This has not been implemented, since we have few examples in which the fragments are too large.

Another way to redefine the fragments is to change the way in which the jform is constructed, by changing the definitions of some of the defined symbols. The main candidate for such a change is $A \equiv B$; we currently use the definition which minimizes the number of vertical paths, but we could equally well use the dual definition which minimizes the number of horizontal paths. See the discussion of the flag REWRITE-EQUIVS in Appendix A for more details.

8.3. Ordering Functions for Fragments

The algorithm sorts the fragments into a list l, and when it is required to choose a fragment from some set S, chooses the element of S which occurs earliest in l. The first element of this list will be the starting fragment chosen in step 4 of Algorithm 4.1.1, and the ordering of the remaining elements determines which fragment will be blocked next in step 5(a).

The ordering which is used at present has the following properties:

1. If $k < n$ then the kth copy of a fragment occurs before the nth copy. (The original is regarded as the 0th copy.)
2. Subject to the restriction above, fragments are ordered least-first by the number of minimal components they have.

Both of these restrictions are reasonable: the former is essential for our management of the duplicate copies, and helps us to prevent a great deal of redundancy in the search, and the latter is intended to reduce the branching of the search as much as possible. We can also attempt to look ahead from each of the minimal components and estimate the branching of the search at the next step.

For reasons outlined in Section 4.5, we probably also want to avoid starting with a fragment which lies below a set-variable substitution (if there are any such substitutions, and if their scope is not the entire jform), and we may opt to move the first such fragment to the beginning of the list so that it will be chosen as the starting fragment.

It would probably be useful to separate the two functions of this list, and decide the starting fragment independently of the ordering on the fragments. In the absence of any but the most obvious heuristics for choosing a starting fragment, this has not yet been implemented. Another possibility is to allow the choice of the next fragment to depend on the fragments which are already blocked, instead of (or as well as) on a global ordering such as the above. Again, this has not been done, for want of a good heuristic.

See the description of flags MS98-FRAGMENT-ORDER and MS98-BASE-PRIM, in Appendix A, for details of the orderings which have been implemented.

8.4. Weighting Functions for Components

In Algorithm 4.4.1 we introduce weights for the components, to reduce the branching of the search. The weighting which is used in the search is governed by the flag MS98-MEASURE, which is discussed in Appendix A. The default weighting simply counts the number of blocked components.

One would expect the choice of weighting to have a significant effect on the time taken to find a proof, but the alternative weightings which have been implemented generally seem to be inferior to the default weighting.

The point of weighting, of course, is to direct the search towards the most promising components, and so we require good heuristics for determining which components are most promising. It is not clear how to discover such heuristics, but this is the sort of thing that genetic algorithms and learning algorithms tend to be good at discovering (think of the weighting as a measure of "fitness"). Characterizing the properties of a component in a manner that is abstract enough to be applicable to many search spaces, so that TPS could learn from past experience, would be an interesting research project.

8.5. Pruning and Reduction Operations

Pruning is required in order to remove redundant components from the search space; a component can be considered redundant when it can safely be deleted without affecting the completeness of the search. The major case of redundancy which we encounter in component search is the case in which a new component contains the same mating as some other component already tried; we generate and store a hashtable of the keys corresponding to the matings which have been tried, and any pair of components will not be merged if the key of the resulting mating is found in the hashtable. New components which violate the search bounds in some way may also be considered redundant.

Theorem 6.1.9, which is used to control the duplications in the search, is one example of the way in which we attempt to prune redundancies out of the search space. We also Theorem 6.1.10 to check whether two minimal components for a fragment are equivalent, and hence to check whether a current component in which that fragment is to be blocked should be merged with one or both of them, (Suppose C is a component, M_1 and M_2 are minimal components for some fragment F, and σ is a relabelling such that $\sigma(F) = F$, $\sigma(C) = C$ and $\sigma(M_1) = M_2$. Then clearly $\sigma(C \cup M_1) = C \cup M_2$.) Note that this is relatively easy to compute, since we only need to generate and store mappings between minimal components.

Another example of search pruning is implemented by the flag MS98-VALID-PAIR, which rejects pairs of "parallel" connections under certain conditions (see Appendix A for the details).

Another pruning operation is applied silently in step 5(b) of the weighted search algorithm (Algorithm 4.4.1). When the completeness checker returns a list l of fragments, at least one of which must be blocked, we are supposed to merge the current component with all stored components which block any fragment in l. This is likely to make the search branch unacceptably, and is also likely to lead to the same mating being constructed in two ways. The latter is unimportant, since it will automatically be prevented by our implementation, but the former can cause trouble. Instead of blocking *every* fragment from l, we choose *one* fragment $f \in l$, and merge the current component with all of the stored components which block f. We then push a copy of the current fragment onto the *pending* list, marking it with both $l \setminus \{f\}$ and a note that the fragment f is not to be touched in any extensions of this component. If the copied component ever returns to the top of the list, we may repeat this process and choose another element from $l \setminus \{f\}$. This will not loop (since, every time we do this, we create all of the immediate extensions which touch f and then mark the copied component with instructions not to create any more extensions touching f), and it obviously does not affect completeness since every possibility will get considered in the long run.

Reduction operations, by contrast, alter the search space itself, usually by removing paths entirely. The only reduction operation in our search is the equality-hiding which happens at the start of Algorithm 7.3.1.

In the first-order and propositional cases, particularly for proofs using analytic tableaux, both pruning methods (such as path orderings and regular paths) and reduction methods (such as path dissolution, subsumption and tautology elimination) have been widely studied and found to be extremely powerful. Some of the results in this area were mentioned in Section 2.6. Most of these methods can be generalized to the case of higher-order mating search, but it is doubtful whether they will be as useful in the higher-order case.

For example, removing all subsumed paths in a propositional negation-normal formula is already an NP-hard problem [Ramesh *et al.*, 1995]. The lifting of this idea to first-order uses first-order unification, and unification instances of clauses. To lift the idea to higher-order logic would require us to deal with a branching unification algorithm in which there is no most general unifier; we would be doing much more work to find many fewer subsumptions.

Similar objections can be applied to the higher-order versions of the other methods above. This is not to say that they cannot be used at all for higher-order logic — only that they should not be used indiscriminately. It would be interesting to identify some higher-order situations in which these methods would be effective.

8.6. Skolem Nodes and Selection Nodes

As discussed in Chapter 2, we may replace essentially existential quantifiers either by using Skolemization or by introducing selection parameters; the choice is governed by the flag SKOLEM-DEFAULT (see Appendix A). A full discussion of Skolemization and selection can be found in [Miller, 1983].

Previous search algorithms in TPS have used Skolemization by default, but selection nodes are often preferable for component search. There are two major reasons for this:

1. Skolem terms can require many arguments; this makes the unification problem more difficult, especially in the presence of the unification algorithm of Chapter 5.
2. The implementation of set-variable substitution in TPS usually results in the entire substituted term becoming an argument to a Skolem function. Not only does this make the term itself longer, and the unification problem more difficult, it also introduces an incompleteness into the search since TPS cannot determine that, for example, $F[\lambda x.Ax \wedge Ax]$ ought to be unifiable with $F[\lambda x.Ax]$.

Unfortunately, using selection parameters can also increase the number of constants in the search space (duplicating a literal which contains a Skolem term results in a new literal with the *same* Skolem term taking different arguments; duplicating a literal with a selection parameter results in a new literal with a *different* selection parameter). This can make unification less tractable, and also interferes with the construction of unification graphs for duplicate copies of connections (relabelling the old unification graph as in Section 5.5 is no longer correct).

Even so, the arguments against Skolemization in component search are substantial enough that it should generally not be used, especially in searches involving set-variable substitution.

Note that if we are using selection parameters, the acyclicity requirement on the imbedding relation (Definitions 2.3.7 and 2.3.8) allows us to prune the unifiers as the component is constructed; we modify the *Merge* algorithm (Algorithm 5.2.7) to prune out from the unification graph any unifiers violating the acyclicity condition. The required modification to the algorithm is simple; we recursively pass an argument to *Merge* which contains the current unifier (i.e. the path from the root of the graph to the current node), and if the expansion variable at the current node has some selection parameter below it in the jform then we check each of the substitutions which will be created by merging and reject any that violate acyclicity. This is best done during *Merge* since we have structure-sharing of unification graphs, and so it is easier to create the merged graph already pruned than to prune it later (which would require us to copy it).

Furthermore, at the start of the search, for each variable x we disallow all substitutions $\{x \mapsto t\}$ in which t contains some selection parameter b

occurring below the expansion node for x in the jform, since these trivially violate acyclicity ($b \prec_{(\{x \mapsto t\},J)} b$ is immediate from Definition 2.3.7).

Finally, we remark that in the presence of bounds on the number of duplications or substitutions for set variables which may occur in a component, we may (and do) implement a pruning algorithm for unification graphs which removes all the unifiers that use selection parameters which occur in parts of the jform which cannot be reached (due to the search bounds) by extensions of the current component.

8.7. Instantiating Definitions

Dual instantiation is a method for handling definitions in the search; it is described in [Bishop and Andrews, 1998]. The essence of the idea is that a definition is replaced by a conjunction of its instantiated and uninstantiated forms, and the search is made to prefer the uninstantiated form; this produces shorter and faster proofs in the case where some definition is not needed by the search, without significantly affecting the search in the case where all the definitions are needed.

Given a fragment ordering which prefers the uninstantiated literal (which, under the usual definition of $\sim_J$, will be a fragment all by itself) to the fragments contained in the instantiated literal, and given a reasonable choice of weightings if weights are being used, the desirable aspects of dual instantiation are largely preserved by component search.

8.8. Variable Orderings for Unification Graphs

Research into ordered binary decision diagrams has determined that the choice of variable ordering critically affects the size of the resulting graph. For example, [Bryant, 1986] demonstrates that the Boolean function $x_1 \cdot x_2 + \ldots + x_{2n-1} \cdot x_{2n}$ under the ordering $x_1 < x_2 < \ldots < x_{2n}$ produces a binary decision diagram with $2n + 2$ vertices, whereas the same function under the ordering $x_1 < x_{n+1} < x_2 < x_{n+2} < \ldots < x_n < x_{2n}$ requires 2^{n+1} vertices. In general, it appears to be for the best if variables which occur close to each other in the formula also occur close to each other in the ordering.

Although unification graphs are different in many ways from ordered binary decision diagrams, it would be surprising if an analogous result were not true for unification graphs. Consequently, the choice of a variable ordering will probably affect the efficiency of the search procedure, and we now consider some of the possible orderings that might be used.

Our variable orderings allow for multiple variables to occur at a single node. It is clear from the definitions that the ordering of the variables within the interior lists is irrelevant to the size of the decision diagram. The critical issues for us will be the ordering of the outer list, and the way in which the variables are divided up among the interior lists.

We consider the interior lists first; how might we partition the variables among these lists?

Unitary: Each variable occurs in a separate interior list; all the interior lists have length 1.

By duplication: x and y occur in the same interior list iff y is a duplicate copy of x or x is a duplicate copy of y.

By closeness: x and y occur in the same inner list iff they are bound by the same universal node of the jform.

One might, of course, define other possible partitions. The advantages of the unitary method are that it minimizes the branching of the unification graph (while maximizing its depth, unfortunately). The advantages of partitioning by duplication are that given a unification graph for a connection (l_1, l_2) it is comparatively trivial, at least in some cases, to generate a graph for (l_1', l_2'), where these are assumed to be duplicate copies of l_1 and l_2. Furthermore, if a quantifier is duplicated during the search it is easy to relabel the old graphs to correspond to the new jform. These issues are discussed in more detail in Section 5.5. The advantages of partitioning by closeness are that it groups together the variables which are likely to occur together (and is most similar to the methods used in ordered binary decision diagrams).

We now turn to methods for ordering the outer list

By occurrence: Take the interior lists in the order they turn up.

By number of unifiers: Take the product of the number of substitutions that have been generated for each of the variables in the interior list, and order the outer list either least-first or greatest-first.

By closeness: Define some measure of how close two interior lists are to each other (for example, by counting how many literals contain variables from both lists), and attempt to minimize the distance between the closest interior lists.

Ordering by occurrence is sufficient for many smaller problems, and requires no computational effort; it tends to put related variables close to each other in the ordering, since the variables are found by a depth-first traversal of the expansion tree. Ordering by number of unifiers appears to be a good choice for larger problems (particularly the greatest-first ordering, since it maximizes the chances of being able to reuse subgraphs). Ordering by closeness is ill-defined as outlined above, although it might be possible to create a useful definition for it.

The flag MS98-VARIABLE-ORDER governs the choice of variable ordering, and is discussed in Appendix A.

Finally, we consider another possible optimization: dynamic reordering. This is widely used in implementations of BDDs; it attempts to reduce the size of the graphs by switching neighboring pairs of variables in the ordering and checking to see if this switch has made the graph smaller or larger. If the graph has become smaller, then the new ordering is used. This may be called repeatedly until a locally optimal ordering is reached. Implementing this technique for unification graphs would be an interesting project, although it might well be too expensive to use frequently.

8.9. Alterations to the Unification Algorithm

Besides the alterations to unification discussed above, it is also not entirely necessary to compute the unification graphs of every minimal component at the beginning of the search; we could at least wait and see whether the component will ever be used. Of course, we will still want some kind of check on the unifiability of the minimal components. The flag MS98-MERGE-DAGS governs how stringent this check will be; the available checks range from pairwise unifiability of the constituent connections to computing the unification graph in full.

Another optimization determines the unifiers which are computed when two components are merged. We observe that many of our weightings are in fact computable without knowing anything about the unifiers of a component; consequently, if we know that the component being constructed will have a lower weight than some other new component (and hence will not appear in the next round), then we can delay computing its unifiers. Again, we use a quick check for unifiability even if the unifiers are not to be explicitly computed, to avoid constructing non-unifiable components.

8.10. Extensions of the Equality Algorithm

The method for reasoning with equalities which was introduced in Chapter 7 can in principle allow the search procedure to declare that a connection (s,t) is acceptable under some substitution σ, regardless of whether or not $\sigma(s) \stackrel{\beta}{=} \sim\sigma(t)$, and to use that connection exactly as if it were a normal connection for the duration of the search. In Chapter 7, we used this fact in order to allow connections in which $\sigma(s)$ and $\sim\sigma(t)$ could be made identical after some bounded number of rewrite steps using some of the equalities in the jform.

The same approach would allow us to introduce more complex justifications for a connection, perhaps by integrating an external system to TPS, or by implementing one of the decision procedures discussed in Section 7.5.3. Of course, the translation of a complete mating to natural deduction form would become more complicated; one would probably either have to introduce a new rewrite justification into the expansion tree (so $\sigma(s)$, which was previously a leaf, would be rewritten to a new leaf $\sigma(t)$ with justification "External", and the resulting natural deduction proof would derive $\sigma(t)$ from $\sigma(s)$ in a single step with a similar justification), or one would have to find a translation from the proof produced by the external system to an ET-proof. The latter is more difficult, but has the advantage of making natural deduction proofs in TPS self-contained and verifiable from within TPS.

CHAPTER 9

Results

9.1. Basic First-Order Problems

The following is a list of some of the first-order problems which had already been solved by path-enumeration searches in TPS.

X2106: $\forall x[Rx \supset Px] \wedge \forall x[\sim Qx \supset Rx] \supset \forall x.Px \vee Qx$
X2107: $Rab \wedge \forall x\forall y[Rxy \supset Ryx \wedge Qxy] \wedge \forall u\forall v[Quv \supset Quu] \supset Qaa \wedge Qbb$
X2108: $\forall x\exists y.Px \supset Py$
X2109: $\exists x[p \wedge Qx] \equiv p \wedge \exists xQx$
X2110: $\exists xRx \wedge \forall y[Ry \supset \exists zQyz] \wedge \forall x\forall y[Qxy \supset Qxx] \supset \exists x\exists y.Qxy \wedge Ry$
X2111: $\forall x[\exists yPxy \supset \forall yQxy] \wedge \forall z\exists yPzy \supset \forall y\forall xQxy$
X2112: $\exists v\forall xPxv \wedge \forall x[Sx \supset \exists yQyx] \wedge \forall x\forall y[Pxy \supset \sim Qxy] \supset \exists u.\sim Su$
X2113: $\forall y\exists wRyw \wedge \exists z\forall x[Px \supset \sim Rzx] \supset \exists x.\sim Px$
X2114: $\forall xRxb \wedge \forall y[\exists zRyz \supset Ray] \supset \exists u\forall vRuv$
X2115: $\forall x[\exists yPxy \supset \forall zPzz] \wedge \forall u\exists v[Puv \vee Mu \wedge Q.fuv] \wedge \forall w[Qw \supset \sim M.gw] \supset \forall u\exists v.P[gu]v \wedge Puu$
X2116: $\forall x\exists y[Px \supset Rx[g.hy] \wedge Py] \wedge \forall w[Pw \supset P[gw] \wedge P.hw] \supset \forall x.Px \supset \exists y.Rxy \wedge Py$
X2117: $\forall u\forall v[Ruu \equiv Ruv] \wedge \forall w\forall z[Rww \equiv Rzw] \supset .\exists xRxx \supset \forall yRyy$
X2118: $\forall x[p \wedge Qx \vee \sim p \wedge Rx] \supset \forall xQx \vee \forall xRx$
X2119: $\exists y\forall x.Py \supset Px$
X2120: $\forall u\forall v\forall w[Puv \vee Pvw] \supset \exists x\forall yPxy$
X2121: $\exists v\forall y\exists z.Pay[hy] \vee Pvy[fy] \supset Pvyz$
X2122: $\exists xRxx \supset \forall yRyy \supset \exists u\forall v.Ruu \supset Rvv$
X2123: $\exists y[Py \supset Qx] \supset \exists y.Py \supset Qy$
X2124: $\exists x[Px \supset Qx] \equiv \forall xPx \supset \exists xQx$
X2125: $\exists x\forall y[Px \equiv Py] \equiv .\exists xPx \equiv \forall yPy$
X2126: $\forall x[Px \equiv \exists yPy] \equiv .\forall xPx \equiv \exists yPy$
X2127: $\exists x\forall y[Py \equiv Px] \supset \forall xPx \vee \forall x.\sim Px$
X2128: $\forall x[Px \equiv \forall yPy] \equiv .\exists xPx \equiv \forall yPy$
X2129: $\exists x\forall y[Px \equiv Py] \equiv [\exists xQx \equiv \forall yPy] \equiv .\exists x\forall y[Qx \equiv Qy] \equiv .\exists xPx \equiv \forall yQy$
X2130: $\forall xPx \supset \sim\exists yQy \vee \exists z.Pz \supset Qz$
X2131: $\forall xPx \supset \exists y.\forall x\forall zQxyz \supset \sim\forall z.Pz \wedge \sim Qyyz$
X2132: $\forall w[\sim Rww] \supset \exists x\exists y.\sim Rxy \wedge .Qyx \supset \forall zQzz$
X2133: $\forall x[\exists yQxy \supset Px] \wedge \forall v\exists uQuv \wedge \forall w\forall z[Qwz \supset Qzw \vee Qzz] \supset \forall zPz$
X2134: $\forall z\exists x[\forall yPxy \vee Qxz] \supset \forall y\exists x.Pxy \vee Qxy$
X2135: $\exists x\forall y.Px \wedge Qy \supset Qx \vee Py$
X2136: $\exists x\exists y\forall u.Pxyz \supset Puxx$
X2137: $\exists x\forall y.Px \supset Qx \vee Py$

X2138: $\forall x \exists y F x y \wedge \exists x \forall e \exists n \forall w [S n w \supset D w x e] \wedge \forall e \exists d \forall a \forall b [D a b d \supset \forall y \forall z . F a y \wedge F b z \supset D y z e] \supset \exists y \forall e \exists m \forall w . S m w \supset \forall z . F w z \supset D z y e$

X2150: $\exists W . \forall X \exists Y P W X Y \supset \exists Z P Z Z W$

X3411: $\forall y [\forall x P x y \supset \forall u Q u y] \supset \exists z . \exists v P v z \supset \exists w . P z w \vee Q w z$

Y2141: $\forall x \exists y [P x \wedge . Q y \vee Q x] \supset \exists z . P z \wedge Q z$

PELL19: $\exists x \forall y \forall z . P y \supset Q z \supset . P x \supset Q x$

PELL21: $\exists x [p \supset F x] \wedge \exists x [F x \supset p] \supset \exists x . [p \supset F x] \wedge . F x \supset p$

PELL25: $\exists x P x \wedge \forall x [F x \supset \sim G x \wedge R x] \wedge \forall x [P x \supset G x \wedge F x] \wedge [\forall x [P x \supset Q x] \vee \exists x . P x \wedge R x] \supset \exists x . Q x \wedge P x$

PELL26: $[\exists x P x \supset \exists x Q x] \wedge [\exists x Q x \supset \exists x P x] \wedge \forall x \forall y [P x \wedge Q y \supset [R x \supset S y] \wedge . S y \supset R x] \supset [\forall x [P x \supset R x] \supset \forall x . Q x \supset S x] \wedge . \forall x [Q x \supset S x] \supset \forall x . P x \supset R x$

PELL27: $\exists x [F x \wedge \sim G x] \wedge \forall x [F x \supset H x] \wedge \forall x [J x \wedge I x \supset F x] \wedge [\exists x [H x \wedge \sim G x] \supset \forall x . I x \supset \sim H x] \supset \forall x . J x \supset \sim I x$

PELL28: $\forall x [P x \supset \forall x Q x] \wedge [\forall x [Q x \vee R x] \supset \exists x . Q x \wedge S x] \wedge [\exists x S x \supset \forall x . F x \supset G x] \supset \forall x . P x \wedge F x \supset G x$

PELL29: $\exists x F x \wedge \exists x G x \supset [\forall x [F x \supset H x] \wedge \forall x [G x \supset J x] \supset \forall x \forall y . F x \wedge G y \supset H x \wedge J y] \wedge . \forall x \forall y [F x \wedge G y \supset H x \wedge J y] \supset \forall x . G x \supset J x$

PELL35: $\exists x \exists y . P x y \supset \forall x \forall y P x y$

PELL40: $\exists y \forall x [F x y \equiv F x x] \supset \sim \forall x \exists y \forall z . F z y \equiv \sim F z x$

PELL42: $\sim \exists y \forall x . [F x y \supset \sim \exists z . F x z \wedge F z x] \wedge . \sim \exists z [F x z \wedge F z x] \supset F x y$

RUSSELL1: $\sim \exists U \forall V . E V U \equiv \sim E V V$

THM25: $\sim \exists Y \forall X . R X Y \equiv \sim \exists Z . R X Z \wedge R Z X$

THM31: $R \subseteq S \equiv R \cap S = R$

THM39: $R \cup S = R \cap S \supset R \subseteq S$

THM68: $\forall X$ likes X bruce $\wedge \forall Y [\exists Z$ likes $Y Z \supset$ likes lyle $Y] \supset \exists U \forall V$ likes $U V$

THM69: $\forall X [\exists Y P X Y \supset \forall Z P Z Z] \wedge \forall R \exists S [P R S \vee M R \wedge Q . F R S] \wedge \forall W [Q W \supset \sim M . G W] \supset \forall U \exists V . P [G U] V \wedge P U U$

THM72: $\forall X \exists Y [P X \supset L X [G . H Y] \wedge P Y] \wedge \forall W [P W \supset P [G W] \wedge P . H W] \supset \forall X . P X \supset \exists Y . L X Y \wedge P Y$

THM75: $\exists v \forall j \exists q . P a j [h j] \vee P v j [k j] \supset P v j q$

THM100: $P S S S S O \wedge \forall T [P S N S N T \supset P N N S N . A T]$
$\wedge \forall T^1 [P N N S N T^1 \supset P S N S N . A T^1]$
$\wedge \forall T^2 [P S S N S T^2 \supset P N S N S . A T^2] \wedge \forall T^3 [P N S N S T^3 \supset P S S N S . A T^3]$
$\wedge \forall T^4 [P S S S N T^4 \supset P N N S N . W T^4] \wedge \forall T^5 [P N N S N T^5 \supset P S S S N . W T^5]$
$\wedge \forall T^6 [P S S N S T^6 \supset P N N N S . W T^6] \wedge \forall T^7 [P N N N S T^7 \supset P S S N S . W T^7]$
$\wedge \forall X \forall Y \forall U [P S X S Y U \supset P N X N Y . G U]$
$\wedge \forall X^1 \forall Y^1 \forall V [P N X^1 N Y^1 V \supset P S X^1 S Y^1 . G V]$
$\wedge \forall T^8 [P S N S S T^8 \supset P N N S N . C T^8] \wedge \forall T^9 [P N N S N T^9 \supset P S N S S . C T^9]$
$\wedge \forall U^1 [P S S N S U^1 \supset P N S N N . C U^1] \wedge \forall V^1 [P N S N N V^1 \supset P S S N S . C V^1]$
$\supset \exists Z P N N N N Z$

THM147: $\sim . \forall p \forall q [\sim T [imp\ pq] \vee \sim T p \vee T q] \wedge \forall p \forall q T [imp\ p . imp\ qp] \wedge$
$\forall p \forall q \forall r T [imp\ [imp\ p . imp\ qr] . imp\ [imp\ pq] . imp\ pr] \wedge$
$\forall p \forall q T [imp\ [imp\ [nt\ p] . nt\ q] . imp\ qp] \wedge \exists a . \sim T . imp\ aa$

The following table gives times, in seconds, for the mating search only (i.e. not including the translation to natural deduction) for each of the theorems above. The measurements were taken on a Dell System Workstation 400MT, with dual 300 MHz Intel Pentium II processors and 500MB of RAM, running RedHat Linux 4.1; TPS was compiled using Allegro Common Lisp

4.3. Two searches were used: the path-enumeration search MS90-3, and the component search MS98-1. No attempt was made to find optimal modes for these theorems; the mode MODE-X2129-A was used for all the searches in MS90-3, and the mode MS98-FO-MODE was used for all searches in MS98-1.

Theorem	MS90-3	MS98-1
X2106	0.05	0.15
X2107	0.57	0.59
X2108	0.01	0.07
X2109	0.08	0.11
X2110	0.08	0.46
X2111	0.06	0.14
X2112	0.10	0.44
X2113	0.06	0.16
X2114	0.08	0.16
X2115	3.39	1.39
X2116	22.77	0.67
X2117	0.36	1.24
X2118	0.04	0.21
X2119	0.02	0.06
X2120	0.04	0.17
X2121	0.13	0.11
X2122	0.04	0.10
X2123	0.03	0.07
X2124	0.06	0.17
X2125	0.14	1.67
X2126	0.15	0.75
X2127	0.10	0.28
X2128	0.16	0.81
X2129	1.30	>1000
X2130	0.04	0.10
X2131	0.07	0.19
X2132	0.10	0.22
X2133	0.17	1.03

Theorem	MS90-3	MS98-1
X2134	0.08	0.23
X2135	0.04	0.13
X2136	0.03	0.12
X2137	0.04	0.11
X2138	0.56	1.59
X2150	0.05	0.14
X3411	0.12	0.25
Y2141	0.06	0.38
PELL19	0.08	0.20
PELL21	0.06	0.28
PELL25	0.14	0.90
PELL26	>1000	1.71
PELL27	0.94	0.91
PELL28	0.40	0.52
PELL29	0.28	0.75
PELL35	0.05	0.21
PELL40	0.16	1.11
PELL42	0.11	43.77
RUSSELL1	0.04	0.19
THM25	0.10	0.69
THM31	0.20	2.61
THM39	0.12	1.30
THM68	0.10	0.22
THM69	3.43	1.43
THM72	23.03	0.73
THM75	0.10	0.10
THM100	>1000	68.05
THM147	4.60	2.87

FIGURE 9.1. Search times for first-order theorems, in seconds.

Most of these search spaces are quite small, and the associated searches took less than a second, so it is not too surprising that path-enumeration was generally the faster of the two procedures (since component search has to spend time enumerating connections and minimal components before it even begins searching). For reasons discussed earlier, component search was not expected to be a particularly good search procedure for first-order problems; the results above demonstrate that it isn't entirely hopeless, at least for smaller problems.

9.2. Basic Higher-Order Problems

We begin by considering some higher-order problems which can be proven by path-enumeration as well as by component search:

X5200: $x_{o\alpha} \cup y_{o\alpha} = \bigcup .\lambda v_{o\alpha}.v = x \vee v = y$
X5201: $x_{o\alpha} \cap y_{o\alpha} = \bigcap .\lambda v_{o\alpha}.v = x \vee v = y$
X5202: $\#f_{\alpha\beta}[x_{o\beta} \cup y_{o\beta}] = \#fx \cup \#fy$
X5203: $\#f_{\alpha\beta}[x_{o\beta} \cap y_{o\beta}] \subseteq \#fx \cap \#fy$
X5205: $\#f_{\alpha\beta}[\bigcap w_{o(o\beta)}] \subseteq \bigcap .\#[\#f]w$
X5207: $\#f_{\alpha\beta}[x_{o\beta} \cap y_{o\beta}] \subseteq \#fx \cap \#fy$
X5208: $\exists S_{o\iota}\forall x_{\iota}[[Sx \vee P_{o\iota}x] \wedge . \sim Sx \vee Q_{o\iota}x] \equiv \forall y_{\iota}.Py \vee Qy$
X5209: $\mathfrak{P}_{o(o\alpha)(o\alpha)}[D_{o\alpha} \cap E_{o\alpha}] = \mathfrak{P}D \cap \mathfrak{P}E$
X5212: $\lambda z_{\alpha}\exists x_{\beta}[g_{o\beta}x \wedge z = f_{\alpha\beta}x] = \#fg$
X5304: $\sim\exists g_{o\alpha\alpha}\forall f_{o\alpha}\exists j_{\alpha}.gj = f$
X5305: $\forall s_{o\alpha}. \sim\exists g_{o\alpha\alpha}\forall f_{o\alpha}.f \subseteq s \supset \exists j_{\alpha}.sj \wedge gj = f$
X5308: $\exists j_{\beta(o\beta)}\forall p_{o\beta}[\exists x_{\beta}px \supset p.jp] \supset .\forall x_{\alpha}\exists y_{\beta}r_{o\beta\alpha}xy \equiv \exists f_{\beta\alpha}\forall xrx.fx$
X5310: $\forall r_{o\beta(o\beta)}[\forall x_{o\beta}\exists y_{\beta}rxy \supset \exists f_{\beta(o\beta)}\forall xrx.fx] \supset \exists j_{\beta(o\beta)}\forall p_{o\beta}.\exists z_{\beta}pz \supset p.jp$
X6004: $E_{o(o\alpha)(o\beta)}[= x_{\beta}]. = y_{\alpha}$
THM15B: $\forall f_{\iota\iota}.\exists g_{\iota\iota}[\text{ITERATE+ } fg \wedge \exists x_{\iota}.gx = x \wedge \forall z_{\iota}.gz = z \supset z = x] \supset \exists y_{\iota}.fy = y$

The abbreviations used in these theorems are as follows:

$\cup$: $\lambda P_{o\alpha}\lambda R_{o\alpha}\lambda z_{\alpha}.Pz \vee Rz$
$\cap$: $\lambda P_{o\alpha}\lambda R_{o\alpha}\lambda x_{\alpha}.Px \wedge Rx$
$\bigcup$: $\lambda D_{o(o\alpha)}\lambda x_{\alpha}\exists S_{o\alpha}.DS \wedge Sx$
$\bigcap$: $\lambda D_{o(o\alpha)}\lambda x_{\alpha}\forall S_{o\alpha}.DS \supset Sx$
$\#$: $\lambda f_{\alpha\beta}\lambda x_{o\beta}\lambda z_{\alpha}\exists t_{\beta}.xt \wedge z = ft$
$\mathfrak{P}$: $\lambda P_{o\alpha}\lambda R_{o\alpha}.R \subseteq P$
$\subseteq$: $\lambda P_{o\alpha}\lambda R_{o\alpha}\forall x_{\alpha}.Px \supset Rx$
E: $\lambda p_{o\beta}\lambda q_{o\alpha}\exists s_{\alpha\beta}.\forall x_{\beta}[px \supset q.sx] \wedge \forall y_{\alpha}.qy \supset \exists_1 x.px \wedge y = sx$
$\exists_1$: $\Sigma^1_{o(o\alpha)}.\lambda x_{\alpha}A_o$
Σ^1: $\lambda P_{o\alpha}\exists y_{\alpha}.P = [= y]$
ITERATE+: $\lambda f_{\alpha\alpha}\lambda g_{\alpha\alpha}\forall p_{o(\alpha\alpha)}.pf \wedge \forall j_{\alpha\alpha}[pj \supset p.f \circ j] \supset pg$
$\circ$: $\lambda f_{\alpha\beta}\lambda g_{\beta\gamma}\lambda x_{\gamma}f.gx$

The following table gives times (in seconds) for the mating search for each of the theorems above. Each time is followed by the percentage of that time which was spent on unification. The fastest known modes were used for each search. Again, most of these search spaces are too small for component search to have a noticeable advantage over path-enumeration. However, even for these smaller problems, component search spends a significantly smaller proportion of its time on unification than does path-enumeration (46% of the total time, as opposed to 82% for path-enumeration).

Theorem	Path-enum search (% unif)	Component search (% unif)
X5200	0.62 (95%)	1.09 (16%)
X5201	0.13 (85%)	1.09 (9%)
X5202	4.42 (91%)	1.12 (28%)
X5203	0.74 (70%)	0.47 (38%)
X5205	0.73 (62%)	0.43 (63%)
X5207	0.63 (90%)	0.15 (27%)
X5208	0.21 (95%)	0.17 (53%)
X5209	0.01 (0%)	0.25 (4%)
X5210	0.37 (89%)	0.65 (40%)
X5211	5.22 (29%)	5.36 (24%)
X5212	0.36 (81%)	0.18 (39%)
X5304	0.02 (50%)	0.18 (67%)
X5305	0.4 (43%)	0.55 (45%)
X5308	0.26 (50%)	0.25 (64%)
X5310	25.71 (95%)	8.44 (81%)
X6004	0.21 (76%)	0.30 (17%)
THM15B	4.16 (60%)	15.59 (42%)

FIGURE 9.2. Search times for higher-order theorems, in seconds.

9.3. Previously Intractable Problems

The results presented so far have not been particularly startling; on the theorems which could already be proven by TPS, component-based search is comparable to path-enumeration search — better in some cases, worse in others. However, the real point of component-based search was to provide a way to tackle larger search spaces which are intractable for existing search procedures.

The following theorems have not previously been proven by any other search procedure in TPS, but can be proven automatically by component search.

THM145.

$$\forall R_{o\alpha\alpha}\forall U_{\alpha(o\alpha)}.\text{TRANSITIVE } R \wedge \forall s_{o\alpha}[\forall z_{\alpha}[sz \supset Rz.Us] \wedge \forall j_{\alpha}.\forall k_{\alpha}[sk \supset Rkj] \supset R[Us]j] \supset \forall f_{\alpha\alpha}.\forall x_{\alpha}\forall y_{\alpha}[Rxy \supset R[fx].fy] \supset \exists w_{\alpha}.Rw[fw] \wedge R[fw]w$$

This is a fixed point theorem for lattices, in a formulation due to Coquand. The theorem is originally due to Knaster [Knaster, 1927] (see [Szász, 1963], Theorem 62); it states that in a complete lattice, every monotone function f has a fixed point. U is is the least upper bound function from non-empty sets to individuals, and R is the ordering on the lattice. The theorem requires no set-variable substitutions, and only a single quantifier duplication, but the unifying substitutions are rather large. TPS finds the

fixed point $U[\lambda x.Rx[fx]]$ and completes the proof in 59.2 minutes, of which 50.9 minutes (86%) are spent on unification.

THM126.

$$\forall h1_{\beta\gamma}\forall h2_{\alpha\beta}\forall s1_{o\gamma}\forall f1_{\gamma\gamma\gamma}\forall s2_{o\beta}\forall f2_{\beta\beta\beta}\forall s3_{o\alpha}\forall f3_{\alpha\alpha\alpha}$$
$$.\text{HOM2 } h1\ s1\ f1\ s2\ f2 \wedge \text{HOM2 } h2\ s2\ f2\ s3\ f3$$
$$\supset \text{HOM2 } [h2 \circ h1]\ s1\ f1\ s3\ f3$$

where

$$\begin{aligned}
\text{HOM2} &= \lambda k_{\alpha\beta}\lambda r_{o\beta}\lambda f_{\beta\beta\beta}\lambda s_{o\alpha}\lambda g_{\alpha\alpha\alpha}.f\text{-CLOSED2 } r \wedge g\text{-CLOSED2 } s \\
&\quad \wedge \text{MAPS }_{o(o\alpha)(o\beta)(\alpha\beta)} krs \\
&\quad \wedge \forall x_\beta \forall y_\beta . rx \wedge ry \supset k[fxy] = g[kx].ky \\
-\text{CLOSED2} &= \lambda k_{\alpha\alpha\alpha}\lambda u_{o\alpha}\text{MAPS2 } kuuu \\
\text{MAPS2} &= \lambda k_{\alpha\beta\gamma}\lambda u_{o\gamma}\lambda v_{o\beta}\lambda z_{o\alpha}\forall x_\gamma \forall y_\beta . ux \wedge vy \supset z.kxy \\
\text{MAPS} &= \lambda k_{\alpha\beta}\lambda u_{o\beta}\lambda v_{o\alpha}\forall x_\beta . ux \supset v.kx
\end{aligned}$$

This states that the composition of two homomorphisms is a homomorphism; a version of this theorem (formulated in the first-order set theory of Gödel, von Neumann and Bernays) was the central theorem of [Boyer *et al.*, 1986]. A simpler version, THM133, in which the sets $s1$, $s2$ and $s3$ are replaced by types α, β and γ (making the closure assumptions unnecessary) is easily provable by any search procedure in TPS, but the search space of THM126 is much larger than that of THM133. The jforms of Figures 3.1 and 3.2 are jforms for this theorem. The search takes 18.28 seconds, of which 2.26 seconds (12%) are spent in unification.

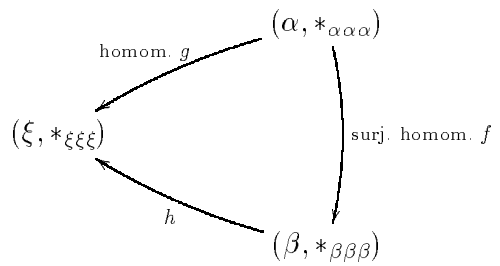
THM270.

$$\forall f_{\beta\alpha}\forall g_{\xi\alpha}\forall h_{\xi\beta}.\forall x_\alpha[h[fx] = gx]$$
$$\wedge\forall y_\beta\exists x[fx = y] \wedge \text{GEN-HOM } f \wedge \text{GEN-HOM } g \supset \text{GEN-HOM } h$$

where

$$\text{GEN-HOM} = \lambda f_{\sigma\tau}\forall x_\tau\forall y_\tau . f[*_{\tau\tau\tau}xy] = *_{\sigma\sigma\sigma}[fx].fy$$

Suppose that α, β and ξ are types representing three sets, equipped with binary functions $*_{\alpha\alpha\alpha}, *_{\beta\beta\beta}$ and $*_{\xi\xi\xi}$. If $f_{\beta\alpha}, g_{\xi\alpha}$ and $h_{\xi\beta}$ are functions such that $h \circ f = g$, f is surjective, and both f and g are homomorphisms with respect to the appropriate functions $*$, then h is also a homomorphism. This problem was suggested by Erica Melis.



The proof requires five quantifier duplications, and the unification problem for it is very difficult; furthermore, there are only two rigid literals in the search space, and hence (with one exception) every pair of literals is unifiable. This means that the proof is well beyond the scope of the path-enumeration searches in TPS. A proof can be found by component search without using the new method for equality reasoning, although this is rather difficult; using the equality reasoning, a proof is found in 0.53 seconds, of which 0.06 seconds (11%) are spent in unification.

CR-THEOREM.

$$\forall R_{o\tau\tau}.\text{REFLEXIVE } R \wedge \text{TRANSITIVE } R \wedge \text{DIAM } R \supset \forall M_\tau \forall N_\tau.\text{EQ*}\ RMN \supset \text{COMMON-REWRITE } RMN$$

where

$$\begin{aligned}
\text{REFLEXIVE} &= \lambda r_{o\alpha\alpha} \forall x_\alpha r x x \\
\text{TRANSITIVE} &= \lambda r_{o\alpha\alpha} \forall x_\alpha \forall y_\alpha \forall z_\alpha . rxy \wedge ryz \supset rxz \\
\text{DIAM} &= \lambda R_{o\tau\tau} \forall A_\tau \forall B_\tau \forall C_\tau . RAB \wedge RAC \\
&\quad \supset \text{COMMON-REWRITE } RBC \\
\text{EQ}* &= \lambda R_{o\tau\tau} \lambda x_\tau \lambda y_\tau \forall P_{o\tau\tau} . R \subseteq P \wedge \text{SYMMETRIC } P \\
&\quad \wedge \forall A_\tau \forall B_\tau [\exists w_\tau [PAw \wedge PwB] \supset PAB] \supset Pxy \\
\text{SYMMETRIC} &= \lambda r_{o\alpha\alpha} \forall x_\alpha \forall y_\alpha . rxy \supset ryx \\
\text{COMMON-REWRITE} &= \lambda R_{o\tau\tau} \lambda x_\tau \lambda y_\tau \exists D_\tau . RxD \wedge RyD
\end{aligned}$$

This is one formulation of the Church-Rosser theorem [Church and Rosser, 1936]: if R is a reflexive transitive relation satisfying the diamond property (DIAM), and M and N are related by the equivalence closure (EQ*) of R, then M and N have a common rewrite. This formulation is Theorem 3.1.12 of [Barendregt, 1984], with one minor modification: we assume R is reflexive and transitive, whereas the presentation in [Barendregt, 1984] takes the reflexive transitive closure of an arbitrary relation R (the unmodified form has not been proven by TPS, since it is non-trivial to prove that the reflexive transitive closure of R is reflexive and transitive). The variable P in the definition of EQ* requires a substitution of the form $\lambda x^1_\tau \lambda x^2_\tau$COMMON-REWRITE $p_{o\tau\tau} x^1 x^2$, which TPS discovers using the set-variable substitutions of [Bishop and Andrews, 1998]. The search space

which is constructed contains three set-variable substitutions, and TPS picks the correct one and finds a proof in 303.45 seconds, of which 9.96 seconds (3%) is unification.

THM146.

$$\forall r_{o\iota\iota}\forall x_\iota\forall y_\iota.\forall q_{o\iota\iota}[r \subseteq q \wedge \forall u_\iota\forall v_\iota\forall w_\iota[quv \wedge rvw \supset quw] \supset qxy]$$
$$\equiv \forall p.r \subseteq p \wedge \text{TRANSITIVE } p \supset pxy$$

This theorem states the equivalence of two definitions of the transitive closure of a relation r; it was suggested by Dana Scott. It requires a set-variable substitution for p of $\lambda u\lambda v\forall w[ruw \supset qxw] \supset qxv$. It is provable in 22.5 minutes, of which 5.1 minutes (23%) is unification.

THM152.

$$\forall r_{o\iota\iota}\forall x_\iota\forall y_\iota.\forall q_{o\iota}[\forall w_\iota[rxw \supset qw] \wedge \forall v_\iota\forall w[qv \wedge rvw \supset qw] \supset qy]$$
$$\equiv \forall p_{o\iota\iota}.r \subseteq p \wedge \text{TRANSITIVE } p \supset pxy$$

This is a simplified version of THM146, above; it can be proven in 3.9 minutes, of which 37 seconds (16%) is unification.

GRP-COMM2.

$$\begin{aligned}&\forall x_\iota[P_{\iota\iota\iota}e_\iota x = x]\\ &\wedge\forall y_\iota[Pye = y]\\ &\wedge\forall z_\iota[Pzz = e]\\ &\wedge\forall x\forall y\forall z[P[Pxy]z = Px.Pyz]\\ &\supset \forall a_\iota\forall b_\iota.Pab = Pba\end{aligned}$$

This theorem states that a group in which every element is self-inverse must be Abelian; it is a standard elementary exercise in most group theory textbooks (for example [Hungerford, 1974], exercise I.1.13). It is provable in 20.97 seconds.

DISTRIB-THM.

$\forall J_{\alpha\alpha\alpha}\forall M_{\alpha\alpha\alpha}.\text{LATTICE } JM \supset .\text{DISTRIBUTIVE } JM \equiv \text{DISTRIBUTIVE } MJ$

where

$$\begin{aligned}\text{LATTICE} &= \lambda J_{\alpha\alpha\alpha}\lambda M_{\alpha\alpha\alpha}.\\ &\quad \text{IDEMPOTENT } J \wedge \text{IDEMPOTENT } M\\ &\quad \wedge\text{ASSOCIATIVE } J \wedge \text{ASSOCIATIVE } M\\ &\quad \wedge\text{COMMUTATIVE } J \wedge \text{COMMUTATIVE } M\\ &\quad \wedge\text{LATTICE-LAW } JM \wedge \text{LATTICE-LAW } MJ\\ \text{IDEMPOTENT} &= \lambda P_{\alpha\alpha\alpha}\forall x_\alpha.Pxx = x\\ \text{ASSOCIATIVE} &= \lambda P_{\alpha\alpha\alpha}\forall x_\alpha\forall y_\alpha\forall z_\alpha.P[Pxy]z = Px.Pyz\\ \text{COMMUTATIVE} &= \lambda h_{\alpha\beta\beta}\forall x_\beta\forall y_\beta.hxy = hyx\\ \text{LATTICE-LAW} &= \lambda G_{\alpha\alpha\alpha}\lambda H_{\alpha\alpha\alpha}\forall x_\alpha\forall y_\alpha.G[Hxy]y = y\\ \text{DISTRIBUTIVE} &= \lambda G_{\alpha\alpha\alpha}\lambda H_{\alpha\alpha\alpha}\forall x_\alpha\forall y_\alpha\forall z_\alpha.Gx[Hyz] = H[Gxy].Gxz\end{aligned}$$

We write J and M for join and meet since the usual notation ($\vee$ and $\wedge$) seems likely to cause confusion. This theorem states that, in a lattice, J distributes over M iff M distributes over J; it is due to Schröder [Schröder, 1890] and can be found in any introductory text on lattice theory (for example [Szász, 1963], Theorem 28). It produces a very large jform, making it intractable for path-enumeration searches. TPS can find a proof in 492.05 seconds.

CD-LATTICE-THM.

$$\forall J_{\alpha\alpha\alpha}\forall M_{\alpha\alpha\alpha}\forall 1_{\alpha}\forall 0_{\alpha}.\text{CD-LATTICE } JM10 \supset \forall x_{\alpha}\forall y_{\alpha}\forall z_{\alpha}.\text{COMPL } JM10xy \wedge \text{COMPL } JM10xz \supset y = z$$

where

$$\begin{aligned}
\text{CD-LATTICE} &= \lambda J_{\alpha\alpha\alpha}\lambda M_{\alpha\alpha\alpha}\lambda 1_{\alpha}\lambda 0_{\alpha}.\text{LATTICE } JM \\
&\quad \wedge\text{DISTRIBUTIVE } MJ \wedge \text{DISTRIBUTIVE } JM \\
&\quad \wedge\text{TOP-ELT } JM1 \wedge \text{BOTTOM-ELT } JM0 \\
&\quad \wedge\text{COMPLEMENTED } JM10 \\
\text{TOP-ELT} &= \lambda J\lambda M\lambda 1_{\alpha}.\forall x_{\alpha}[M1x = x] \wedge \forall x.J1x = 1 \\
\text{BOTTOM-ELT} &= \lambda J\lambda M\lambda 0_{\alpha}.\forall x_{\alpha}[M0x = 0] \wedge \forall x.J0x = x \\
\text{COMPLEMENTED} &= \lambda J\lambda M\lambda 1_{\alpha}\lambda 0_{\alpha}\forall x_{\alpha}\exists y_{\alpha}\text{COMPL } JM10xy \\
\text{COMPL} &= \lambda J\lambda M\lambda 1_{\alpha}\lambda 0_{\alpha}\lambda x_{\alpha}\lambda y_{\alpha}.Jxy = 1 \wedge Mxy = 0
\end{aligned}$$

The definitions of LATTICE and DISTRIBUTIVE are as before. This theorem states that a complemented distributive lattice (i.e. a Boolean algebra) has a uniquely defined complement; it is also a standard theorem of lattice theory. Again, the main problem for TPS is that the search space is enormous. Component search can find a proof in 252.54 seconds.

MODULAR-THM.

$$\forall J_{\alpha\alpha\alpha}\forall M_{\alpha\alpha\alpha}.\text{LATTICE } JM \wedge [\text{DISTRIBUTIVE } JM \wedge \text{DISTRIBUTIVE } MJ] \supset \text{MODULAR } MJ$$

where

$$\text{MODULAR} = \lambda G_{\alpha\alpha\alpha}\lambda H_{\alpha\alpha\alpha}\forall x_{\alpha}\forall y_{\alpha}\forall z_{\alpha}.Hxz = z \supset Hx[Gyz] = G[Hxy]z$$

and all other definitions are as before. This states that a distributive lattice is modular (another standard theorem of lattice theory); it can be proven in 0.59 seconds.

MODULAR-EQUIV-THM.

$$\forall J_{\alpha\alpha\alpha}\forall M_{\alpha\alpha\alpha}.\text{LATTICE } JM \supset .\text{MODULAR } MJ \equiv \text{MODULAR-DEF2 } MJ$$

where

$$\text{MODULAR-DEF2} = \lambda G\lambda H\forall x_{\alpha}\forall y_{\alpha}\forall z_{\alpha}.Hx[Gy.Hxz] = H[Hxy].Hxz$$

This theorem is originally due to Jordan [Jordan, 1949] (see also [Szász, 1963], Theorem 30), and provides an alternative characterization of modularity which we shall use in subsequent theorems. TPS proves this in 411 seconds.

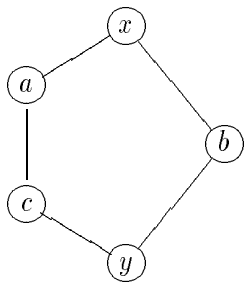
PENTAGON-THM2B.

$\forall J_{\alpha\alpha\alpha}\forall M_{\alpha\alpha\alpha}$.LATTICE $JM \supset$
.PENTAGON $MJ \supset\sim$MODULAR-DEF2 MJ

where

$$\begin{aligned}
\text{PENTAGON} \;=\; & \lambda G_{\alpha\alpha\alpha}\lambda H_{\alpha\alpha\alpha}\exists x_\alpha\exists y_\alpha\exists a_\alpha\exists b_\alpha\exists c_\alpha.\text{DISTINCT-5 } xyabc \\
& \wedge Gxy = y \wedge Hxy = x \wedge Gxa = a \wedge Hxa = x \wedge Gxb = b \\
& \wedge Hxb = x \wedge Gxc = c \wedge Hxc = x \wedge Gab = y \wedge Hab = x \\
& \wedge Gac = a \wedge Hac = c \wedge Gay = y \wedge Hay = a \wedge Gbc = y \\
& \wedge Hbc = x \wedge Gby = y \wedge Hby = b \wedge Gcy = y \wedge Hcy = c \\
\text{DISTINCT-5} \;=\; & \lambda x_\alpha\lambda y_\alpha\lambda a_\alpha\lambda b_\alpha\lambda c_\alpha.\sim[a = b]\wedge\sim[a = c]\wedge\sim[a = x] \\
& \wedge\sim[a = y]\wedge\sim[b = c]\wedge\sim[b = x]\wedge\sim[b = y]\wedge\sim[c = x] \\
& \wedge\sim[c = y]\wedge\sim.x = y
\end{aligned}$$

PENTAGON:



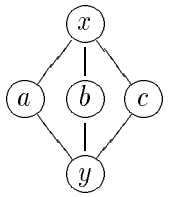
A lattice which contains a pentagon (as in the diagram above) is not modular. This is the easier direction of a theorem of Dedekind [Dedekind, 1900] (see, for example, [Szász, 1963], Theorem 32); in fact, not being modular is equivalent to containing a pentagon, but the other direction of the equivalence is much too difficult for TPS to prove automatically. DISTINCT-5 simply asserts that the five elements x, y, a, b and c are distinct. The proof takes 368.7 seconds.

3-DIAMOND-THM.

$\forall J_{\alpha\alpha\alpha}\forall M_{\alpha\alpha\alpha}$.LATTICE $JM \supset$ .3-DIAMOND $MJ \supset\sim$DISTRIBUTIVE MJ

where

$$\begin{aligned}
\text{3-DIAMOND} \;=\; & \lambda G_{\alpha\alpha\alpha}\lambda H_{\alpha\alpha\alpha}\exists x_\alpha\exists y_\alpha\exists a_\alpha\exists b_\alpha\exists c_\alpha.\text{DISTINCT-5 } xyabc \\
& \wedge Gxy = y \wedge Hxy = x \wedge Gxa = a \wedge Hxa = x \wedge Gxb = b \\
& \wedge Hxb = x \wedge Gxc = c \wedge Hxc = x \wedge Gab = y \wedge Hab = x \\
& \wedge Gac = y \wedge Hac = x \wedge Gay = y \wedge Hay = a \wedge Gbc = y \\
& \wedge Hbc = x \wedge Gby = y \wedge Hby = b \wedge Gcy = y \wedge Hcy = c
\end{aligned}$$

3-DIAMOND: 

A lattice which contains five points as in the above diagram cannot be distributive. This is part of the easier direction of a theorem due to Birkhoff [Birkhoff, 1934] (a lattice is distributive iff it contains neither a pentagon nor a copy of the above diagram). The theorem can be found in, for example, [Szász, 1963], Theorem 33. Again, the full theorem is well beyond the capabilities of TPS; this partial result can be proven in 320.39 seconds.

EQUIV-01-02, EQUIV-01-03, EQUIV-02-03.

$$\begin{aligned}
\text{EQUIV-01-02} &= \forall f_{\gamma\gamma\gamma}\forall e_{\gamma}.\text{GROUP1}_{\ o\gamma(\gamma\gamma\gamma)}fe \equiv \text{GROUP2}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
\text{EQUIV-01-03} &= \forall f_{\gamma\gamma\gamma}\forall e_{\gamma}.\text{GROUP1}_{\ o\gamma(\gamma\gamma\gamma)}fe \equiv \text{GROUP3}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
\text{EQUIV-02-03} &= \forall f_{\gamma\gamma\gamma}\forall e_{\gamma}.\text{GROUP2}_{\ o\gamma(\gamma\gamma\gamma)}fe \equiv \text{GROUP3}_{\ o\gamma(\gamma\gamma\gamma)}fe
\end{aligned}$$

where GROUP1, GROUP2 and GROUP3 are three different definitions of a group:

$$\begin{aligned}
\text{GROUP1} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}.\text{ASSOC}_{\ o(\gamma\gamma\gamma)}f \wedge \text{UNIT}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
&\quad \wedge\text{INVERSE}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
\text{GROUP2} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}.\text{ASSOC}_{\ o(\gamma\gamma\gamma)}f \wedge \text{LEFT-UNIT}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
&\quad \wedge\text{LEFT-INVERSE}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
\text{GROUP3} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}.\text{ASSOC}_{\ o(\gamma\gamma\gamma)}f \wedge \text{RIGHT-UNIT}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
&\quad \wedge\text{RIGHT-INVERSE}_{\ o\gamma(\gamma\gamma\gamma)}fe\\
\text{UNIT} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}\forall a_{\gamma}.fea = a \wedge fae = a\\
\text{LEFT-UNIT} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}\forall a_{\gamma}.fea = a\\
\text{RIGHT-UNIT} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}\forall a_{\gamma}.fae = a\\
\text{INVERSE} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}\forall a_{\gamma}\exists b_{\gamma}.fab = e \wedge fba = e\\
\text{LEFT-INVERSE} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}\forall a_{\gamma}\exists b_{\gamma}.fba = e\\
\text{RIGHT-INVERSE} &= \lambda f_{\gamma\gamma\gamma}\lambda e_{\gamma}\forall a_{\gamma}\exists b_{\gamma}.fab = e\\
\text{ASSOC} &= \lambda f_{\gamma\gamma\gamma}\forall a_{\gamma}\forall b_{\gamma}\forall c_{\gamma}.f[fab]c = fa.fbc
\end{aligned}$$

These three theorems state the equivalence of three different definitions of a group, and were suggested by Chris Benzmüller. They are only tractable when equality reasoning is used. The first two (EQUIV-01-02 and EQUIV-01-03) are obviously analogous to each other, each contain one trivial direction, and can be proven in about 27 seconds each. The last theorem (EQUIV-02-03) effectively requires all of the work of the previous two, and so takes about twice as long (55 seconds).

9.4. Comparison with Path-Based Searches

Component search seems to be comparable in speed to path-based searches on the problems which they can both solve, but is also capable of handling search spaces which require too many quantifier duplications to be tractable for the path-based searches. The equality reasoning procedure was used for a number of the new searches above and, despite the considerable shortcomings of the implementation, it clearly increases the power of the search. Higher-order unification takes a much smaller proportion of the search time than for path-based searches, at least when equality reasoning is not used. Finally, we observe that the search is not overly dependent on the initial decision about how much to amplify the original jform; all but two of our first-order examples can be proven quickly using the mode MS98-FO-MODE, and all but four of the higher-order examples can be proven using the mode MS98-HO-MODE. In other words, it is not necessary to duplicate exactly the required quantifiers, or even to duplicate all quantifiers exactly the minimal required number of times, in order to find a proof; this is obviously a desirable property in any proof search that claims to be "automatic".

It seems likely that there is a class of theorems for which component search is particularly suited, probably including theorems which contain irrelevant information (e.g. proofs from a set of axioms that do not use all the axioms), theorems which require significant use of higher-order unification, and theorems which break naturally into fragments. A good example of a theorem which does *not* fall into any of these categories, and for which component search is not at all suited, is X2129 (this is Andrews' Challenge Problem, from the 4th CADE; see [Quaife, 1990; Issar, 1991] for descriptions of some automated proofs of this problem). This is a first-order problem, and it is unusual in that every fragment contains a single literal, the vertical paths are much shorter than the horizontal paths and pass through comparatively few quantifiers, and the majority of the quantifiers have a single literal as their scope. Even that brief description suggests that path-focused duplication is more likely to find a proof than component search, and this is in fact the case.

CHAPTER 10

Summary and Future Work

10.1. Summary

We have introduced the component search algorithm, which addresses some of the deficiencies of path-enumeration searches. The algorithm makes tractable a number of previously intractable problems, and is comparable to path-enumeration on those problems which can be solved by both algorithms. Component search is particularly suited to higher-order problems, since it allows for the use of unification graphs to represent the substitutions for partial matings. Unification graphs are based on the method of binary decision diagrams, and can be efficiently manipulated to generate the intersection of two sets of unifiers, as required by the search algorithm.

10.2. Future Work

A number of possible avenues for future research were suggested in previous chapters (Chapter 8 in particular). We summarize these issues and a few others here.

Control Structures, I: It would be worth investigating improved methods for pruning the search space and eliminating redundancies. In particular, component search only uses particular cases of Theorem 6.1.10, for want of an efficient implementation; there may also be other redundant matings which could be eliminated from the search space. One might try to identify situations in which some of the reduction strategies of Section 2.6 could be applied efficiently, or to devise other reduction strategies which apply to component search in particular.

Control Structures, II: Component search has two built-in methods for directing the search: weightings and fragment orderings. Characterizing good methods for weighting and for choosing the next fragment is an interesting problem (getting TPS to do so itself, and learn from its experience, is even more interesting).

Minimality: The current test for non-minimal matings (see Section 3.6) is too expensive to be practical, especially since comparatively few non-minimal matings are constructed in a typical search. A better understanding of the situations in which non-minimal matings are constructed might allow the test to be restricted only to cases in which it is likely to be useful.

Preserving Information: The search algorithm, although it preserves much more information than previous algorithms, still loses a significant amount of information when it returns to amplify the jform. Some of this information could be preserved and reused on the next pass through the procedure (for example, minimal components will still be minimal in the amplified jform). Whether one can preserve useful information about unification will depend largely on whether the bounds on unification are relaxed on the next pass thorough the procedure; see Chapter 5 for details.

Dynamic Reordering: Given the effectiveness of this technique in implementations of binary decision diagrams, it would almost certainly be worth implementing it for unification graphs.

Equality: The inadequacies of our equality algorithm were discussed in Chapter 7; the existing method should be regarded more as a proof of concept than as a usable procedure. It is not necessary to reinvent the wheel in order to get a better equality algorithm; given that TPS is one of many theorem-proving systems which are now integrated with ΩMEGA, it should be possible to arrange for TPS to call a system which specializes in equality reasoning by communicating through ΩMEGA. The search space for rewrite proofs is still, in general, larger than one would like, and it would probably be useful to investigate further the ideas of parity checking and discovering interpretations, as discussed in Section 7.5.

Integration: The above remarks about equality reasoning also apply more generally; is it possible for TPS to cooperate usefully with other theorem-proving systems (or proof-planners, or computer algebra packages) in the same manner?

Parallelism: A parallel component search algorithm was sketched as Algorithm 8.1.1; if a suitable method can be devised for the processors to communicate, it would be worth investigating this further.

APPENDIX A

Flags

In TPS, the user-definable settings are called *flags*. There are many flags in TPS (403 at the time of writing), many of which affect the behavior of automated proof search. However, only a few of these flags are actually important parameters of the search; the vast majority have little or no effect in most cases. The latter are often flags which were introduced to experiment with different approaches to a problem, and which subsequently proved to have one setting which was overwhelmingly better than all of the other possibilities.

This appendix lists and describes all of the flags which are used by the implementation of component search in TPS, which is called MS98-1. It is primarily intended as a reference for users of TPS, but also documents some of the less-fruitful experiments into the component search procedure. The flags occur in two subjects, `MS98` and `MS98-MINOR`.

BREAK-AT-QUANTIFIERS type: BOOLEAN, default: NIL

If T, then fragments will be broken so as not to contain any quantifiers (which is condition 3 of the definition of $\sim_J$ given in Definition 3.1.2). If NIL, fragments may contain quantifiers.

DEFAULT-MS type: SEARCHTYPE, default: MS90-3

The default mating search procedure. To use component search, this flag must be set to MS98-1.

FF-DELAY type: BOOLEAN, default: NIL

If T, delay unifying flex-flex pairs for single connections, and unify them in context when some flex-rigid pairs are added. If NIL, unify them as usual. This flag was introduced for experimenting with the unification method, and should generally be left set to NIL.

FIRST-ORDER-MODE-MS type: BOOLEAN, default: NIL

If T, then first-order unification is called during mating search, else higher-order unification is used. TPS will set this flag automatically when `DIY` is called, but not when `GO` is called. The automatic setting can be overridden (to be always NIL) in component search by setting the flag MS98-FORCE-H-O.

HPATH-THRESHOLD type: NULL-OR-POSINTEGER, default: 1

If NIL, break the jform into fragments corresponding to the top-level conjunctions in the search space. If n, break at conjunctions and also on disjunctions having more than n horizontal paths. The standard definition of $\sim_J$ given in Definition 3.1.2 is given by setting HPATH-THRESHOLD to 1 and BREAK-AT-QUANTIFIERS to T.

MAX-MATES type: POSINTEGER-OR-INFINITY, default: 2

The maximum number of mates which each literal is allowed to have. Set MAX-MATES to INFINITY to allow an unlimited number of mates for any literal. This is one of the most important flags, particularly for large search spaces.

MAX-SUBSTS-VAR type: NULL-OR-INTEGER, default: NIL

This is the depth bound on unification mentioned in Chapter 5; it applies to higher-order problems only. When MAX-SUBSTS-VAR is n, each variable is allowed ground substitutions constructed from at most n steps of imitation and projection (see [Huet, 1975; Snyder and Gallier, 1989]). Substitutions are not constructed in long $\beta\eta$-normal form, so (for example) the substitution $p_{o\alpha\alpha} \mapsto R_{o\alpha\alpha}$ is considered to have depth 1, rather than 3 (it requires one imitation and two projections to construct $p_{o\alpha\alpha} \mapsto \lambda x_\alpha \lambda y_\alpha R_{o\alpha\alpha} x y$).

Obviously, the user should choose a value for this flag that allows all of the required substitutions to be found, but too large a value may make the search space intractable. This is one of the most important flags for the search.

MAXIMIZE-FIRST type: BOOLEAN, default: NIL

When this flag is set to T, then each time a component is being extended (at step 5 of Algorithm 4.4.1), do not create any new components which exceed MAX-MATES 1 until there are no other ways to extend the component. The components which have MAX-MATES > 1 are stored for later consideration, in order to preserve completeness. This has only been implemented for higher-order problems, and generally seems to be of very little use; you are advised to leave this flag set to NIL.

MERGE-MINIMIZE-MATING type: BOOLEAN, default: T

This flag is used during merging, after a complete mating has been found. If it is set to T, merging will attempt to minimize the mating by removing any unnecessary connections. Otherwise, it won't. T is the default setting for all other mating searches, and will sometimes produce a more readable ND proof, but for large jforms this minimization can take a very long time (often much longer than the search itself). For component search, especially on large search spaces, you should set this flag to NIL.

MIN-QUANTIFIER-SCOPE type: BOOLEAN, default: NIL

When this flag is T, the scope of quantifiers is minimized before starting expansion proofs. If an eproof is found with this flag set to T, during the translation of the eproof to an ND proof we use `ruleq` (see [Andrews, 1986]) to fill the gap between the theorem as originally stated and its min-quantifier-scope version. This flag is common to all mating searches in TPS; it is not generally important in component search except for those search spaces which are so large that absolutely every flag must be set to its optimal value if a proof is to be found.

MS98-BASE-PRIM type: BOOLEAN, default: NIL

If T, we allow the search to begin with a fragment which lies below a set-variable substitution. If NIL, we always choose a fragment which is outside

the set-variable substitutions (if possible). In general, this flag should be set to NIL, for reasons outlined in Section 4.5.

MS98-DUP-BELOW-PRIMSUBS type: BOOLEAN, default: NIL

When T, duplicate the quantifiers which occur below a set-variable substitution NUM-OF-DUPS times. When NIL, don't duplicate these quantifiers. There are some problems which can only be solved by setting this flag to T, but in general the default setting is adequate.

MS98-DUP-PRIMSUBS type: BOOLEAN, default: NIL

When T, the command `ms98-dup` (which is called when MS98-INIT is set appropriately) duplicates variables which have set-variable substitutions; when NIL, it doesn't. (Note that duplicating the variable will not duplicate the primsub; it will produce another copy of the unsubstituted-for tree below that expansion node.) Again, the default setting is generally adequate for most purposes.

MS98-FIRST-FRAGMENT type: SYMBOL-OR-INTEGER, default: NIL

If non-NIL, this will move a single fragment to the beginning of the literal ordering, as follows:

T: Set of support strategy, more or less. The starting fragment will be the last non-duplicate fragment enumerated. This will be the rightmost part of the wff to be proven.

n: (for integer *n*) the starting fragment will be whichever fragment contains LEAFn. If this leaf is part of a duplicate fragment, or does not exist at all, then this will behave like T.

This flag overrides MS98-BASE-PRIM; the chosen fragment may always be part of a primitive substitution. See also MS98-FRAGMENT-ORDER.

MS98-FORCE-H-O type: BOOLEAN, default: NIL

If T, use higher-order unification graphs even for first-order searches. If NIL, use the normal first-order unification. This flag was introduced for experiments with using unification graphs for first-order problems. For reasons outlined at the beginning of Chapter 5, it should be set to NIL.

MS98-FRAGMENT-ORDER type: INTEGER+, default: 1

The order in which the fragments are considered. This principally affects which fragment will become the starting point of the search, and which of the weak fragments will be blocked next. See also MS98-FIRST-FRAGMENT.

0: consider the number of ways to block the given fragment.

1: consider the number of ways that the results for 0 might be extended (i.e. look ahead two steps in the search process).

2: as for 1, but then weight in favour of ground fragments.

The general considerations for the choice of fragment order are outlined in Section 4.5.

MS98-INIT type: INTEGER+, default: 0

Before doing MS98-1 search, do the following:

0: do nothing at first; after each failure, duplicate one more quantifier.

1: duplicate all outer quantifiers NUM-OF-DUPS times.

2: apply substitutions for set variables and duplicate all outermost quantifiers NUM-OF-DUPS times.

3: cycle through substitutions for set variables one at a time, and duplicate all outermost quantifiers NUM-OF-DUPS times. The time spent on each such substitution will be at least MAX-SEARCH-LIMIT seconds, unless the search fails before then.

This flag absolutely must be set correctly in order to find a proof (except for very small problems). Guidelines for setting this are: use 3 if the problem may require a large set-variable substitution (say, containing more than one connective or quantifier), use 2 if it may require a small set-variable substitution, use 1 if quantifier duplications alone are likely to be enough, and use 0 only for first-order problems, and preferably not even then.

If duplications or substitutions for set variables are introduced, then the other flags which control them must also be set. Check the subject `PRIMSUBS` in TPS, as well as the other flags listed here.

MS98-LOW-MEMORY type: BOOLEAN, default: NIL

If T, try to keep memory use low (principally by finding and erasing any irrelevant information in the hash tables used by the unification graphs of Chapter 5). This will probably make the search take longer, and so this flag should only be set to T if there seems to be a danger of running out of space.

MS98-MAX-COMPONENTS type: NULL-OR-POSINTEGER, default: NIL

If non-NIL, the maximum number of components that can be considered on any iteration of the search. For example, if we reach step 5 of Algorithm 4.4.1 with a *current* list of a thousand components, but MS98-MAX-COMPONENTS is set to 50, then we will put the last 950 elements of *current* onto *pending* and just consider the first fifty elements.

This was an experimental and somewhat arbitrary way of preventing the search from branching too much; it is generally useless, and should be ignored.

MS98-MAX-PRIMS type: NULL-OR-POSINTEGER, default: 1

The maximum number of substitutions for set variables allowed in any component (analogous to MS98-NUM-OF-DUPS). For most of the problems that TPS can solve, the default value is adequate.

MS98-MEASURE type: INTEGER+, default: 0

Determines the measure which is used on a component C, as follows:

0: count the fragments blocked and then weight heavily against the situation described by MS98-VALID-PAIR.

1: weight in favour of large components in which few fragments are weak.

2: as for 1 but also take account of the number of duplications which are touched.

3: just count the number of fragments blocked by C.

4: as for 2 but also count the number of matings for the smallest fragment which is weak in C.

5: multiply the number of matings for the smallest fragment which is weak in C by the number of substitutions which unify C.

6: use the ratio of blocked to touched fragments in C and the ratio of the number of blocked fragments to the number of connections.
7: prefer matings where positive leaves are mated to negative leaves and vice versa.
8: use the ratio of blocked to touched fragments.
9: favour large components satisfying max-mates 1.
10: do as for 9 and then weight heavily against the situation described by MS98-VALID-PAIR.
11: do as for 6 and then weight heavily against the situation described by MS98-VALID-PAIR.
12: do as for 8 and then weight heavily against the situation described by MS98-VALID-PAIR.
13: weight in favour of components with max-mates 1 and then weight heavily against the situation described by MS98-VALID-PAIR.
14: do as for 7 and then weight heavily against the situation described by MS98-VALID-PAIR.
15: take the average of 11 and 14.
other: return a constant.

These are the various weightings available for Algorithm 4.4.1. The case labelled "other" implements Algorithm 4.1.1, as noted after the definition of Algorithm 4.4.1. The settings are in no particular order, since different weightings have been implemented throughout the development of this procedure. The two settings 0 and 3, which are by far the simplest, also seem to be the most effective.

MS98-MERGE-DAGS type: INTEGER+, default: 0

For higher-order searches only. Affects the way in which the unification graphs of minimal components are computed from the graphs of their constituent connections. Computing the unification graph of every possible minimal component is time-consuming and unnecessary, since many of these graphs will never be used again, and so we merely attempt to filter out some of the non-unifiable minimal components in advance. There are three ways to do this, arranged weakest first:

0: Check that the graphs of the connections are pairwise compatible. Only compute the full graph of a component when necessary.
1: Check that the graphs of the connections are compatible taken all together. Only compute the full graph when necessary.
2: Always compute the full graph. This overrides FF-DELAY.

0 will allow some non-unifiable components into the main search algorithm (they will be removed as soon as the algorithm attempts to use them, of course). 1 does not allow this, but does not actually compute and store the unification graph for the component until it is necessary to do so. 2 explicitly computes all of the unification graphs in advance of the search. This flag should be set to 0 (which almost always results in a much quicker search than the other options), unless you have some good reason to do otherwise.

MS98-MINIMALITY-CHECK type: BOOLEAN, default: NIL

If T, check each new component for minimality and reject those which are non-minimal. If NIL, then skip the minimality check. See Section 3.6 for details.

MS98-NUM-OF-DUPS type: NULL-OR-POSINTEGER, default: NIL

If MS98-NUM-OF-DUPS is NIL, each component can use every quantifier duplication in the search space. If MS98-NUM-OF-DUPS is n, each component may use at most n of the quantifier duplications. It makes no sense for this flag to be smaller than NUM-OF-DUPS. (Example: Suppose the search space has five outermost quantifiers, NUM-OF-DUPS is 2 and MS98-NUM-OF-DUPS is 3. Each quantifier will be duplicated twice, producing ten quantifier duplications in all. Of these ten duplications, at most three may be used by any one component.)

Choosing a good value for n is very important in a large search space with many duplicated quantifiers; for smaller searches, NIL is a good setting for this flag.

MS98-PRIMSUB-COUNT type: NULL-OR-POSINTEGER, default: 3

The maximum number of primsubs to be applied to each set variable in the expansion tree. This is only used when MS98-INIT is 2, and choosing a value much larger than 3 will usually make the search space too large to be tractable.

MS98-REWRITE-DEPTH type: POSINTEGER, default: 2

When attempting to rewrite one term into another using the equality rules (see Chapter 7), this flag determines the maximum number of steps of rewriting that are allowed. This is a very important flag for searches that require equality rules, and utterly irrelevant elsewhere.

MS98-REW-PRIMSUBS type: BOOLEAN, default: NIL

When T, MS98-INIT does set-variable substitutions for Leibniz variables which are contained within equality rewrites; when NIL, it doesn't. The default setting is generally adequate.

MS98-REWRITE-MODEL type: BOOLEAN, default: NIL

If T, then TPS will automatically attempt to generate an interpretation for the current rewrite rules, and will use this to slim down the unification tree. Generally, setting this flag to T in the presence of rewrite rules will improve the performance of the search; however, it may also make the search incomplete and hence fail to find the proof. See the discussion in Section 7.5 for more details.

MS98-REWRITE-PRUNE type: BOOLEAN, default: T

If T, delete any substitutions which are duplicates modulo rewriting (this can be slow). If NIL, don't. Reducing the number of unifiers in this way can speed up the unification procedures a little (by reducing the size of unification diagrams), although it can also increase the amount of rewriting which is necessary to find a proof. The default setting for this flag is generally reasonable.

MS98-REWRITE-SIZE type: NULL-OR-POSINTEGER, default: NIL

The maximum size of a (lambda-normalized) term that can be produced by rewriting, measured as the number of nodes in the parse tree of that term. NIL means that there is no maximum. Setting an appropriate maximum size can greatly increase the efficiency of the equality reasoning procedure, particularly if MS98-REWRITE-DEPTH is large. This bound applies to all of the intermediate terms in a rewrite proof of $A = B$, not only to the terms A and B.

In general, when rewrites are present, setting *some* non-NIL value for this flag is a good idea, but it needn't be exactly the minimum possible value.

MS98-REWRITE-UNIF type: NULL-OR-POSINTEGER, default: NIL

When a rewrite rule can introduce a new variable (for example, $x * x^{-1} = 1$, when applied from right to left), this flag governs the size of the allowed substitutions for that variable. Essentially, this is a special case of MAX-SUBSTS-VAR, and the same considerations apply to setting this flag as apply to MAX-SUBSTS-VAR.

MS98-REWRITES type: BOOLEAN, default: NIL

When T, make all of the global equalities into rewrites. This is the flag which switches on the equality reasoning. If this is set to NIL, all of the rewriting flags above are irrelevant, and no equality reasoning is used. See Chapter 7 for a discussion of the equality reasoning in TPS.

MS98-UNIF-HACK type: BOOLEAN, default: NIL

If T, do not introduce new constants during unification. This really is a hack, since in general we do need to introduce new constants ($\top$, $\bot$ and possibly some extra constants of depth zero to unify flex-flex pairs, if there are no existing constants of the appropriate type). In many problems, constants already exist at most types, and the only new constants which will be introduced are $\top$ and $\bot$; these are irrelevant to most problems, and only serve to increase the number of substitutions. The default value, NIL, is required for completeness, but the value T will speed up most searches.

MS98-UNIF-HACK2 type: BOOLEAN, default: NIL

If NIL, generate unifiers as normal. If T, then when unifiers are generated, prevent the occurrence of ground subformulas of type o (except for $\top$ and $\bot$). This is justifiable because, in our logic, $\top$ and $\bot$ are the only distinct objects of type o, so any ground expression of type o is equivalent to one or the other. However, this equivalence has to be proven, and the search space may not contain the appropriate elements for a proof; accordingly, this flag is considered a hack.

Again, the default value NIL is required for completeness, but the value T will speed up most searches.

MS98-VALID-PAIR type: INTEGER+, default: 1

This flag was implemented mostly for experimental purposes.

Given two disjuncts $X \vee Y$ and $A \vee B$, this flag determines when we are allowed to make a component containing connections X–A and Y–B (assuming they're unifiable connections). If MAX-MATES is 1, these connections cannot be correct (the path through X and A which is spanned by X–A has

a variant through X and B which must be spanned, and since MAX-MATES is 1 this spanning connection cannot involve X or B, whence it also spans the first path). For MAX-MATES > 1, similar reasoning shows that each of X, Y, A and B must have an additional mate, and all of these connections must be mutually unifiable.

The higher the number, the more stringent the test; any correct mating is guaranteed to pass any of these tests (except for the first, labelled 0):

0: reject any such connections even if max-mates is not 1. This results in an incomplete search, but is sometimes usable anyway.
1: If MAX-MATES is 1, reject all such pairs of connections (this is justified by the argument above). Otherwise accept them.
2: As for 1, plus we require an extra mate for each of X, Y, A and B.
3: As for 2, plus we require that all of these new mates be pairwise compatible with each other.
4: As for 3, plus we require that all of these new mates be simultaneously compatible with each other.

Options 3 and 4 are only implemented for higher-order searches, and options 2, 3 and 4 are computationally expensive. Since option 0 is incomplete, we recommend using the default value of 1 for this flag.

MS98-VARIABLE-ORDER type: INTEGER+, default: 1

Determines the variable ordering for the unification graph. Only affects higher-order searches. Suppose N is the maximum number of unifiers for a given list of variables, and K is the length of the list. For values 0–3, the variables are first grouped into lists of duplicate copies (so each variable is listed with its duplicates, if any)

0: Sort by N, largest first.
1: Sort by N, smallest first.
2: Sort by K, largest first.
3: Sort by K, smallest first.
10–13: Group the variables into lists of length 1, and then proceed as 0–3.
20–23: Group the variables into lists that occur together (i.e. two variables go into the same list if their expansion nodes are not separated by any junctive node in the etree) and then proceed as for 0–3.
30–33: Group the variables as for 0–3, and then reduce the lists to length 1 while keeping the variables in the same order.
40–43: Group the variables as for 20–23, and then reduce the lists to length 1 while keeping the variables in the same order.
100–143: As for 0–43, but find all of the variables whose (universal) quantifier occurs above some selection node, and move the groups containing those variables to the head of the variable ordering.

The setting of this flag is very important in some searches, and almost irrelevant in others. Efficiency in computing the ordering is not really an issue (the variables only need to be ordered once), but the ordering will

determine the size of the unification graphs, and hence the efficiency of the unification procedure. The default value is reasonable for most purposes; for very large searches, one should experiment with different values to find the one that makes unification run fastest.

MS98-VERBOSE type: BOOLEAN, default: NIL

If T, print extra information during the search. This will print so much information that it will slow down the search, and is really only useful for debugging or demonstration purposes; leave this flag set to NIL most of the time.

NUM-OF-DUPS type: INTEGER+, default: 2

This flag is used by MS98-INIT to determine how many times each outermost quantifier should be duplicated. It may be set to 0 (if, for example, you want substitutions for set variables but no duplications). It is used (with a slightly different meaning) by most of the other searches in TPS, and is one of the most important flags in TPS. There is an optimal value — exactly the right number of duplicates for a proof, and no more — but component search will generally also succeed with values that are a little larger than optimal, except for very large searches. MS98-NUM-OF-DUPS provides a related bound on duplication. It is essential to choose a good value for this flag.

REWRITE-DEFNS type: REWRITE-DEFNS-LIST, default: (EAGER)

A list whose first element is one of NONE, EAGER, LAZY1 and DUAL, and whose other (optional) elements are lists whose first element is one of these four options and whose other elements are the names of definitions. The first element is the default behavior for rewriting definitions, and the other lists are lists of exceptions to this default, with a different behavior specified.

NONE: do not rewrite this definition at all.
EAGER: rewrite all of these definitions, in one step, as soon as possible.
LAZY1: rewrite these, one step at a time, when there are no more EAGER rewrites to do.
DUAL: use dual instantiation [Bishop and Andrews, 1998].
LAZY2: synonym for DUAL.

For example: the value (EAGER) would be interpreted as "Rewrite every definition in one step", whereas (DUAL (EAGER TRANSITIVE) (NONE INJECTIVE SURJECTIVE)) would be interpreted as "Rewrite TRANSITIVE whenever it appears. Don't ever rewrite INJECTIVE or SURJECTIVE. Rewrite every other definition using dual instantiation."

The default setting is usually appropriate for component search.

REWRITE-EQUALITIES type: REWRITE-DEFNS, default: ALL

One of the following:

NONE: do not rewrite equalities
ONLY-EXT: rewrite only those equalities that can be rewritten using extensionality.
LEIBNIZ: rewrite all equalities using the Leibniz definition.
ALL: rewrite all equalities:

- to an equivalence for those of type ooo
- to the extensional form $[\lambda f_{\alpha\beta}\lambda g_{\alpha\beta}\forall x_\beta fx = gx]$ for those of type $o(\alpha\beta)(\alpha\beta)$
- to the Leibniz form $[\lambda x_\alpha\lambda y_\alpha\forall q_{o\alpha}.qx \supset qy]$ for those of type $o\alpha\alpha$ which are not rewritable by either of the above methods.

LAZY2**:** As for ALL, but using dual instantiation [Bishop and Andrews, 1998].

The default setting is usually appropriate for component search.

REWRITE-EQUIVS type: POSINTEGER, default: 1

This chooses one of the two ways of constructing an etree from an equivalence $A \equiv B$:

1: chooses the option with the fewest vertical paths (positive occurrences become $[A \wedge B] \vee [\sim A \wedge \sim B]$, negative occurrences become $[A \supset B] \wedge [B \supset A]$)

2: chooses the option with the fewest horizontal paths (negative occurrences: $[A \wedge B] \vee [\sim A \wedge \sim B]$, positive occurrences: $[A \supset B] \wedge [B \supset A]$)

3: behaves as for 2 except for the first equivalence it finds, when it behaves as for 1. (This means that a wff which is a quantified equivalence will produce an etree which can be split.)

4: always chooses $[A \supset B] \wedge [B \supset A]$

5: always chooses $[A \wedge B] \vee [\sim A \wedge \sim B]$

Any other setting will behave like 1.

This does not work with MIN-QUANTIFIER-SCOPE T; in that case, etrees will be constructed as in case 1, regardless of the setting of this flag.

This flag was defined for an experiment into alternative definitions of the $\sim_J$ relation which defines the fragments. The default behavior (1) rewrites $A \equiv B$ into four fragments, each containing a single literal, whereas (2) and (3) rewrite it into two fragments of two literals each. It was hoped that these other settings would improve the performance of component search, but in fact they seem to have very little effect. The flag should probably be left set to the default value.

SKOLEM-DEFAULT type: SYMBOL, default: SK1

Default method for skolemizing, in which wffs of the form $\exists y.M$ are replaced by $M(g(...))$. There are three possible ways to do this:

SK1: is the original method due to Skolem, where the Skolem constants g take as arguments all the x such that $\forall x$ occurs in the wff and $\exists y.M$ is in its scope.

SK3: is the method in which the arguments of g are the free variables of $\exists y.M$.

NIL: means don't Skolemize at all; use selection nodes instead.

SK1 is the default used by all other search procedures, but it can be disastrous for component search, especially in the presence of set-variable substitutions, because the Skolem terms produced can become extremely

large, requiring correspondingly large bounds on unification. SK3 is slightly better, but has essentially the same problem.

NIL is the preferred setting for component search (at least when unification graphs are being used); this does have the unfortunate effect of increasing the number of constant symbols in the search space (and hence the number of unifiers), but it is still preferable to the alternative, which is to increase the unification bounds.

APPENDIX B

Structures

This appendix describes the Lisp structures used in the implementation of component search in TPS. It is provided mostly as a convenience to anyone attempting to read the Lisp code for MS98-1, although it does also illustrate many of the implementation considerations discussed in previous chapters.

B.1. Components

The Lisp structure corresponding to a component is defined by `defstruct component`; it has the following slots:

clist: The actual mating, stored as a list of pairs of literals.

key: An integer key generated from the mating (each connection is assigned a unique prime number; the key is the product of the prime numbers associated with the mating).

litkey: An integer key generated from the literals (as above, but each literal is assigned a prime number).

lits: A list of the literals which occur in the mating. This is easily computed from the mating, but is used so often that it's worth storing.

blocks: A list of the fragments which are blocked by the mating.

touches: A list of the fragments which are touched by the mating.

name: A unique symbol which identifies this component.

measure: The weight of this component.

tried: A list of the names of components which we have already attempted to merge with this component.

original: A list of the fragments which we have attempted to block, in the cases when the completeness checker presents us with a list of fragments *at least one of which* must be blocked.

openpath: The result of calling the completeness checker, where appropriate. NIL otherwise. If non-NIL, it contains either a list of fragments at least one of which must be blocked, or an open path through the jform. In the latter case, the open path is presented as a cons of two lists corresponding to the two paths of Lemma 3.5.12.

final-subs: The unifying substitutions for this mating. In the first-order case, it will generally be a list of substitutions, and in the higher-order case it will generally be a unification graph. There are two special values: `fail` denotes no acceptable unifiers (components with this value will be deleted), and `delay` denotes that the unifiers have yet

to be computed (but have usually passed some preliminary test such as pairwise compatibility).

ddf: T if this component does duplications first (i.e. uses duplicate copies of a fragment before their originals); NIL otherwise.

B.2. Unification

The Lisp structure corresponding to a node in a unification graph is defined by `defstruct dnode`; it has the following slots:

sons: A list of dotted pairs of the form `(label . node)`, where `node` is either a dnode or the symbol NIL (which is used to represent the node ACCEPT). The `label` will be the label of the arc leading from this node to the son `node`, represented as a list of nonnegative integers.

keys: A list of three elements `(A B C)`, where `A` is a unique integer name for this node, `B` is the number of paths below this node (the sum of the number of paths below the sons of this node) and `C` is the number of branches below this node (the length of `sons` plus the sum of the number of branches below each of them).

merge: An empty slot, used during the phase in which unused unification graphs are removed from the array. In this phase, all current graphs are recursively tagged with a marker in this slot, and then all graphs in the hashtable which do not bear the marker are deleted.

The elements of `keys` are provided to speed up the check for isomorphism. Since *Collapse* (Algorithm 5.2.8) is only called by *Merge* (Algorithm 5.2.7) when one of the nodes being merged has both zero and nonzero arcs, we also use the keys to signal this: the name `A` is a negative integer if this situation obtains, and positive otherwise.

There is one other structure, a `qnode`, which is used during the initial generation of the substitutions. This is merely a node of a Huet-style unification tree (compare the previously-existing structures `node` and `unode` from the path-enumeration procedures), defined to contain as few slots as possible. We do not describe it further here.

B.3. Equality Reasoning

The main structure for the equality reasoning is the rewrite tree, represented by the structure `rtree`; it has the following slots:

rule: The name of the rewrite rule by which this node was obtained from its parent.

right: NIL if the rule was applied from left to right; T otherwise.

gwff: The gwff represented by this node.

code: An integer key, generated from the symbols contained in this gwff, used as a preliminary filter for testing equality of wffs. Having the same code is necessary but not sufficient for equality of gwffs.

parent: The parent of this node (or NIL if this is a root node).

sons: The sons of this node (a possibly empty list of nodes).

To test a pair of gwffs for equality modulo rewriting, we construct a rewrite tree for each of them. We also construct a discrimination tree for each of them; this will contain all of the gwffs in the rewrite tree. To determine whether we have a rewrite proof, each time we generate a node in one rewrite tree, we check for the presence of the new gwff in the other discrimination tree.

Discrimination trees are discussed in more detail in [McCune, 1992]; they are designed to store a set of wffs as a tree in which each node represents a symbol, and a path from the root to a leaf represents the depth-first enumeration of the parse tree of a term. They are by no means the only method of term indexing, nor even the best method — see [Graf, 1995] for a thorough discussion of many other term-indexing methods — but for our purposes they are acceptable since they are simple and deterministic, and the operation of inserting a new term is very fast by comparison to other methods.

The discrimination trees are represented by the structure `dtree`, which has the following slots:

symbol: The symbol represented by this node (a constant, a variable, a lambda, etc ...).

rewrites: For a non-leaf, this is ignored. For a leaf, the possible rewrites of the gwff represented by the path from the root to this leaf. This is represented as a (possibly empty) list of rewrite nodes; if the rewrites of this gwff have not been computed yet, this value will be set to T.

refers: For a leaf (representing a gwff), this is a pointer to the corresponding node of the rewrite tree.

sons: The children of this node, or NIL.

APPENDIX C

Example

This appendix shows the output of one run of MS98-1 on the jform in Figure 3.2 (which is a version of THM126 in which the quantifier duplications have already been done). The full output would take about thirty pages; some of the details have been edited out. Major omissions are marked `[...]`.

```
<Mate46>ms98-1
```

We start by invoking MS98-1 from the MATE top level; it begins by computing the unifiers down to the given unification depth.

```
1 --> LAMBDA w^1(B).~S^2(OB) w^1
2 --> S^2(OB)
3 --> LAMBDA w^3(B) S^3(OA).H^2(AB) w^3
4 --> LAMBDA w^5(B) Q(OA).H^2(AB) w^5
There are 4 unifiers for each of (q^0<OB>)

5 --> LAMBDA w^27(A).~S^3(OA) w^27
6 --> LAMBDA w^27(A).~Q(OA) w^27
7 --> S^3(OA)
8 --> Q(OA)
There are 4 unifiers for each of (q^0<OA>)

9 --> X^0(G)
10 --> Y^0(G)
11 --> X^1(G)
12 --> F^1(GGG) X^0(G) X^0
13 --> F^1(GGG) Y^0(G) X^0(G)
14 --> F^1(GGG) X^1(G) X^0(G)
15 --> F^1(GGG) X^2(G) X^0(G)
16 --> F^1(GGG) Y^1(G) X^0(G)
17 --> F^1(GGG) X^0(G) Y^0(G)
18 --> F^1(GGG) Y^0(G) Y^0
19 --> F^1(GGG) X^1(G) Y^0(G)
20 --> F^1(GGG) X^2(G) Y^0(G)
21 --> F^1(GGG) Y^1(G) Y^0(G)
22 --> F^1(GGG) X^0(G) X^1(G)
23 --> F^1(GGG) Y^0(G) X^1(G)
24 --> F^1(GGG) X^1(G) X^1
25 --> F^1(GGG) X^2(G) X^1(G)
26 --> F^1(GGG) Y^1(G) X^1(G)
27 --> F^1(GGG) X^0(G) X^2(G)
28 --> F^1(GGG) Y^0(G) X^2(G)
29 --> F^1(GGG) X^1(G) X^2(G)
30 --> F^1(GGG) X^2(G) X^2
31 --> F^1(GGG) Y^1(G) X^2(G)
32 --> F^1(GGG) X^0(G) Y^1(G)
33 --> F^1(GGG) Y^0(G) Y^1(G)
34 --> F^1(GGG) X^1(G) Y^1(G)
35 --> F^1(GGG) X^2(G) Y^1(G)
36 --> F^1(GGG) Y^1(G) Y^1
37 --> X^2(G)
38 --> Y^1(G)
There are 30 unifiers for each of (x^10<G> y^6<G> x^11<G> x^12<G> y^7<G> x^17<G> x^18<G>)

39 --> a(B)
40 --> F^2(BBB) a(B) a
41 --> H^1(BG) X^0(G)
42 --> H^1(BG) Y^0(G)
43 --> H^1(BG) X^1(G)
```

```
44 --> H^1(BG) X^2(G)
45 --> H^1(BG) Y^1(G)
There are 7 unifiers for each of (x^5<B> y^4<B> x^9<B> y^7<B> x^10<B> x^11<B> y^8<B>)
```

Note the introduction of a new constant a_β, which is necessary since the jform doesn't contain any constants of that type.

```
46 --> X^0(A)
47 --> Y^0(A)
48 --> F3^0(AAA) X^0(A) X^0
49 --> F3^0(AAA) Y^0(A) X^0(A)
50 --> F3^0(AAA) X^0(A) Y^0(A)
51 --> F3^0(AAA) Y^0(A) Y^0
52 --> H^2(AB) a(B)
53 --> H^2(AB).H^1(BG) X^0(G)
54 --> H^2(AB).H^1(BG) Y^0(G)
55 --> H^2(AB).H^1(BG) X^1(G)
56 --> H^2(AB).H^1(BG) X^2(G)
57 --> H^2(AB).H^1(BG) Y^1(G)
There are 12 unifiers for each of (x^3<A> y^3<A>)

Variable ordering is:
((q^0<0A>) (q^0<0B>) (y^8<B>) (x^11<B>) (x^10<B>) (y^3<A>) (x^3<A>) (y^7<G>) (x^12<G>) (y^6<G>)
(x^10<G>) (y^4<B> y^7<B>) (x^5<B> x^9<B>) (x^11<G> x^17<G> x^18<G>))
```

Variables have been grouped by duplication, and then ordered from least to most potential unifiers. Now we check each of the connections for unifiability.

```
Checking 780 potential connections.
LEAF109 LEAF108 is new (fail)
LEAF109 LEAF105 is new (fail)
LEAF109 LEAF104 is new (fail)
LEAF109 LEAF76 is new (OK)
LEAF109 LEAF75 is new (fail)
[...]
LEAF109 LEAF55 is a copy of LEAF109 LEAF29 ((x^5<B> . x^9<B>) (y^4<B> . y^7<B>)) (fail)
LEAF109 LEAF57 is a copy of LEAF109 LEAF31 ((y^4<B> . y^7<B>)) (fail)
LEAF109 LEAF56 is a copy of LEAF109 LEAF30 ((x^5<B> . x^9<B>)) (fail)
```

The connections above can be obtained by renaming the variables of previously-tested connections, allowing us to check their unifiability very quickly.

```
LEAF108 LEAF43 is OK is new (OK)
LEAF105 LEAF33 is OK is new (OK)
LEAF105 LEAF29 is new (fail)
LEAF105 LEAF31 is new (fail)
[...]
There are 90 acceptable connections (out of 699).
```

We have now checked all of the connections, although most of the output has been omitted for the sake of brevity. There were 699 connections in all, of which 90 were unifiable (the figure of 780 mentioned above is the number of pairs of literals, which is an upper bound on the number of connections; a pair of literals cannot be a connection unless it spans some vertical path).

```
Components:
(56 . 34.1) (56 . 34.2) (57 . 34.1) (57 . 34.2) (23 . 33.1) (23 . 33.2) (30 . 34.1) (30 . 34.2)
(30 . 55) (31 . 34.1) (31 . 34.2) (31 . 55) (29 . 56) (29 . 57) (33 . 23) (34 . 56) (34 . 57)
(34 . 30) (34 . 31) (39 . 23) (40 . 23) (43 . 34.1) (43 . 34.2) (43 . 34) (44 . 55) (44 . 29)
(65 . 34.1) (65 . 34.2) (65 . 55) (65 . 29) (65 . 34) (66 . 62) (66 . 63) (71 . 34.1)
(71 . 34.2) (71 . 55) (71 . 29) (71 . 34) (72 . 34.1) (72 . 34.2) (72 . 55) (72 . 29) (72 . 34)
(75 . 44) (75 . 66) (76 . 61) (89 . 33.1) (89 . 33.2) (89 . 24) (89 . 25) (89 . 33) (89 . 39)
(89 . 40) (90 . 33.1) (90 . 33.2) (90 . 24) (90 . 25) (90 . 33) (90 . 39) (90 . 40) (88 . 23)
(95 . 62) (95 . 63) (96 . 62) (96 . 63) (94 . 61) (98 . 33.1) (98 . 33.2) (98 . 24) (98 . 25)
(98 . 33) (98 . 39) (98 . 40) (99 . 66) (104 . 33.1) (104 . 33.2) (104 . 24) (104 . 25)
(104 . 33) (104 . 39) (104 . 40) (105 . 33.1) (105 . 33.2) (105 . 24) (105 . 25) (105 . 33)
(105 . 39) (105 . 40) (108 . 43) (109 . 76)
```

Above is a list of the 90 connections from which we will now build the minimal components.

```
Checking 150 possible minimal components +
((LEAF23 . LEAF33.1) (LEAF105 . LEAF25) (LEAF105 . LEAF24)) is OK+
((LEAF23 . LEAF33.2) (LEAF105 . LEAF25) (LEAF105 . LEAF24)) is OK+
((LEAF33 . LEAF23) (LEAF105 . LEAF25) (LEAF105 . LEAF24)) is OK+
((LEAF39 . LEAF23) (LEAF105 . LEAF25) (LEAF105 . LEAF24)) is OK+
((LEAF40 . LEAF23) (LEAF105 . LEAF25) (LEAF105 . LEAF24)) is OK-+
((LEAF23 . LEAF33.1) (LEAF104 . LEAF25) (LEAF105 . LEAF24)) is OK+
((LEAF23 . LEAF33.2) (LEAF104 . LEAF25) (LEAF105 . LEAF24)) is OK+
[...]
Fragment 1:(OR LEAF24 LEAF25 LEAF23) has 126 minimal components.
```

There were 150 triples of connections that might form a minimal component for this fragment; quick checking for compatibility of unifiers ruled out 24 of them, leaving 126.

```
Checking 96 possible minimal components --+
((LEAF44 . LEAF29) (LEAF34 . LEAF31) (LEAF34 . LEAF30)) is OK-----------+
((LEAF44 . LEAF29) (LEAF31 . LEAF34.2) (LEAF34 . LEAF30)) is OK-----+
[...]
Fragment 2:(OR LEAF30 LEAF31 LEAF29) has 9 minimal components.
```

Here we did rather better: of 96 possibilities, only 9 survived.

```
Checking 48 possible minimal components +
((LEAF34 . LEAF56) (LEAF105 . LEAF33)) is OK+
((LEAF34 . LEAF57) (LEAF105 . LEAF33)) is OK+
((LEAF34 . LEAF30) (LEAF105 . LEAF33)) is OK+
[...]
Fragment 3:(OR LEAF33 LEAF34) has 36 minimal components.
Checking 48 possible minimal components +
((LEAF56 . LEAF34.2) (LEAF105 . LEAF33.2)) is OK+
((LEAF57 . LEAF34.2) (LEAF105 . LEAF33.2)) is OK+
((LEAF30 . LEAF34.2) (LEAF105 . LEAF33.2)) is OK+
[...]
Fragment 4:(OR LEAF33.2 LEAF34.2) has 36 minimal components.
Checking 48 possible minimal components +
((LEAF56 . LEAF34.1) (LEAF105 . LEAF33.1)) is OK+
((LEAF57 . LEAF34.1) (LEAF105 . LEAF33.1)) is OK+
((LEAF30 . LEAF34.1) (LEAF105 . LEAF33.1)) is OK+
[...]
Fragment 5:(OR LEAF33.1 LEAF34.1) has 36 minimal components.
Checking 432 possible minimal components ------------------------------+
((LEAF75 . LEAF44) (LEAF108 . LEAF43) (LEAF105 . LEAF40) (LEAF104 . LEAF39)) is OK-----
[...]
Fragment 6:(OR LEAF39 LEAF40 ((FORALL q^0<OB>) (OR LEAF43 LEAF44))) has 1 minimal components.
```

We must have set BREAK-AT-QUANTIFIERS to NIL in order to get the above fragment ...

```
Checking 96 possible minimal components --+
((LEAF44 . LEAF55) (LEAF34 . LEAF57) (LEAF34 . LEAF56)) is OK-----------+
((LEAF44 . LEAF55) (LEAF57 . LEAF34.2) (LEAF34 . LEAF56)) is OK-----+
[...]
Fragment 7:(OR LEAF56 LEAF57 LEAF55) has 9 minimal components.
Checking 18 possible minimal components -------+
((LEAF94 . LEAF61) (LEAF96 . LEAF63) (LEAF95 . LEAF62)) is OK--------+
((LEAF76 . LEAF61) (LEAF66 . LEAF63) (LEAF66 . LEAF62)) is OK-
Fragment 8:(OR LEAF62 LEAF63 LEAF61) has 2 minimal components.
Checking 20 possible minimal components +
((LEAF66 . LEAF62) (LEAF65 . LEAF34)) is OK+
((LEAF66 . LEAF63) (LEAF65 . LEAF34)) is OK-+
((LEAF99 . LEAF66) (LEAF65 . LEAF34)) is OK--+
[...]
Fragment 9:(OR LEAF65 LEAF66) has 11 minimal components.
Checking 100 possible minimal components ---+
((LEAF109 . LEAF76) (LEAF75 . LEAF44) (LEAF72 . LEAF34) (LEAF71 . LEAF34)) is OK-----------+
((LEAF109 . LEAF76) (LEAF75 . LEAF44) (LEAF72 . LEAF34.2) (LEAF71 . LEAF34)) is OK---+
[...]
Fragment 10:(OR LEAF71 LEAF72 ((FORALL q^0<OA>) (OR LEAF75 LEAF76))) has 9 minimal components.
Checking 7 possible minimal components +
((LEAF89 . LEAF33.1)) is OK+
((LEAF89 . LEAF33.2)) is OK+
[...]
Fragment 11:LEAF89 has 7 minimal components.
Checking 7 possible minimal components +
((LEAF90 . LEAF33.1)) is OK+
((LEAF90 . LEAF33.2)) is OK+
[...]
Fragment 12:LEAF90 has 7 minimal components.
Checking 1 possible minimal components +
((LEAF88 . LEAF23)) is OK
Fragment 13:LEAF88 has 1 minimal components.
Checking 2 possible minimal components +
((LEAF95 . LEAF62)) is OK+
((LEAF95 . LEAF63)) is OK
Fragment 14:LEAF95 has 2 minimal components.
Checking 2 possible minimal components +
((LEAF96 . LEAF62)) is OK+
((LEAF96 . LEAF63)) is OK
```

```
Fragment 15:LEAF96 has 2 minimal components.
Checking 1 possible minimal components +
((LEAF94 . LEAF61)) is OK
Fragment 16:LEAF94 has 1 minimal components.
Checking 7 possible minimal components +
((LEAF98 . LEAF33.1)) is OK+
((LEAF98 . LEAF33.2)) is OK+
[...]
Fragment 17:LEAF98 has 7 minimal components.
Checking 1 possible minimal components +
((LEAF99 . LEAF66)) is OK
Fragment 18:LEAF99 has 1 minimal components.
Checking 7 possible minimal components +
((LEAF104 . LEAF33.1)) is OK+
((LEAF104 . LEAF33.2)) is OK+
[...]
Fragment 19:LEAF104 has 7 minimal components.
Checking 7 possible minimal components +
((LEAF105 . LEAF33.1)) is OK+
((LEAF105 . LEAF33.2)) is OK+
[...]
Fragment 20:LEAF105 has 7 minimal components.
Checking 1 possible minimal components +
((LEAF108 . LEAF43)) is OK
Fragment 21:LEAF108 has 1 minimal components.
Checking 1 possible minimal components +
((LEAF109 . LEAF76)) is OK
Fragment 22:LEAF109 has 1 minimal components.
```

All the minimal components have been computed. Now we choose an ordering on the fragments.

```
Lengths : (2 6 10 21 22 18 9 3 13 14 15 16 8 17 19 20 11 12 1 7 4 5)
```

Unfortunately, as we will see, our first fragment (2) is not part of any complete mating . . .

```
Now merging 9 current and 266 minimal components, with 0 stored.
```

We have started with the 9 minimal components for fragment 2. Each "." below marks the start of the attempt to enlarge one of these components.

```
. (blocks (2) and touches (2 5 6) ; 1 to try
```

So, for example, the first component touches but does not block both 5 and 6. Since 6 comes before 5 in our fragment ordering, we attempt to block 6. There is one candidate for this.

```
. (blocks (2) and touches (2 4 5 6) ; 1 to try
. (blocks (2) and touches (2 3 5 6) ; 1 to try
. (blocks (2) and touches (2 4 5 6) ; 1 to try
. (blocks (2) and touches (2 4 6) ; 1 to try
. (blocks (2) and touches (2 3 4 6) ; 1 to try
. (blocks (2) and touches (2 3 5 6) ; 1 to try
. (blocks (2) and touches (2 3 4 6) ; 1 to try
. (blocks (2) and touches (2 3 6) ; 1 to try
Merging produced 0 acceptable components.
Failed on fragment 2
Discarding current components and looking for stored components blocking 6.
```

None of the attempts to enlarge the minimal components for 2 produced any new components. We discard the fragment 2 and all components which touch it (reducing the number of minimal components from 266 to 234), and turn to the next fragment on our list, which is 6. There is only one component which blocks fragment 6.

```
Now merging 1 current and 234 minimal components, with 0 stored.
. (blocks (6 19 20 21) and touches (6 10 19 20 21) ; 9 to try
(109 . 76) (72 . 34.1) (71 . 34.1) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
(109 . 76) (72 . 34.2) (71 . 34.1) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(109 . 76) (72 . 34) (71 . 34.1) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(109 . 76) (72 . 34.1) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(109 . 76) (72 . 34.2) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
(109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(109 . 76) (72 . 34.1) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(109 . 76) (72 . 34) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
Merging produced 6 acceptable components.
```

Here we had more luck; nine attempts at merging produced six new components.

```
Now merging 6 current and 234 minimal components, with 0 stored.
. (blocks (6 10 19 20 21 22) and touches (4 5 6 10 19 20 21 22) ; 26 to try
. (blocks (6 10 19 20 21 22) and touches (3 5 6 10 19 20 21 22) ; 26 to try
. (blocks (6 10 19 20 21 22) and touches (4 5 6 10 19 20 21 22) ; 26 to try
```

```
. (blocks (6 10 19 20 21 22) and touches (3 4 6 10 19 20 21 22) ; 26 to try
(105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(65 . 34) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
(34 . 57) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
(34 . 56) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
. (blocks (6 10 19 20 21 22) and touches (3 5 6 10 19 20 21 22) ; 26 to try
. (blocks (6 10 19 20 21 22) and touches (3 4 6 10 19 20 21 22) ; 26 to try
(104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(65 . 34) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
(34 . 57) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
(34 . 56) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
Merging produced 8 acceptable components.
```

Of the six components from the last step, two of them could be enlarged, producing four more components each.

```
Now merging 8 current and 234 minimal components, with 0 stored.
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 10 19 20 21 22) ; 26 to try
(104 . 33.2) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 9 10 19 20 21 22) ; 10 to try
(66 . 63) (65 . 34.1) (65 . 34) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43)
(105 . 40) (104 . 39) is OK
(66 . 62) (65 . 34.1) (65 . 34) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43)
(105 . 40) (104 . 39) is OK
(66 . 63) (65 . 34) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40)
(104 . 39) is OK
(66 . 62) (65 . 34) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40)
(104 . 39) is OK
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 7 10 19 20 21 22) ; 9 to try
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 7 10 19 20 21 22) ; 9 to try
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 10 19 20 21 22) ; 26 to try
(105 . 33.2) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 9 10 19 20 21 22) ; 10 to try
(66 . 63) (65 . 34.1) (65 . 34) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43)
(105 . 40) (104 . 39) is OK
(66 . 62) (65 . 34.1) (65 . 34) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43)
(105 . 40) (104 . 39) is OK
(66 . 63) (65 . 34) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40)
(104 . 39) is OK
(66 . 62) (65 . 34) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40)
(104 . 39) is OK
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 7 10 19 20 21 22) ; 9 to try
. (blocks (3 6 10 19 20 21 22) and touches (3 4 6 7 10 19 20 21 22) ; 9 to try
Merging produced 10 acceptable components.
Now merging 10 current and 234 minimal components, with 0 stored.
. (blocks (3 4 6 10 19 20 21 22) and touches (3 4 6 10 14 19 20 21 22) ; 1 to try
(94 . 61) (96 . 63) (95 . 62) (104 . 33.2) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44)
(108 . 43) (105 . 40) (104 . 39) is OK
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 5 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 5 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 4 6 10 19 20 21 22) and touches (3 4 6 10 14 19 20 21 22) ; 1 to try
(94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33) (109 . 76) (72 . 34.2) (71 . 34) (75 . 44)
(108 . 43) (105 . 40) (104 . 39) is OK
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 5 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 5 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 6 8 9 10 19 20 21 22) ; 2 to try
. (blocks (3 6 9 10 19 20 21 22) and touches (3 4 6 8 9 10 19 20 21 22) ; 2 to try
Merging produced 2 acceptable components.
Now merging 2 current and 234 minimal components, with 2 stored.
. (blocks (3 4 6 8 10 14 15 16 19 20 21 22) and touches (3 4 6 8 10 14 15 16 18 19 20 21 22) ;
3 to try
(99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62) (104 . 33.2) (105 . 33) (109 . 76) (72 . 34)
(71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(99 . 66) (65 . 34.2) (94 . 61) (96 . 63) (95 . 62) (104 . 33.2) (105 . 33) (109 . 76) (72 . 34)
```

```
(71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
(99 . 66) (65 . 34) (94 . 61) (96 . 63) (95 . 62) (104 . 33.2) (105 . 33) (109 . 76) (72 . 34)
(71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
. (blocks (3 4 6 8 10 14 15 16 19 20 21 22) and touches (3 4 6 8 10 14 15 16 18 19 20 21 22) ;
3 to try
(99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33) (109 . 76) (72 . 34.2)
(71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(99 . 66) (65 . 34.2) (94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33) (109 . 76) (72 . 34.2)
(71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
(99 . 66) (65 . 34) (94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33) (109 . 76) (72 . 34.2)
(71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) isn't unifiable.
Merging produced 2 acceptable components.
Now merging 2 current and 234 minimal components, with 4 stored.
. (blocks (3 4 6 8 9 10 14 15 16 18 19 20 21 22) and touches (3 4 5 6 8 9 10 14 15 16 18 19 20
21 22) ; 26 to try
(98 . 33.1) (99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62) (104 . 33.2) (105 . 33) (109 . 76)
(72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
. (blocks (3 4 6 8 9 10 14 15 16 18 19 20 21 22) and touches (3 4 5 6 8 9 10 14 15 16 18 19 20
21 22) ; 26 to try
(98 . 33.1) (99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33) (109 . 76)
(72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(57 . 34.1) (98 . 33.1) (99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33)
(109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
(56 . 34.1) (98 . 33.1) (99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62) (105 . 33.2) (104 . 33)
(109 . 76) (72 . 34.2) (71 . 34) (75 . 44) (108 . 43) (105 . 40) (104 . 39) is OK
Merging produced 4 acceptable components.
Now merging 4 current and 234 minimal components, with 4 stored.
. (blocks (3 4 5 6 8 9 10 14 15 16 17 18 19 20 21 22) and touches (3 4 5 6 8 9 10 13 14 15 16 17
18 19 20 21 22) ; 1 to try
(88 . 23) (90 . 25) (89 . 24) (98 . 33.1) (99 . 66) (65 . 34.1) (94 . 61) (96 . 63) (95 . 62)
(104 . 33.2) (105 . 33) (109 . 76) (72 . 34) (71 . 34.2) (75 . 44) (108 . 43) (105 . 40) (104 . 39)
is OK
```

Better than "OK", in fact, since our completeness checker realises that this is a complete mating ...

```
Success! The following is a complete mating:
((LEAF88 . LEAF23) (LEAF90 . LEAF25) (LEAF89 . LEAF24) (LEAF98 . LEAF33.1) (LEAF99 . LEAF66)
(LEAF65 . LEAF34.1) (LEAF94 . LEAF61) (LEAF96 . LEAF63) (LEAF95 . LEAF62) (LEAF104 . LEAF33.2)
(LEAF105 . LEAF33) (LEAF109 . LEAF76) (LEAF72 . LEAF34) (LEAF71 . LEAF34.2) (LEAF75 . LEAF44)
(LEAF108 . LEAF43) (LEAF105 . LEAF40) (LEAF104 . LEAF39))
```

Now we call ADD-CONN, from the MATE top level, to add the new connections into a mating that will be recognised by the rest of TPS. *Strictly speaking, we could save some time and simply construct this mating without passing through ADD-CONN; however, this step is quite fast and helps to catch any bugs in MS98-1 by re-checking our mating using the previously-existing procedures.*

```
Adding new connection: (LEAF88 . LEAF23)  0
Adding new connection: (LEAF90 . LEAF25)  0
Adding new connection: (LEAF89 . LEAF24)  0
Adding new connection: (LEAF98 . LEAF33.1)  0
Adding new connection: (LEAF99 . LEAF66)  0
Adding new connection: (LEAF65 . LEAF34.1)  0
Adding new connection: (LEAF94 . LEAF61)  0
Adding new connection: (LEAF96 . LEAF63)  0
Adding new connection: (LEAF95 . LEAF62)  0
Adding new connection: (LEAF104 . LEAF33.2)  0
Adding new connection: (LEAF105 . LEAF33)  0 0-0 0
Adding new connection: (LEAF109 . LEAF76)  0
Adding new connection: (LEAF72 . LEAF34)  0
Adding new connection: (LEAF71 . LEAF34.2)  0
Adding new connection: (LEAF75 . LEAF44)  0 0-0 0-0-0 0
Adding new connection: (LEAF108 . LEAF43)  0
Adding new connection: (LEAF105 . LEAF40)  0
Adding new connection: (LEAF104 . LEAF39)  0
Substitution Stack:

y^6(G)  ->  Y^0(G)
x^10(G)  ->  X^0(G)
x^18(G)  ->  X^1(G)
x^10(B)  ->  H^1(BG) X^1(G)
y^3(A)  ->  Y^0(A)
x^3(A)  ->  X^0(A)
x^17(G)  ->  X^2(G)
x^11(G)  ->  Y^1(G)
q^0(OA)  ->  LAMBDA w^57(A) Q(OA) w^57
```

```
y^8(B)  ->  H^1(BG) Y^1(G)
x^11(B)  ->  H^1(BG) X^2(G)
q^0(OB)  ->  LAMBDA w^59(B) Q(OA).H^2(AB) w^59
y^7(G)  ->  Y^1(G)
x^12(G)  ->  X^2(G)
The time used in each process:
```

Process Name	Realtime	Internal-runtime	GC-time	I-GC-time
MS THM126	4	4.51	0.39	4.12
U THM126	2	1.52	0.19	1.33

The new mating is indeed complete. It took 4.12 seconds to find it, of which 1.33 were spent on unification. At this point, we may call MERGE-TREE and ETREE-NAT to construct a natural deduction proof of the theorem.

Bibliography

[Andrews and Bishop, 1996] Peter B. Andrews and Matthew Bishop. On Sets, Types, Fixed Points, and Checkerboards. In Miglioli et al. [1996], pages 1–15.

[Andrews *et al.*, 1984] Peter B. Andrews, Dale A. Miller, Eve Longini Cohen, and Frank Pfenning. Automating Higher-Order Logic. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, Contemporary Mathematics series, vol. 29, pages 169–192. American Mathematical Society, 1984.

[Andrews *et al.*, 1996] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A Theorem Proving System for Classical Type Theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[Andrews, 1976] Peter B. Andrews. Refutations by Matings. *IEEE Transactions on Computers*, C-25:801–807, 1976.

[Andrews, 1980] Peter B. Andrews. Transforming Matings into Natural Deduction Proofs. In Bibel and Kowalski [1980], pages 281–292.

[Andrews, 1981] Peter B. Andrews. Theorem Proving via General Matings. *Journal of the ACM*, 28:193–214, 1981.

[Andrews, 1986] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

[Andrews, 1989] Peter B. Andrews. On Connections and Higher-Order Logic. *Journal of Automated Reasoning*, 5:257–291, 1989.

[Baas and Leitsch, 1994] M. Baas and A. Leitsch. On Skolemisation and Proof Complexity. *Fundamenta Informaticæ*, 20:353–379, 1994.

[Bachmair *et al.*, 1986] Leo Bachmair, Nachum Dershowitz, and Jieh Hsiang. Orderings for Equational Proofs. In LICS-1 [1986], pages 346–357.

[Bachmair *et al.*, 1987] Leo Bachmair, Nachum Dershowitz, and David Plaisted. Completion Without Failure. In *Proceedings of the Colloquium on the Resolution of Equations in Algebraic Structures (CREAS)*, Lakeway, Texas, 1987.

[Bachmair *et al.*, 1992] L. Bachmair, H. Ganzinger, H. Lynch, and W. Snyder. Basic Paramodulation and Superposition. In Kapur [1992], pages 462–476.

[Bachmair, 1991] Leo Bachmair. *Canonical Equational Proofs*. Birkhäuser, Boston, 1991.

[Bailin and Barker-Plummer, 1993] Sidney C. Bailin and Dave Barker-Plummer. Z-match: An Inference Rule for Incrementally Elaborating Set Instantiations. *Journal of Automated Reasoning*, 11:391–428, 1993. Errata: JAR 12 (1994), 411–412.

[Barendregt, 1984] H. P. Barendregt. *The λ-Calculus*. Studies in logic and the foundations of mathematics, North-Holland, 1984.

[Baumgartner *et al.*, 1995] Peter Baumgartner, Reiner Hähnle, and Joachim Posegga, editors. *Theorem Proving with Analytic Tableaux and Related Methods. 4th International Workshop. (TABLEAUX '95)*, volume 918 of *Lecture Notes in Artificial Intelligence*, Schloß Rheinfels, St. Goar, Germany, May 1995. Springer-Verlag.

[Baumgartner *et al.*, 1996] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper Tableaux. In *Proceedings of JELIA-96, European Workshop on Logic in AI*, volume 1126 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1996.

[Baumgartner, 1997] Peter Baumgartner. Hyper Tableaux — The Next Generation. Technical Report 32/97, Fachbereich Informatik, Universität Koblenz-Landau, 1997.

[Benzmüller and Kohlhase, 1998] Christoph Benzmüller and Michael Kohlhase. Extensional Higher-Order Resolution. In Kirchner and Kirchner [1998], pages 56–71.

[Benzmüller *et al.*, 1997] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a Mathematical Assistant. In McCune [1997b], pages 252–255.

[Benzmüller *et al.*, 1999] Christoph Benzmüller, Matthew Bishop, and Volker Sorge. Integrating TPS and ΩMEGA. *Journal of Universal Computer Science*, May 1999. (to appear).

[Benzmüller, 1998] Christoph Benzmüller. An Adaptation of Paramodulation and RUE-resolution to Higher-Order Logic. SEKI Report SR-98-07, Universität des Saarlandes, 1998.

[Bibel and Kowalski, 1980] W. Bibel and R. Kowalski, editors. *Proceedings of the 5th International Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, Les Arcs, France, 1980. Springer-Verlag.

[Bibel *et al.*, 1995] W. Bibel, S. Bruning, U. Egly, D. Korn, and T. Rath. Issues in Theorem Proving Based on the Connection Method. In Baumgartner et al. [1995], pages 1–16.

[Bibel, 1981] Wolfgang Bibel. On Matrices with Connections. *Journal of the ACM*, 28(4):633–645, October 1981.

[Bibel, 1987] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, second edition, 1987.

[Billon, 1996] Jean-Paul Billon. The Disconnection Method. In Miglioli et al. [1996].

[Birkhoff, 1934] G. Birkhoff. Applications of Lattice Algebra. *Proc. Camb. Phil. Soc.*, 30:115–122, 1934.

[Bishop and Andrews, 1998] Matthew Bishop and Peter B. Andrews. Selectively Instantiating Definitions. In Kirchner and Kirchner [1998], pages 365–380.

[Blasius *et al.*, 1981] K. Blasius, N. Eisinger, J. Siekmann, G. Smolka, A. Herold, and C. Walther. The Markgraph Karl Refutation Procedure. In IJCAI-7 [1981], pages 511–518.

[Bledsoe and Feng, 1993] W. W. Bledsoe and Guohui Feng. Set-Var. *Journal of Automated Reasoning*, 11:293–314, 1993.

[Bol *et al.*, 1991] R.N. Bol, K.R. Apt, and J.W. Klop. An Analysis of Loop Checking Mechanisms for Logic Programming. *Theoretical Computer Science*, 85:35–79, 1991.

[Boyer *et al.*, 1986] Robert Boyer, Ewing Lusk, William McCune, Ross Overbeek, Mark Stickel, and Lawrence Wos. Set Theory in First-Order Logic: Clauses for Gödel's Axioms. *Journal of Automated Reasoning*, 2:287–327, 1986.

[Brüning, 1994] S. Brüning. *Techniques for Avoiding Redundancy in Theorem Proving Based on the Connection Method*. PhD thesis, TH Darmstadt, 1994.

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira. Deduction Revision by Intelligent Backtracking. In J. A. Campbell, editor, *Implementations of Prolog*, pages 194–215. Ellis Horwood, 1984.

[Bryant, 1986] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bundy, 1994] Alan Bundy, editor. *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, Nancy, France, 1994. Springer-Verlag.

[Chang and Lee, 1973] C. Chang and R.C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[Church and Rosser, 1936] Alonzo Church and J.B. Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, 3:472–482, 1936.

[Church, 1940] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Cox and Pietrzykowski, 1981] Philip T. Cox and Tomasz Pietrzykowski. Deduction Plans: A Basis for Intelligent Backtracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:52–65, 1981.

[Cox, 1977] Philip T. Cox. *Deduction Plans: A Graphical Proof Procedure for the First-Order Predicate Calculus*. PhD thesis, University of Waterloo, 1977.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[Davis, 1963] Martin Davis. Eliminating the Irrelevant from Mechanical Proofs. In *Experimental Arithmetic, High Speed Computing and Mathematics*, Proceedings of Symposia in Applied Mathematics XV, pages 15–30. American Mathematical Society, 1963.

[Dedekind, 1900] R. Dedekind. Über die von drei Moduln erzeugte Dualgruppe. *Mathematische Annalen*, 53:371–403, 1900.

[Degtyarev and Voronkov, 1995a] Anatoli Degtyarev and Andrei Voronkov. General Connections via Equality Elimination. Technical Report UPMAIL-93, Computing Science Department, Uppsala University, Sweden, 1995.

[Degtyarev and Voronkov, 1995b] Anatoli Degtyarev and Andrei Voronkov. Simultaneous Rigid E-Unification is Undecidable. Technical Report UPMAIL-105, Computing Science Department, Uppsala University, Sweden, 1995.

[Degtyarev and Voronkov, 1997] Anatoli Degtyarev and Andrei Voronkov. What You Always Wanted to Know About Rigid E-Unification. Technical Report UPMAIL-143, Computing Science Department, Uppsala University, Sweden, 1997.

[Degtyarev *et al.*, 1997] Anatoli Degtyarev, Yuri Gurevich, Paliath Narendran, Margus Veanes, and Andrei Voronkov. The Decidability of Simultaneous Rigid E-Unification with One Variable. Technical Report UPMAIL-139, Computing Science Department, Uppsala University, Sweden, 1997. (to appear in Theoretical Computer Science).

[Dershowitz, 1993] Nachum Dershowitz. A Taste of Rewriting. In P.E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning. International Lecture Series 1991–92*, pages 199–228. Springer-Verlag, 1993.

[Digricoli, 1979] Vincent J. Digricoli. Resolution by Unification and Equality. In William H. Joyner, editor, *4th Workshop on Automated Deduction*, Austin, Texas, 1979.

[Eisinger *et al.*, 1991] Norbert Eisinger, Hans-Jürgen Ohlbach, and Axel Präcklein. Reduction Rules for Resolution-Based Systems. *Artificial Intelligence*, 50(2):141–181, 1991.

[Eisinger, 1986] N. Eisinger. What You always Wanted to Know About Clause Graph Resolution. In Siekmann [1986], pages 316–336.

[Fay, 1979] M. Fay. First-Order Unification in an Equational Theory. In William H. Joyner, editor, *4th Workshop on Automated Deduction*, Austin, Texas, 1979.

[Fitting, 1990] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1990.

[Gallier *et al.*, 1990] Jean H. Gallier, P. Narendran, David Plaisted, and Wayne Snyder. Rigid E-Unification: NP-Completeness and Applications to Theorem Proving. *Information and Computation*, 87:129–195, 1990.

[Gallier *et al.*, 1992] Jean H. Gallier, Paliath Narendran, Stan Raatz, and Wayne Snyder. Theorem Proving Using Equational Matings and Rigid E-Unification. *Journal of the ACM*, 39:377–429, 1992.

[Gentzen, 1935] G. Gentzen. Untersuchungen über das Logische Schließen I und II. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935. Translated in [Gentzen, 1969].

[Gentzen, 1969] G. Gentzen. Investigations into Logical Deductions. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland Publishing Co., Amsterdam, 1969.

[Gilmore, 1960] P.C. Gilmore. A Proof Method for Quantification Theory. *IBM Journal of Research and Development*, 4:28–35, 1960.

[Goldfarb, 1981] Warren D. Goldfarb. The Undecidability of the Second-Order Unification Problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Graf, 1995] Peter Graf. *Term Indexing*, volume 1053 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1995.

[Gurevich and Voronkov, 1997] Yuri Gurevich and Andrei Voronkov. Monadic Simultaneous Rigid E-Unification and Related Problems. In *ICALP '97: 24th International Colloquium on Automata, Languages and Programming, Bologna, Italy*, volume 1256 of *Lecture Notes in Computer Science*, pages 154–165. Springer-Verlag, July 1997. (to appear in Theoretical Computer Science).

[Harrison, 1996] John Robert Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, Churchill College, Cambridge, 1996. Revised version available as Technical Report TR408, Computer Laboratory, Cambridge University.

[Herbrand, 1930] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et des Lettres de Varsovie, Classe III Sciences Mathematiques et Physiques*, 33, 1930. Translated in [Herbrand, 1971].

[Herbrand, 1971] Jacques Herbrand. *Logical Writings*. Harvard University Press, 1971. Edited by Warren D. Goldfarb.

[Hörnig and Bibel, 1982] K.M. Hörnig and W. Bibel. Improvements of a Tautology-Testing Algorithm. In Loveland [1982], pages 326–341.

[Huang *et al.*, 1994] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jorn Richts, and Jörg Siekmann. Ω-MKRP: A Proof Development Environment. In Bundy [1994], pages 788–792.

[Huet, 1973] Gerard P. Huet. The Undecidability of Unification in Third-order Logic. *Information and Control*, 22:257–267, 1973.

[Huet, 1975] Gerard P. Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Huet, 1981] Gerard P. Huet. A Complete Proof of Correctness of the Knuth and Bendix Completion Algorithm. *Journal of Computer and System Sciences*, 23:11–21, 1981.

[Hungerford, 1974] Thomas W. Hungerford. *Algebra*. Springer-Verlag New York, Inc., 1974.

[IJCAI-7, 1981] IJCAI. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, B.C., Canada, 1981.

[Issar, 1990] Sunil Issar. Path-Focused Duplication: A Search Procedure for General Matings. In *AAAI–90. Proceedings of the Eighth National Conference on Artificial Intelligence*, volume 1, pages 221–226. AAAI Press/The MIT Press, 1990.

[Issar, 1991] Sunil Issar. *Operational Issues in Automated Theorem Proving Using Matings*. PhD thesis, Carnegie Mellon University, 1991. 147 pp.

[Janičić *et al.*, 1997] Predrag Janičić, Ian Green, and Alan Bundy. A Comparison of Decision Procedures in Presburger Arithmetic. In *Proceedings of the VIII International Conference on Logic and Computer Science (LIRA 97)*, pages 99–101, Novi Sad, Yugoslavia, September 1997. Also available as Research Report 872, Department of AI, University of Edinburgh.

[Johnson, 1994] R. Johnson. A Blackboard Approach to Parallel Temporal Tableaux. In P. Jorrand and V. Sgurev, editors, *Artificial Intelligence, Methodologies, Systems and Applications (AIMSA '94)*, pages 183–192, Sofia, Bulgaria, 1994. World Scientific, Singapore.

[Jordan, 1949] P. Jordan. Zum Dedekindschen Axiom in der Theorie der Verbände. *Abhandl. Hamburg*, 16:71–73, 1949.

[Jouannaud and Kirchner, 1991] J.-P. Jouannaud and C. Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification. In J.-L. Lassez and G. Plotkin, editors, *Festschrift for Robinson*. MIT Press, Cambridge, MA, 1991.

[Kapur, 1992] D. Kapur, editor. *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, Saratoga Springs, NY, USA, 1992. Springer-Verlag.

[Kirchner and Kirchner, 1998] Claude Kirchner and Hélène Kirchner, editors. *Proceedings of the 15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, 1998. Springer-Verlag.

[Knaster, 1927] B. Knaster. Une Théorème sur les Fonctions d'Ensembles. *Annales Soc. Polonaise Math.*, 6:133–134, 1927.

[Knuth and Bendix, 1970] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebra. In J. Leech, editor, *Computational Word Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Kryvyi, 1998] S.L. Kryvyi. Unification Problem in Equational Theories. *Cybernetics and Systems Analysis*, 33(6):874–899, 1998.

[Kurfeß, 1990] F. Kurfeß. Potentiality of Parallelism in Logic. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems: International Workshop Proceedings*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 3–25, Dagstuhl Castle, Germany, 1990. Springer-Verlag.

[Lankford, 1975] D.S. Lankford. Canonical Inference. Technical Report ATP–32, University of Texas, 1975.

[Lee and Plaisted, 1989] Shie-Jue Lee and David A. Plaisted. Theorem Proving Using Hyper-Matching Strategy. *Methodologies for Intelligent Systems*, 4:467–476, 1989.

[Lee and Plaisted, 1992] Shie-Jue Lee and David A. Plaisted. Eliminating Duplication with the Hyper-Linking Strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.

[Lee, 1990] Shie-Jue Lee. *CLIN: An Automated Reasoning System Using Clause Linking*. PhD thesis, University of North Carolina at Chapel Hill, 1990.

[LICS-1, 1986] *Proceedings of the IEEE Symposium on Logic in Computer Science*, Cambridge, MA, USA, 1986. IEEE Computer Society Press, Los Alamitos, California, USA.

[LICS-8, 1993] *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science*, Montreal, Canada, 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[Loveland, 1982] Donald W. Loveland, editor. *Proceedings of the 6th International Conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, New York, USA, 1982. Springer-Verlag.

[McCune, 1992] William McCune. Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. *Journal of Automated Reasoning*, 9(2):147–167, 1992.

[McCune, 1997a] William McCune. 33 Basic Test Problems: A Practical Evaluation of Some Paramodulation Strategies. In Robert Veroff, editor, *Automated Reasoning and its Applications: Essays in Honor of Larry Wos*, pages 71–114. MIT Press, 1997.

[McCune, 1997b] William McCune, editor. *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, Townsville, North Queensland, Australia, 1997. Springer-Verlag.

[Middeldorp and Hamoen, 1994] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Journal of Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[Miglioli *et al.*, 1996] Pierangelo Miglioli, Ugo Moscato, Daniele Mundici, and Mario Ornaghi, editors. *Theorem Proving with Analytic Tableaux and Related Methods. 5th International Workshop. (TABLEAUX '96)*, volume 1071 of *Lecture Notes in Artificial Intelligence*, Terrasini, Italy, May 1996. Springer-Verlag.

[Miller *et al.*, 1982] Dale A. Miller, Eve Longini Cohen, and Peter B. Andrews. A Look at TPS. In Loveland [1982], pages 50–69.

[Miller, 1983] Dale A. Miller. *Proofs in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1983. 81 pp.

[Miller, 1984] Dale A. Miller. Expansion Tree Proofs and Their Conversion to Natural Deduction Proofs. In Shostak [1984], pages 375–393.

[Miller, 1987] Dale A. Miller. A Compact Representation of Proofs. *Studia Logica*, 46(4):347–370, 1987.

[Milner, 1978] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[Müller and Weber, 1994] Olaf Müller and Franz Weber. Theory and Practice of Minimal Modular Higher-Order E-Unification. In Bundy [1994], pages 650–664.

[Murray and Rosenthal, 1993] Neil V. Murray and Erik Rosenthal. Dissolution: Making Paths Vanish. *Journal of the ACM*, 40(3):504–535, July 1993.

[Nesmith *et al.*, 1996] Dan Nesmith, Matthew Bishop, Peter B. Andrews, Sunil Issar, Frank Pfenning, and Hongwei Xi. TPS *User's Manual*, 1996. 72+ii pp.

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[Nipkow and Qian, 1991] Tobias Nipkow and Zhenyu Qian. Modular Higher-Order E-Unification. In Ronald V. Book, editor, *Proceedings of the 4th International Conference on Rewriting Techniques and Applications*, volume 488 of *Lecture Notes in Computer Science*, pages 200–214. Springer-Verlag, 1991.

[Nipkow, 1993] Tobias Nipkow. Functional Unification of Higher-Order Patterns. In LICS-8 [1993].

[Paterson and Wegman, 1978] M.S. Paterson and M.N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Pfenning and Nesmith, 1990] Frank Pfenning and Dan Nesmith. Presenting Intuitive Deductions via Symmetric Simplification. In Stickel [1990], pages 336–350.

[Pfenning, 1984] Frank Pfenning. Analytic and Non-Analytic Proofs. In Shostak [1984], pages 394–413.

[Pfenning, 1987] Frank Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie Mellon University, 1987. 156 pp.

[Prawitz, 1970] Dag Prawitz. A Proof Procedure with Matrix Reduction. In M. Laudet, D. Lacombe, L. Nolin, and M. Schutzenberger, editors, *Symposium on Automatic Demonstration, Versailles, France*, Lecture Notes in Mathematics 125, pages 207–214. Springer-Verlag, 1970.

[Prehofer, 1997] Christian Prehofer. *Solving Higher-Order Equations: From Logic to Programming*. Birkhäuser Boston, 1997.

[Presburger, 1930] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Sprawozdanie z I Kongresu metematyków slowiańskich, Warszawa 1929*, pages 92–101,395, Warsaw, Poland, 1930. Annotated English version in [Stansifer, 1984].

[Quaife, 1990] Art Quaife. Andrews' Challenge Problem Revisited. *Association for Automated Reasoning Newsletter*, 15:3–7, May 1990.

[Quigley, 1998] Claire Quigley. An Alternative Approach to Matingsearch. (Unpublished). Computing Science Department, Glasgow University, December 1998.

[Quine, 1955] W. V. Quine. A Proof Procedure for Quantification Theory. *Journal of Symbolic Logic*, 20:141–149, 1955.

[Ramesh *et al.*, 1995] Anavai Ramesh, Neil V. Murray, Bernhard Beckert, and Reiner Hähnle. Fast Subsumption Checks Using Anti-Links. Interner Bericht 24/95, Universität Karlsruhe, Fakultät für Informatik, April 1995.

[Robinson and Wos, 1969] Arthur Robinson and Larry Wos. Paramodulation and TP in First Order Theories with Equality. *Machine Intelligence*, 4:135–150, 1969.

[Robinson, 1965] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12:23–41, 1965.

[Schröder, 1890] E. Schröder. *Algebra der Logik*. Teubner-Verlag, Leipzig, 1890.

[Schumann, 1990] Johann Schumann. Parallel Theorem Provers: An Overview. In B. Fronhöfer and G. Wrightson, editors, *Parallelization in Inference Systems: International Workshop Proceedings*, volume 590 of *Lecture Notes in Artificial Intelligence*, pages 26–50, Dagstuhl Castle, Germany, 1990. Springer-Verlag.

[Shostak, 1984] R. E. Shostak, editor. *Proceedings of the 7th International Conference on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, Napa, California, USA, 1984. Springer-Verlag.

[Siekmann, 1984] Jörg Siekmann. Universal Unification. In Shostak [1984], pages 1–42.

[Siekmann, 1986] Jörg H. Siekmann, editor. *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, Oxford, England, 1986. Springer-Verlag.

[Siekmann, 1989] Jörg Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7:207–274, 1989.

[Slagle, 1974] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 1974.

[Smullyan, 1968] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, 1968.

[Snyder and Gallier, 1989] Wayne Snyder and Jean Gallier. Higher-Order Unification Revisited: Complete Sets of Transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Snyder, 1990] Wayne Snyder. Higher-Order E-Unification. In Stickel [1990], pages 573–587.

[Snyder, 1991] Wayne Snyder. *A Proof Theory for General Unification*. Birkhäuser Boston, 1991.

[Stansifer, 1984] R. Stansifer. Presburger's Article on Integer Arithmetic: Remarks and Translation. Technical Report TR84–639, Cornell University Computer Science Department, 1984.

[Stickel, 1985] Mark E. Stickel. Automated Deduction by Theory Resolution. *Journal of Automated Reasoning*, 1:333–355, 1985.

[Stickel, 1990] M. E. Stickel, editor. *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, Kaiserslautern, Germany, 1990. Springer-Verlag.

[Szász, 1963] Gabor Szász. *Introduction to Lattice Theory*. Academic Press, New York and London, 1963.

[Tarski, 1951] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951. Previous version published as a technical report by the RAND Corporation, 1948; prepared for publication by J.C.C. McKinsey.

[Vagin and Salapina, 1998] V.N. Vagin and N.O. Salapina. Parallel Inference on Connection Graphs. In *Proceedings of the 1998 IEEE International Symposium on Intelligent Control*, pages 204–209, Gaithersburg, MD, USA, 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[Voronkov, 1998] Andrei Voronkov. Simultaneous Rigid E-Unification and Other Decision Problems Related to the Herbrand Theorem. Technical Report UPMAIL-152, Computing Science Department, Uppsala University, Sweden, 1998.

[Wallen, 1990] Lincoln A. Wallen. *Automated Deduction in Nonclassical Logics*. MIT Press, 1990.

Index

$\xrightarrow{\beta}$, 8
β-normal form, 8
β-reduction, 8
$\overset{\beta}{=}$, 8
$\xrightarrow{\sim}$, 8
$\overset{\beta}{=}_{\sim}$, 8
$\prec_{(\rho,J)}$, 12
$\simeq$, 74
$\sim_J$, 19, 93
ΩMEGA, 84

acyclic, 12
amplification, 11
Apply algorithm, 60
atom, 9

blackboard architecture, 92
blocked, 20
break-at-quantifiers, 93, 115

CLIN, 48
collapse, 32, 61
Collapse algorithm, 61–66, 128
combine, 25
compatible
  substitutions, 59
  unification graphs, 60
complete
  component, 25
  mating, 13
completeness, 45
  checking, 29
  restricted, 44
completion, 82
  unfailing, 82
components, 24
  combining, 25
  complete, 25
  current, 42, 46, 91
  elementary, 75
  minimal, 24
  pending, 46
  stored, 42, 46, 91
  strong, 25
  weak, 25
conjunction node, 10
connection, 12
  acceptable, 42, 91
  method, 1

deep formula, 9
default-ms, 115
discrimination tree, 80
disjunction
  heuristic, 21
  node, 10
DIY, 115
duplication, 11, 71
  by need, 72
duplications
  total number of, 72
dynamic reordering, 99

E-unification, 83
eq-connections, 83
eq-unifiers, 83
EQP, 84
equality
  elimination, 83
  global, 79
  Leibniz, 79
  local, 79
  reasoning, 82
  unification modulo, 79
equivalent matings, 74
essentially existential, 9
essentially universal, 9
ET-proofs, 9
existential node, 10
expansion
  nodes, 10
  proofs, 9, 10
  terms, 10
  trees, 9, 10

extensionality, 83

ff-delay, 115, 119
first-order-mode-ms, 115
flex-flex pairs, 52
fragment, 19
  ordering, 94
  weak, 22

GO, 115

horizontal paths, 12
hpath-closure, 33
hpath-complement, 32
hpath-threshold, 94, 115
Huet's algorithm, 52
hyperlinking, 48

imbedding relation, 12
interpretations, 84

jforms, 10

Leibniz variables, 79
linked conjunct, 48
literal, 9

mating, 11
  complete, 13
  minimal, 37
  strong, 22
  weak, 22
mating search method, 1
max-mates, 92, 116, 121, 122
max-search-limit, 118
max-substs-var, 116, 121
maximize-first, 116
merge
  for substitutions, 60
  for unification graphs, 61
Merge algorithm, 61, 62, 64–66, 81, 97, 128
merge-minimize-mating, 116
min-quantifier-scope, 116, 124
most general common instance, 60
MS98, 115
ms98-1, 115
ms98-base-prim, 95, 116, 117
ms98-dup, 117
ms98-dup-below-primsubs, 117
ms98-dup-primsubs, 117
ms98-first-fragment, 117
ms98-force-h-o, 115, 117
ms98-fragment-order, 95, 117
ms98-init, 78, 117, 120, 123
ms98-low-memory, 118
ms98-max-components, 118
ms98-max-prims, 118
ms98-measure, 47, 95, 118
ms98-merge-dags, 100, 119
ms98-minimality-check, 120
MS98-MINOR, 115
ms98-num-of-dups, 72, 73, 118, 120, 123
ms98-primsub-count, 120
ms98-rew-primsubs, 120
ms98-rewrite-depth, 120, 121
ms98-rewrite-model, 87, 120
ms98-rewrite-prune, 120
ms98-rewrite-size, 120
ms98-rewrite-unif, 121
ms98-rewrites, 121
ms98-unif-hack, 121
ms98-unif-hack2, 121
ms98-valid-pair, 96, 118, 119, 121
ms98-variable-order, 99, 122
ms98-verbose, 123

narrowing, 82
negation normal form, 8
negative occurrences, 9
normal jform, 10
num-of-dups, 72, 117, 118, 120, 123

OBDD, 52
order-components, 28
ordered binary decision diagrams, 52

p-acceptable, 12
paramodulation, 82
parity, 84
partial mating, 13
path dissolution, 16
path-acceptable, 12
permutation
  of jform, 74
  of universal nodes, 74
positive occurrences, 9
PRIMSUBS, 118
Prune algorithm, 54, 56, 66
pruning, 95

Reduce algorithm, 54–56, 61, 63–66
reduction operation, 16, 96
redundancy, 16, 95
refinement, 93
refutation, 13
relabelling, 74
rewrite rules
  active, 81

rewrite tree, 80
rewrite-defns, 123
rewrite-equalities, 123
rewrite-equivs, 94, 124
rewriting, 82
rigid E-unification, 83
ruleq, 116

search algorithm
  basic, 41
  parallel, 91
  weighted, 45
  with equality, 81
selection
  nodes, 10, 69
  parameters, 10, 97
shallow formula, 9
skolem-default, 97, 124
Skolemization, 69, 97
soundness, 44
spans, 12
strong, 22
substitution, 8
substitutions
  for set-variables, 77
  primitive, 77
subsumption, 16
superposition, 83
  rigid, 83

touched, 20
translation, 87
types, 7

unification, 51
  delaying, 100
  first-order, 15, 51, 96
  graph, 56, 97, 98
  higher-order, 15, 51, 96
  pruning, 97
unifier, 11
universal node, 10

valid-p, 86, 87
variable ordering, 66, 98
variable renaming, 74
vertical paths, 11

weak, 22
well-formed formulae, 7
wffs, 7