

Horn Clauses and Feature-Structure Logic: Principles and Unification Algorithms

Stephen J. Hegner [†]
University of Bergen
Department of Informatics
Bergen High-Technology Center
N-5020 Bergen, Norway

Internet: hegner@ii.uib.no

(Revised: June 1993)

This paper appears as Report No. 1, June 1993, in the series *Reports in Logic, Language, and Information*, published by: University of Oslo, Department of Linguistics, P.O. Box 1102, N-0317 Oslo, Norway.

An abbreviated version of this report appears in the book *Constraints, Language and Computation*, edited by M. Rosner, C. J. Rupp, and R. Johnson, published by Academic Press, 1994, pp. 111-147.

Abstract

The desirability of Horn clauses in logical deductive systems has long been recognized. The reasons are at least threefold. Firstly, while inference algorithms for full logics of any reasonable extent are typically intractable, for systems restricted to Horn clauses the picture is much better. (For example, in ordinary propositional logic, while the full satisfiability problem is NP-complete, a linear-time algorithm exists for Horn clauses.) Secondly, the knowledge-representation capabilities of Horn clauses, while weaker than those of the full logic, remain remarkably rich; indeed, far richer than that of simple conjunctive logic alone. Thirdly, Horn clauses define the maximal subset of a full logic which has the property of admitting generic models, which roughly means that for any set of Horn clauses, there is a least model of the clauses in that set.

It is the purpose of this paper to initiate an investigation of Horn clause logic for an extended class of feature structures. After laying the groundwork for this context, we provide two key results. In the first, we show how the property of admitting generic models can be extended to feature structure logic, and demonstrate the importance of this property in generalizing the use of ordinary feature structures in unification-based formalisms. Our second contribution is a tractable unification algorithm for extended feature structures constrained by Horn clauses. This algorithm is an integration of the traditional unification algorithm for feature structures and the linear-time inference algorithm for propositional logic.

[†]Current address: Department of Computer Science and Electrical Engineering, Votey Building, University of Vermont, Burlington, VT 05405, USA; Internet: hegner@emba.uvm.edu.

0. Introduction

Motivation

Broadly speaking, in a unification-based approach to parsing, the parser generates units of partial information about the final result. As these units are generated, the critical operation of unification is employed to combine them into more global descriptions. If all goes well, when the process is completed, the desired semantic representation may be extracted from the global description obtained from the unification of all of the partial components.

Feature structures (or attribute-value structures, as they are sometimes called), have been a cornerstone of knowledge representation in traditional unification-based approaches to natural language parsing [Shi86], [FLV89]. The reason for their utility in this context is twofold. First of all, their capacity for knowledge representation is two dimensional. At the same time, a feature structure M may be regarded as a specific data structure describing the form of a given piece of input (*e.g.*, a sentence or sentence fragment), and, through the set of structures into which it may grow (*i.e.*, the set of feature structures $\text{Ext}(M)$ which it subsumes), it may be regarded as defining a family of possibilities into which that basic piece of input may grow. In other words, a feature structure may be regarded, at the same time, both as a representation of partial information and as a final, fully determined result.

The second key property of feature structures is that a highly efficient algorithm exists for the critical operation of unification. Given two feature structures M_1 and M_2 , which may be regarded as identifying their sets of possible extensions $\text{Ext}(M_1)$ and $\text{Ext}(M_2)$ respectively, this algorithm determines whether or not there is a common extension (*i.e.*, whether or not $\text{Ext}(M_1) \cap \text{Ext}(M_2)$ is nonempty), and, if so, it also determines a unique (up to isomorphism) feature structure M_3 with the property that $\text{Ext}(M_3) = \text{Ext}(M_1) \cap \text{Ext}(M_2)$. M_3 is the so-called *unification* of M_1 and M_2 , often denoted $M_1 \sqcup M_2$.

While this traditional approach is both efficient and useful, it does have a serious limitation. Namely, semantic constraints cannot be directly expressed within the units of partial information generated by the parser. Indeed, the only kind of constraint that can be expressed is the purely syntactic one stating that the final result must be a member of $\text{Ext}(M)$ for a given feature structure M . For this reason, a number of researchers have investigated the problem of extending this framework to include various forms of constraints. Along with the increase in expressive power which constraints provide, however, comes a corresponding increase in the computational complexity of unification algorithms. Sometimes, this increase in complexity is nominal. For example, in [Lan89], as part of a comprehensive study of the issue of negation in feature structures, Langholm has provided a formalism in which constraints involving simple (atomic) negation may be handled with essentially no increase in computational complexity over the standard unification algorithm. Unfortunately, the situation is not so favorable in the context of more general constraints. Rounds and Kasper [RK86] have shown that testing for satisfaction of general logical constraints on feature structures is an NP-complete problem, and so

barring a major unexpected breakthrough in the theory of computational complexity, the best known algorithms for solving such problems will remain of exponential complexity. And since the unification problem in the presence of constraints may be viewed as one of testing for satisfiability, it follows that if we are to allow general constraints, we cannot expect a tractable (*i.e.*, polynomial-time) unification algorithm.

There is a further drawback to adopting a full feature-structure logic as a constraint language. Namely, it becomes impossible to extract from a set of constraints a canonical or *generic* model which represents exactly the information which has been determined to be true at the current point in the unification process. In the general case, the best that we can do is identify a set of possible alternatives for the representative model.

We will show that Horn clauses are the largest subset of the full set of feature-structure logic constraints which avoid both of these difficulties. Highly efficient algorithms for satisfiability testing have long been known. The most comprehensive and recent work is that of Dowling and Gallier [DG84], who provide (under reasonable assumptions) a linear-time satisfiability algorithm for Horn propositional logic. This is without question a large improvement over the exponential time complexity of the best known algorithm for full propositional logic. Furthermore, Makowsky [Mak87] has shown, for proposition and for first-order logic, that the property of a set of structures admitting generic models is equivalent to that set being definable by Horn clauses. The major goal of this report is to extend these two results to the logic of feature structures.

In establishing these results, it is not our claim that Horn clause constraints are all that is necessary. There is already a substantial body of research on unification in the presence of disjunctive constraints on feature structures, of which [Kas87], [ED88], [DE90], and [Str91] are but a sample. But all of these works investigate disjunction within the context of ordinary feature structures. *Our position is that Horn-clause-constrained feature structures, rather than ordinary feature structures, ought to be considered as a natural starting point for the investigation of more general classes of constraints.* This has at least two advantages. Firstly, only those types of constraints which cannot be handled within the Horn clause context will be external; this would ensure efficient unification in a maximal percentage of cases. Secondly, the circumstances under which a single generic model characterizes the result will be completely apparent. While such genericity is not always possible or even desired, it is certainly useful to know exactly when it does apply. It is in this spirit that we present this work on the rôle of Horn clauses in unification-based formalisms.

Horn Clauses and Knowledge Representation

Basically, Horn clauses are rules or implicational constraints; that is, formulas of the form $\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_n \Rightarrow \rho$, in which the σ_i 's and ρ are positive atomic statements. No negation is allowed in either the antecedents or the consequents. Horn clauses form the basis for both the programming language *Prolog* [SS86] and the database query language *Datalog* [CGT90]. But surprisingly — at least to this author —

there seems to be little use of this form of knowledge representation in the computational linguistics literature. The only paper of which the author is aware which deals directly with implicational constraints is that of Kasper [Kas88]. However, his approach is one of special techniques on particular types of conditions, rather than a general approach which works for all Horn clauses of the logic which he and Rounds developed. The survey report [Wed90] mentions implicational constraints in its Section 1.3, but then only briefly, suggesting that they may be reduced to computations on negation and disjunction. While this sort of reduction is certainly possible in a formal sense, such an approach is unlikely to produce the kind of efficient unification algorithm which is associated with Horn clauses.

The emphasis of this paper is upon the mathematical aspects of the representation language and the algorithmic aspects of unification, and not upon the detailed problems of modelling linguistic knowledge using Horn formulas. However, it is important that the reader have a general idea of the kinds of constraints which Horn formulas can and cannot model. The following four items provide an abstract taxonomy of Horn clauses.

- (H1) A clause of the form ρ , consisting of a single positive literal, is just a *fact*. (More formally, we may think of a fact as a rule of the form $\top \Rightarrow \rho$, where \top is the atom which is always true.)
- (H2) A clause of the form $\neg\sigma$, consisting of a single negative literal, is a *negated fact*. (More formally, we may think of a negated fact as a rule of the form $\sigma \Rightarrow \perp$, where \perp is the atom which is always false.)
- (H3) A clause of the form $\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_m \Rightarrow \rho$ is called a *rule* or an *implication*.
- (H4) A clause of the form $\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_m \Rightarrow \perp$ is called a *compound negation*.

To illustrate the scope of the use of Horn clauses in the representation of linguistic knowledge, we now present five simple examples illustrating implication and compound negation. (Facts and negated facts are so fundamental that hopefully examples are not necessary.) As our purpose is to give a feeling for the scope of this form of representation, rather than a rigorous presentation, we avoid the formalities of feature structures entirely, and represent the atomic facts using English-language statements. However, the reader familiar with feature structures should have no trouble translating these into formal representations. In 2.14, we give more precise mathematical representations of these forms of constraints.

(Example L1) In many languages, the subject and verb of a sentence must agree in person and number. Informally, this may be represented as a rule of the following form.

If *the person of the subject is defined*
 and *the person of the verb is defined*
then *the person of the subject and the person of the verb are coalesced.*

The term *coalesced* means that the two values are represented by the same data object. Coalescing of data fields is a principal aspect of feature structures.

(Example L2) Consider the constraint which stipulates that a transitive verb always takes a direct object. This may be represented via a rule of the form

If *the type of the verb is transitive*
then *there must be a direct object.*

(Example L3) Consider the constraint which mandates that an intransitive verb cannot take a direct object. Another way to think of this constraint is that it is impossible to have both an intransitive verb and an indirect object. Thus, we can represent this constraint as the following compound negation.

If *the type of the verb is intransitive and a direct object is defined*
then *contradiction.*

(Example L4) The Norwegian language has two types of possessive object pronouns, reflexive and nonreflexive. As a specific example, consider the English sentence *John washed his car*. There are two ways to translate this sentence into Norwegian. If we use the reflexive pronoun *sin* as the translation of *his*, the meaning is that John washed his own car. On the other hand, if we use the nonreflexive version *hans*, then the meaning is that John washed someone else's car. (See [SSVV90] for a thorough discussion of this aspect of Norwegian, and see [Per87, p. 1006] for a more general discussion of this sort of example.) Suppose that our knowledge representation language is rich enough that we may speak of the *owner* of a direct object. We may then represent the use of the reflexive form by the following two rules.

If *the subject and the owner of the object are coalesced*
then *use the reflexive form.*

If *the reflexive form is used*
then *the subject and the owner of the object are coalesced.*

In constructing this representation using Horn clauses, we have employed a simple technique. The initial constraint is an “if and only if” requirement, *i.e.*, *use the reflexive form* if and only if *the subject and the owner of object are coalesced*. We decomposed this two-way rule into two one-way rules.

(Example L5) Let us show how certain rules which appear to be disjunctive may in fact be expressed in a Horn clause format. Consider the example concerning the German preposition *in*, which is taken from [DE90]. As a preposition representing direction (motion), the object of a prepositional phrase based upon *in* takes the accusative form. On the other hand, as a preposition representing a static position, its object takes the dative form. As a disjunctive constraint, this may be expressed as follows.

(*The relationship is static and the case is dative*)
or
(*the relationship is directional and the case is accusative.*)

Here is how we may represent the same constraints in a rule form.

If *the relationship is static* **then** *the case is dative.*

If *the relationship is directional* **then** *the case is accusative.*

If *the case is dative* **then** *the relationship is static.*

If *the case is accusative* **then** *the relationship is directional.*

Actually, this set of rules is not quite the same as the disjunctive representation. The rules only bind the case to the relationship, while the disjunction also stipulates that the relationship is either static or directional, while the case is either accusative or dative. However, this sort of value binding can usually be accomplished in other ways, such as by assigning types to attributes. In any case, the example shows that the essence of the constraint may be expressed in rule form.

The type of knowledge which is not recapturable using Horn feature logic is positive disjunction; *i.e.*, formulas of the form $\sigma_1 \vee \sigma_2$, with both σ_1 and σ_2 positive atomic statements. It is not our claim that disjunctive knowledge is unnecessary, or that it can always be translated into Horn rule form. Rather, the thrust of our argument is twofold.

- (1) The versatility of knowledge representation which is available using Horn clauses, coupled with the efficient unification algorithm which is available, strongly suggests that this form of knowledge representation should be used more extensively.
- (2) As we argued above, when disjunction is necessary, techniques to handle it should be built on top of the Horn clause unification algorithm, rather than the unification algorithm for unconstrained feature structures. In this way, all forms of constraints may be handled in the most expeditious fashion possible.

Prerequisites and Outline of this Report

While we have tried to make the technical aspects of the paper reasonably self contained, we do assume certain background. First and foremost, a familiarity with the classical aspects of unification of feature structures will prove most helpful. The monograph [Shi86] and the report [FLV89] both provide gentle introductions to unification within the context of linguistics, while the report [Rea91] provides a rigorous mathematical perspective. The paper [AN86] provides a perspective on such structures from a programming languages (rather than computational linguistics) point of view. This latter paper contains a particularly understandable presentation of the classical unification algorithm for feature structures. The algorithmic notation which we use is Pascal-like, and we assume that the reader has some facility in

reading such specifications. We assume at least a minimal familiarity with the ideas which found simple mathematical logics, such as may be found in the first few chapters of [Dav89], as well as familiarity with the basic notation and ideas of algorithm analysis, as may be found in [MS91]. Finally, MBFS's are modelled using a simple form of finite automaton. While no detailed knowledge of automata theory is necessary, the knowledge of some simple concepts from this field, as may be found in [CL89] may prove helpful.

The ultimate goal of this report, the unification algorithm for feature structures constrained by Horn clauses, is presented in Section 4. Sections 1 through 3 provide the building block necessary to assemble and understand this algorithm. Firstly, in Section 1, we introduce a (conceptually straightforward) generalization of feature structures which we term *multiple-base feature structures* or *MBFS's*. We then present the traditional unification algorithm for these structures in a way in which it can later be used as the first building block for our ultimate algorithm. In Section 2, we define the logic which allows us to express constraints on MBFS's, and we define precisely what is meant by a Horn clause in that context. In Section 3, we review the fast inference algorithm of Dowling and Gallier [DG84] which is the second building block of our final unification algorithm. Section 4, as already mentioned, then details the merger of these two building-block algorithms. Finally, Section 5 contains a few conclusions and suggestion for further investigation.

Some of the ideas of this paper have already appeared in preliminary form in an earlier conference paper [Heg91]. However, that report did not detail the algorithms the way that this report does, nor did it make the extension from ordinary feature structures to multiple-base feature structures.

1. Multiple-Base Feature Structures

In this section, we introduce *multiple-base feature structures*, which are a generalization of ordinary feature structures admitting multiple starting nodes. We then provide a unification algorithm for such structures.

Fundamental Definitions and Properties

We begin with a brief review of ordinary feature structures. This presentation is meant to put our work in perspective, rather than to provide a tutorial introduction. Those unfamiliar with these ideas are encouraged to consult the references identified in the introduction.

1.1 The ideas behind feature structures Feature structures are extensible record-like structures. Figure 1.1 depicts two typical feature structures in graphical form. (The node names — the xi 's and yi 's — are not of central importance. We have just included them to provide convenient reference points.) The corresponding record-like representation for these structures is shown in Figure 1.2. With respect to ordinary record-like structures in programming languages, the most distinguishing property of feature structures is their ability to share fields. Note that in the

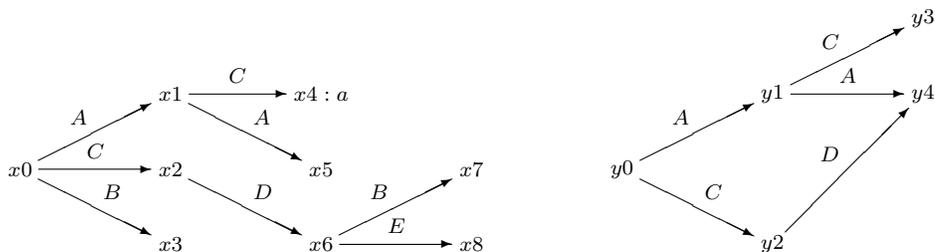


Figure 1.1: Two typical feature structures.

<pre> record A: record C: a; A: nil; end record; C: record D: record B: nil; E: nil end record; end record; B: nil; end record; </pre>	<pre> record A: record C: nil; A: nil tag:y4 end record; C: record D: nil tag:y4 end record; end record; </pre>
--	---

Figure 1.2: Record-like representation of two typical feature structures.

description on the right in Figure 1.2, the notation `tag:y4` occurs. This means that the two fields of the record are *coalesced* — they are the same item of data. In the graphical representation of Figure 1.1, the corresponding paths converge to the same node.

In unification-based approaches to parsing, components of the parsing process generate partial descriptions of the final result, which are then combined via the process of unification. Often, these partial descriptions are represented as feature structures. The unification of two feature structures is the smallest feature structure which contains all of the knowledge contained in either of the operands. The unification of the two structures of Figure 1.1 is shown in Figure 1.3 below.

This unification is obtained by a simple algorithm. For convenience, let us call the left feature structure in Figure 1.1 M_l , the right structure M_r , and the structure in Figure 1.3 M . First, we identify the root nodes $\{x_0, y_0\}$ of M_l and M_r , and associate this pair with the root node of M . Then we look for common attributes of this pair of nodes. In the example, A and C are common attributes. We then identify the corresponding nodes $\{x_1, y_1\}$ and $\{x_2, y_2\}$ of these attributes. We repeat this process until no pair of identified nodes has common attributes which have not been merged. The node labels in M reflect the final merging. There is only one way in which this process can fail, arising from the restriction that only nodes with no

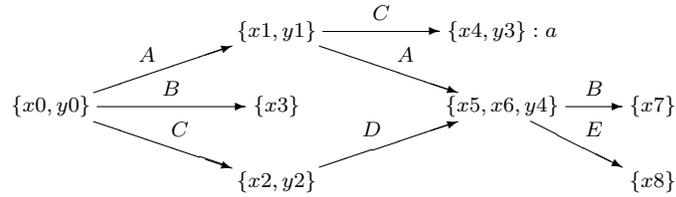


Figure 1.3: The unification of the feature structures of Figure 1.1.

outgoing edges can be labelled with atomic values, and there may only be one atomic value per node. Thus, if node $y3$ in M_r were labelled with b , then the unification would fail, because node $\{x4, y3\}$ in M would have to be labelled with both a and b . Similarly, if node $y4$ in M_r had a label (a , say), then unification would not be possible, since the corresponding node $x6$ in M_l has outgoing edges.

When one feature structure N_1 grows into another N_2 , we say that N_1 *subsumes* N_2 , and write $N_1 \sqsubseteq N_2$.¹ The unification of N_1 and N_2 is denoted by $N_1 \sqcup N_2$.² We always have that $N_1 \sqsubseteq N_1 \sqcup N_2$, so in particular, each structure in Figure 1.1 subsumes the structure in Figure 1.3.

Feature structures can grow in three ways, and all three are illustrated by this unification example. Firstly, new attributes may be added. For example, in following the path CD from the root in M we see a record structure with attributes C and E , while in M_r this same path shows no attributes. Secondly, new atomic values may be added to nodes. In M_r , the path AC from the root shows no assigned value, yet in M , we find a assigned to the node at the end of that path. Finally, paths may be coalesced. In M_l the paths AA and CD from the root are distinct, yet in M they are coalesced.

1.2 A shortcoming of ordinary feature structures While unification is the critical process in unification-based parsing formalisms, it is not always the case that two structures are unified at the root. Consider the context of the four feature structures depicted in Figure 1.4.

It makes perfect sense to unify the structure with root $x20$ and the substructure rooted at $x11$ of the structure with root $x10$, yielding the structure depicted in Figure 1.5 below.

Even the unification algorithm makes perfect sense in this context; we just work with the feature substructure rooted at $x11$, ignoring the part unreachable from $x11$. In terms of a cohesive formalism, there is a problem, however. Suppose we perform a sequence of unifications of the structures depicted in Figure 1.4, in the following order. Unify $x20$ and $x30$, unify $x11$ and $x20$, unify $x10$ and $x40$, unify $x41$ and $x30$.

¹Unfortunately, some authors define subsumption in exactly the reverse way, *i.e.*, N_1 subsumes N_2 iff N_2 may grow into N_1 , so the reader new to this field should check the sense of the definitions carefully when studying other work.

²Again, be warned that some authors write $N_1 \sqcap N_2$ for the unification of N_1 and N_2 .



Figure 1.4: Four simple feature structures.

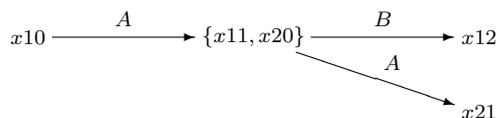


Figure 1.5: A simple unification in which one base node is not a root.

It is easy to see that each unification results in a feature structure, and that the final result is as shown in Figure 1.6. One of the hallmark properties of unification-based

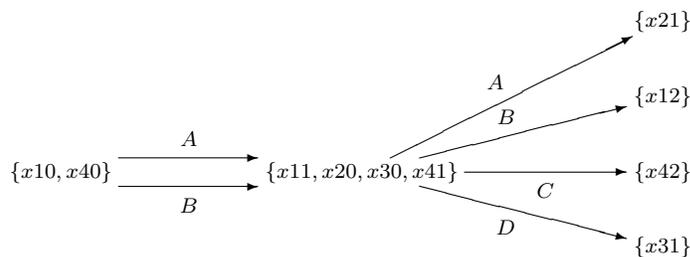


Figure 1.6: Result of a sequence of unifications on the structures of Figure 1.4

formalisms is that the unifications may be performed in any order; more formally, the system is confluent in the sense that the final result is independent of the order of the operations. However, suppose that we switch the order of the last two operations in the unification example, so that our sequence is now unify x_{20} and x_{30} , unify x_{11} and x_{20} , unify x_{41} and x_{30} , unify x_{10} and x_{40} . The result after all but the last operation is performed is depicted in Figure 1.7 below. This structure is not a feature structure in the traditional sense, because there is not a unique root node from which all other nodes are reachable. While we may not want a *final* result which looks like this, if we are going to admit *intermediate* results in this form — and we must so do if we wish our system to be confluent — then we must extend the notion of feature structure to admit multiple roots. With this idea in mind, we now turn to a more formal development of the notion of a feature structure with multiple roots, which we term a *multiple-base feature structure*.

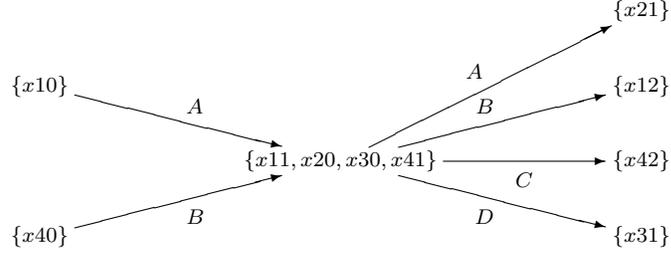


Figure 1.7: Result of another sequence of unifications on the structures of Figure 1.4

1.3 Some basic notation The formal model of a multiple-base feature structure which we use is that of a certain form of finite automaton. We assume familiarity with the basic ideas and notation from that field, as may be found in, *e.g.*, [CL89]. In particular, we assume that the reader is familiar with the description of a finite automaton as a rooted graph.

As for specific notation, if X is a set, then X^* denotes the set of all strings of elements of X . The empty string is denoted by λ . When we speak of a *partial function* $f : A \rightarrow B$, we explicitly allow the possibility that f is in fact a total function. Throughout this paper, we use the notation $f(x) \downarrow$ to mean that the partial function f is defined on argument x , and we use $f(x) \uparrow$ to denote that it is undefined.

1.4 Multiple-base feature structures A *context* for multiple-base feature structures is a triple $\mathcal{C} = (\mathcal{E}, \mathcal{V}, \mathcal{B})$ in which \mathcal{E} is a finite or countably infinite set of *attributes*, \mathcal{V} is a finite or countably infinite set of *atomic values*, and \mathcal{B} is a countably infinite set of *base labels*. A *multiple-base feature structure* (MBFS for short) over $\mathcal{C} = (\mathcal{E}, \mathcal{V}, \mathcal{B})$ is a quadruple $M = (Q, \delta, \alpha, \beta)$ in which

- (fs-i) Q is a finite set, called the set of *states*.
- (fs-ii) $\delta : Q \times \mathcal{E} \rightarrow Q$ is a partial function, called the *state-transition function*.
- (fs-iii) $\alpha : Q \rightarrow \mathcal{V}$ is a partial function, called the *assignment function*.
- (fs-iv) $\beta : \mathcal{B} \rightarrow Q$ is a partial function, called the *base-labelling function*. The set $\beta(\mathcal{B}) = \{\beta(b) \mid \beta(b) \text{ is defined}\}$ is called the set of *initial states*.

The following conditions shall always be enforced.

- (fs-v) Every state is reachable from some initial state. More precisely, for each $q \in Q$, there is a $q_0 \in \beta(\mathcal{B})$ and $\omega \in \mathcal{E}^*$ such that $\delta^*(q_0, \omega) = q$. Note that this implies, in particular, that if Q is nonempty, then there must be at least one initial state; *i.e.*, $\beta(b)$ must be defined for some $b \in \mathcal{B}$. (Throughout the paper, $\delta^* : Q \times \mathcal{E}^* \rightarrow Q$ denotes the extension of δ

to strings defined by $\delta^*(q, \lambda) = q$ and $\delta^*(q, \omega \cdot e) = \delta(\delta^*(q, \omega), e)$ for all $q \in Q$, $e \in \mathcal{E}$, and $\omega \in \mathcal{E}^*$.

(fs-vi) Atomic values can only label terminal nodes. More precisely, for each $q \in Q$, if there is an $e \in \mathcal{E}$ such that $\delta(q, e) \downarrow$, then $\alpha(q) \uparrow$.

The class of all MBFS's over the context \mathcal{C} is denoted by $\text{MBFS}(\mathcal{C})$. *Throughout this paper, unless explicitly stated otherwise, we shall always assume that all MBFS's are over a specific context \mathcal{C} .*

A state of the form $\beta(\ell)$ for $\ell \in \mathcal{B}$ is called a *base state* or *initial state* for M . We can easily recover the traditional notion of a feature structure by limiting \mathcal{B} to be a one-element set $\{\ell\}$, and requiring that $\beta(\ell)$ be defined. Then every MBFS will have exactly one initial state, identified by $\beta(\ell)$.³

1.5 Differences between our definition and those of others Our definition is based upon the automaton-theoretic definition of ordinary feature structures which is due to Kasper and Rounds [RK86]. In addition to allowing multiple initial states, there are a few other differences. First of all, we do not legislate away the possibility that the automaton may be cyclic; that is, that it may have a state q for which there is a nonempty string ω with $\delta^*(q, \omega) = q$. While the issue of whether or not cyclicity is a necessary property for linguistic work is a controversial one, with respect to the work reported here, there is no problem in allowing cyclic structures. Secondly, we do not require that the assignment function α be injective. Many formalisms require that an atomic value may be used to label at most one node. This condition may be easily accommodated, if desired, and we will note how to alter our unification algorithms to take it into account.

1.6 Example The feature structures depicted in Figure 1.1 may be regarded as MBFS's once we provide a base-labelling function. So, let us label $x0$ with ℓ_1 and $y0$ with ℓ_2 . Now, not only is each of M_l and M_r with these labels an MBFS, but the entire diagram also represents a *single* MBFS. Indeed, any collection of ordinary feature structures may be regarded as a single MBFS, once we label their initial states. This is the great utility of the MBFS formalism — as the parsing process provides new feature structures, we label their base nodes and “throw them in” with the existing structure, to get a new, larger MBFS. A single representation covers all.

Just to make sure that there is no confusion, we review the other concepts in terms of this example. The state set Q is just the set of nodes of the graph, and the transitions of δ are defined by the edges of the graph. So, for example, $\delta(x0, A) = x1$. The assignment function α is defined only for the node $x4$ in this example, with $\alpha(x4) = a$.

³The definition of a traditional feature requires that there be exactly one initial state, while our definition admits the possibility that there be no states at all. This is not an essential difference, and will not affect our results in any way.

1.7 Morphisms and subsumption Let $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$ and $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$ be MBFS's. A *morphism* $h : M_1 \rightarrow M_2$ is a (total) function $h : Q_1 \rightarrow Q_2$ which is subject to the following conditions.

(mor-i) For each pair $(q, e) \in Q_1 \times \mathcal{E}$, $\delta_1(q, e) \downarrow$ implies that $\delta_2(h(q), e) \downarrow$ and $h(\delta_1(q, e)) = \delta_2(h(q), e)$.

(mor-ii) For each $q \in Q_1$, $\alpha_1(q) \downarrow$ implies $\alpha_2(h(q)) \downarrow$ and $\alpha_1(q) = \alpha_2(h(q))$.

(mor-iii) For each $\ell \in \mathcal{B}$, if $\beta_1(\ell) \downarrow$, then $\beta_2(h(\ell)) \downarrow$ and $\beta_2(h(\ell)) = h(\beta_1(\ell))$.

In the case that there is a morphism $h : M_1 \rightarrow M_2$, we say that M_1 *subsumes* M_2 , and write $M_1 \sqsubseteq M_2$.

1.8 Lemma — extension of (mor-i) Let $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$ and $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$ be MBFS's, and let $h : M_1 \rightarrow M_2$ be a morphism. Then for any $\omega \in \mathcal{E}^*$ and any $q \in Q_1$, if $\delta_1^*(q, \omega)$ is defined, then so too is $\delta_2^*(h(q), \omega)$, with $h(\delta_1^*(q, \omega)) = \delta_2^*(h(q), \omega)$

PROOF: The proof is a simple inductive argument, which is left to the reader. \square

1.9 Proposition — uniqueness of morphisms Let $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$ and $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$ be MBFS's. Then there is at most one morphism $M_1 \rightarrow M_2$.

PROOF: Let $f, g : M_1 \rightarrow M_2$ be morphisms. By property (fs-v), every $q \in Q_1$ is of the form $\delta_1^*(\beta_1(\ell), \omega)$ for some $\ell \in \mathcal{B}$ and some $\omega \in \mathcal{E}^*$. Furthermore, by property (mor-iii), $f(\beta_1(\ell)) = \beta_2(\ell) = g(\beta_1(\ell))$, and so using the previous lemma we have $f(q) = f(\delta_1^*(\beta_1(\ell), \omega)) = \delta_2^*(f(\beta_1(\ell)), \omega) = \delta_2^*(g(\beta_1(\ell)), \omega) = g(\delta_1^*(\beta_1(\ell), \omega)) = g(q)$. Hence $f = g$, and so the morphism is unique. \square

1.10 Example Consider the structures depicted in Figure 1.1 as a single MBFS, as described in 1.6. Now regard the structure depicted in Figure 1.3 as an MBFS as well, labelling the node $\{x0, y0\}$ with two labels, ℓ_1 and ℓ_2 . In other words, $\beta(\ell_1) = \beta(\ell_2) = \{x0, y0\}$. (Note that there is absolutely no requirement that distinct labels be associated with distinct nodes. Indeed, the process of coalescing nodes may very well coalesce labels as well.) Then the function which sends each node in Figure 1.1 to the node in Figure 1.3 of which it is formally a member, *e.g.*, $x6 \mapsto \{x5, x6, y4\}$, defines a morphism of MBFS's.

Notice that the MBFS of Figure 1.3 could have other node labels without destroying the property of a morphism. For example, if the node $\{x1, y1\}$ had label ℓ_3 , that would be of no consequence in determining whether or not a morphism exists. However, if the MBFS of Figure 1.1 had other node labels, say $y1$ were labelled ℓ_4 , then it would be mandatory that $\{x1, y1\}$ also be labelled with ℓ_4 .

1.11 Isomorphism of MBFS's We start with two simple but important observations. First of all, for any MBFS $M = (Q, \delta, \alpha, \beta)$, the identity function $\mathbf{1}_Q : Q \rightarrow Q$ clearly defines a morphism $\mathbf{1}_M : M \rightarrow M$, which we call the *identity morphism* for M . Secondly, given any three MBFS's $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$, $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$,

and $M_3 = (Q_3, \delta_3, \alpha_3, \beta_3)$, and any two morphisms $f : M_1 \rightarrow M_2$ and $g : M_2 \rightarrow M_3$, the composition $g \circ f : M_1 \rightarrow M_3$, defined by the composition $g \circ f : Q_1 \rightarrow Q_3$ of the underlying functions, is an MBFS morphism. Now we follow the standard categorical definition of isomorphism [HS73, 5.13]. Specifically, we say that M_1 and M_2 are *isomorphic* if there are morphisms $f : M_1 \rightarrow M_2$ and $g : M_2 \rightarrow M_1$ such that $f \circ g = \mathbf{1}_{M_2}$ and $g \circ f = \mathbf{1}_{M_1}$. In this case, we write $M_1 \cong M_2$, and we call f and g *isomorphisms*.

We have the following characterization of MBFS isomorphism. Note that (iii) below gives the more intuitive notion of an automaton morphism — isomorphic MBFS's are the same up to a renaming of the states.

1.12 Lemma — characterization of isomorphisms *Let $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$ and $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$ be MBFS's. Then the following conditions are equivalent.*

- (i) M_1 and M_2 are isomorphic.
- (ii) $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$.
- (iii) *There is a morphism $h : M_1 \rightarrow M_2$ which satisfies the following three conditions.*
 - (iso-i) *For each pair $(q, e) \in Q_1 \times \mathcal{E}$, $\delta_1(q, e) \downarrow$ iff $\delta_2(h(q), e) \downarrow$, and $h(\delta_1(q, e)) = \delta_2(h(q), e)$ when both are defined.*
 - (iso-ii) *For each $q \in Q_1$, $\alpha_1(q) \downarrow$ iff $\alpha_2(h(q)) \downarrow$, and $h(\alpha_1(q)) = \alpha_2(h(q))$ when both are defined.*
 - (iso-iii) *For each $\ell \in \mathcal{B}$, $\beta_1(\ell) \downarrow$, iff $\beta_2(\ell) \downarrow$, and $\beta_2(\ell) = h(\beta_1(\ell))$ when both are defined.*

PROOF: (iii) \Rightarrow (ii): Assume that h satisfies (iso-i) through (iso-iii) above. It is easy to verify that both h and its inverse function $h^{-1} : Q_2 \rightarrow Q_1$ define MBFS morphisms. Indeed, conditions (iso-i) through (iso-iii) are just strengthenings of conditions (mor-i) through (mor-iii) to make them “bidirectional.” Hence $M_1 \sqsubseteq M_2$ and $M_2 \sqsubseteq M_1$, as was to be shown.

(ii) \Rightarrow (i): Let $f : M_1 \rightarrow M_2$ and $g : M_2 \rightarrow M_1$ be morphisms. Then $g \circ f : M_1 \rightarrow M_1$ is also a morphism. But we know that $\mathbf{1}_{M_1} : M_1 \rightarrow M_1$ is a morphism, whence $g \circ f = \mathbf{1}_{M_1}$, by the uniqueness result 1.5. Similarly, $f \circ g = \mathbf{1}_{M_2}$, and so M_1 and M_2 are isomorphic.

(i) \Rightarrow (iii) If M_1 and M_2 are isomorphic, then we have morphisms $f : M_1 \rightarrow M_2$ and $g : M_2 \rightarrow M_1$ with $g \circ f = \mathbf{1}_{M_1}$ and $f \circ g = \mathbf{1}_{M_2}$. The idea of the proof is to iterate the (mor-n) conditions for $n = i, ii, iii$ twice, first for f and then for g , to get condition (iso-n). We illustrate for condition (iso-i). If $(q, e) \in Q_1 \times \mathcal{E}$ is such that $\delta_1(q, e) \downarrow$, then $\delta_2(f(q), e) \downarrow$. But then, applying rule (mor-i) again for g , we have that $\delta_2(f(q), e) \downarrow$ implies that $\delta_1((g \circ f)(q), e) = \delta_1(q, e) \downarrow$. Hence $\delta_1(q, e) \downarrow$ iff $\delta_2(f(q), e) \downarrow$, whence condition (iso-i) holds. The proofs of (iso-ii) and (iso-iii) are similar. \square

1.13 The order structure of $\text{MBFS}(\mathcal{C})$ On $\text{MBFS}(\mathcal{C})$, \sqsubseteq is a preorder; that is, a reflexive and transitive relation. It is not a partial order because we may have chains of the form $M_1 \sqsubseteq M_2 \sqsubseteq M_1$ in the case that M_1 and M_2 are isomorphic. However, in view of 1.12 above, this is the only way in which such chains can arise. Therefore, if we identify isomorphic structures, we will get a partial order. More precisely, let $[M]$ denote the equivalence class, under isomorphism, of the MBFS M . We use the notation $[\text{MBFS}(\mathcal{C})]$ to denote the set of all such equivalence classes, and we write $[M_1] \sqsubseteq [M_2]$ to mean that $M_1 \sqsubseteq M_2$. Then, in view of 1.12, this notation is well defined and defines a partial order on $[\text{MBFS}(\mathcal{C})]$.

As a matter of course, it will seldom be necessary to distinguish between isomorphic MBFS's. Therefore, when no confusion can arise, we will simply regard $\text{MBFS}(\mathcal{C})$ as a partially ordered set, with the implicit assumption that isomorphic MBFS's are "the same." In short, we often omit the brackets [...].

In general, least upper bounds need not exist in $[\text{MBFS}(\mathcal{C})]$, just as they need not exist for ordinary feature structures. However, just as in the case of ordinary feature structures, when an upper bound of two MBFS's exists, then so too does a least upper bound (lub). Furthermore, this lub may be efficiently computed by algorithm. We now turn our focus to this algorithm.

The Unification of Multiple-Base Feature Structures

1.14 The idea of the unification algorithm The idea of the unification algorithm has already been sketched (for ordinary feature structures) in 1.1. The algorithm for MBFS's is completely analogous, except that we start by pairing each pair of correspondingly labelled states, rather than by pairing the single initial states of ordinary feature structures.

The unification algorithm for feature structures is well known; details may be found in the paper [AN86], among others. Our extension to MBFS's is straightforward. However, as we will use this algorithm as a building block for our unification algorithm for Horn Extended MBFS's, we present it in considerable detail. In this presentation, we have specifically in mind to provide a framework which can be expanded into our more comprehensive algorithm for MBFS's with constraints. Therefore, the representation given here may not be the most economical or elegant as a self-standing algorithm. Throughout this paper, we refer to this algorithm as *Algorithm 1*.

Throughout, we use a Pascal-like description for our data structures and algorithms. However, we also take certain liberties in using high-level operations not part of standard Pascal. In addition, we deviate somewhat from Pascal notation in a few places. Particularly, we use the Algol notation `ref <type>` to denote a *reference type* (in implementation terms, *pointer type*) to the type `<type>`, rather than the Pascal notation `^<type>`. Also in keeping with the Algol convention, we always assume that sufficient dereferencing takes place so that things make sense. Thus, for example, if we have

```
type r = record
    field: ...
```

```

        end record;
var x: ref r
begin
  x.field := ...
end;

```

then x is *dereferenced* from type `ref r` to type `r` so that the assignment makes sense.

1.15 The underlying data structures The context $\mathcal{C} = (\mathcal{E}, \mathcal{V}, \mathcal{B})$ consists of three ordered sets, which we represent as three ordered types. (The exact order is immaterial; it is used solely for efficient identification and retrieval.) For simplicity, we may take these ordered types to be `integer`, but all that we absolutely require is that there be a total ordering available. These types are shown in Figure 1.8.

```

type base_label = <ordered_type>;
type edge_label = <ordered_type>;
type node_label = <ordered_type>;

```

Figure 1.8: The data types defining a context $\mathcal{C} = (\mathcal{E}, \mathcal{V}, \mathcal{B})$.

The heart of an MBFS is represented as an interconnection of nodes and edges, as represented in Figure 1.9. The representation of an edge has two essential components, its label and a reference to the next node, *i.e.*, the node to which the edge points. A third field, `eq_cl_link`, is used to identify the equivalence class of a partition in which the node resides, and will be discussed further below. The representation of a node also contains two essential components. Firstly, with each node, an ordered list of the outgoing edges is maintained. The `key` field states that the `label` field of an edge is the key to be used in ordering the list. Secondly, a type is associated with each node. The type `labelled` means that an atomic value labels that node. For example, the node x_4 in Figure 1.1 is labelled with a . The type `complex` means that the list of outgoing edges of the node is nonempty. The node x_1 in Figure 1.1 is of that type. The type `unlabelled` is reserved for nodes with no outgoing edges and no label, such as node x_5 in Figure 1.1. The type *nodetype* provides these three types, as well as a field giving the atomic label, if it exists.

In describing a machine, it is also necessary to identify the base labels. The type `label_ref` provides a base label for a node. An MBFS is then described by a list of objects of type `label_ref`, one for each label.

1.16 Supporting algorithms The key supporting algorithms for the unification algorithm are the union and find algorithms. These algorithms are very well known, so we only present their calling structure, as shown in Figure 1.10. For more details, a particularly good presentation, complete with working Pascal programs, may be found in [MS91, Sec. 3.2].

The algorithms support the maintenance of an equivalence relation on the set of nodes of an MBFS. The procedure `eq_class_find` takes as input the node

```

type edge = record
  label:      edge_label;
  next_node: ref node;
end record; {edge}

type node = record
  outgoing_edges: list_of(ref edge;
                        key: label);
  type:          nodetype;
  eq_cl_link:    ref node;
end record; {node}

type base_nodetype = (unlabelled, labelled, complex);

type nodetype = record
  base: base_nodetype;
  label: node_label; {Used only if base is of type labelled.}
end record;

type label_ref = record
  base_label: base_label;
  data:       ref node;
end record;

type MBFS = record
  start_nodes: list_of(label_ref;
                      key: base_label)
end record;

```

Figure 1.9: The data types for representing nodes, edges, and MBFS's for Algorithm 1.

```

procedure eq_class_find(in_node: ref set_node;
                      out_node: ref set_node);

procedure eq_class_union(in_node_1,
                       in_node_2: ref node;
                       which_is_root: 1..2);

```

Figure 1.10: The union-find procedure forms.

identified by `in_node`, and provides as output a canonical representative `out_node` from the equivalence class in which that node resides. We may thus test to see if two nodes are in the same equivalence class by running `eq_class_find` on each and testing to see whether or not the same representative is provided in each case. The procedure `eq_class_union` takes two equivalence class representatives and forms a new equivalence class which is the union of the two old ones. The third argument `which_is_root` indicates which of the two representatives is the representative for the new, merged equivalence class.

The field `eq_cl_link` of the data structure `edge` is used to support this equivalence relation structure. The details need not concern us here, save that the field will reference the node itself if and only if it is the representative of its equivalence

class. Thus, in starting with an equivalence relation in which each node is in a class by itself, all such fields are initialized to reference themselves.

These operations may be realized with an amortized time complexity which is almost linear in the size of the set of nodes. Specifically, if the number of nodes is n , then starting with the partition in which each node is in its own equivalence class, and performing $m \geq n$ finds and $n - 1$ unions, the amortized complexity is $O(m \cdot A(n, m))$, where $A(\cdot, \cdot)$ is an inverse Ackermann function which is, for all practical purposes, bounded by 5. See [Tar75] for details.

The algorithm also requires a merging operation on lists of edges. The basic form of the merging procedure is as shown in Figure 1.11 below. The first two

```

procedure merge(in_1,
               in_2,
               out: list_of(ref edge);
               &options);

```

Figure 1.11: Form of the merge procedure.

arguments are the lists to be merged, while the third identifies the resulting merged list. The `&options` specifies options on the nature of the merge. Specific cases will be described as they occur. Again, we note that supporting a list structure on a set of edges requires (at least) one extra field on the `edge` data type. As data structures which support such operations are well known, we do not expand upon them here, save to remark that numerous realizations exist in which merging, with or without the deletion of duplicates, can be performed in time linear in the sum of the lengths of the inputs.

We also use a `queue` data type, which is so well known that we do not document it further.

1.17 The main algorithm The main algorithm follows the general ideas of the unification algorithm of Aït-Kaci [AN86], so we only make a few remarks. Initially, all nodes are placed into their own equivalence classes (by setting each `eq_cl_link` field to reference the node itself). The operation `newqueue(node_pairs)` initializes a new, empty queue of pairs of nodes. The procedure `unify_MBFS` is the main procedure which unifies two MBFS's; it starts by enqueueing the corresponding starting nodes, and subsequently invoking `main_loop`. This latter procedure deletes pairs from the queue and, if they are in different equivalence classes, calls `coalesce_nodes` to place them in the same class. This procedure is in turn supported by two other procedures. Function `type_meet`, shown in Figure 1.13, just computes the type of a merged node from the types of its arguments. It also sets the flag `success` to false in case there is a failure because of multiple atomic labels, or an atomic label and an outgoing edge from the same node. (An actual implementation would handle failure more gracefully. We assume the existence of an external dæmon which halts the unification process when `success` is set to false.) The procedure `merge_edge_sets`

is invoked from `coalesce_nodes` for the purpose of merging the set of edges of two coalesced nodes into one list. In that procedure, the special arguments to the call to `merge` indicate that whenever two elements are found with the same keys, only the element from the first list is retained. In other words, duplicates are eliminated. Then, the pair of nodes found by following the matched edge labels is enqueued into `node_queue`. When the algorithm terminates successfully, the MBFS represented is the unification of the two input arguments.

It is interesting to note that the queue need not be a “true” queue. Rather, the elements may be deleted in any order. As long as all pairs which need to be coalesced are eventually processed, this may be done in any order.

Finally, if we wish to enforce the constraint that no two distinct nodes have the same atomic label, then, in the initial enqueueing, in addition to node pairs with like base labels, we also include pairs of nodes with the same atomic label. Otherwise, no changes need to be made.

We close this section with a quick sketch that the algorithm just presented indeed computes the unification of two MBFS’s, and that its complexity is acceptable.

1.18 Theorem — correctness of the algorithm *The algorithm is correct. More precisely, we have the following.*

- (a) *If the algorithm terminates with the variable `success` true, then it generates a least upper bound of its arguments under the ordering \sqsubseteq .*
- (b) *The algorithm succeeds iff its arguments have an upper bound.*

PROOF: First assume that the algorithm terminates with `success true`. Let $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$ and $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$ be the two input MBFS’s (assuming without loss of generality that $Q_1 \cap Q_2 = \emptyset$, so that we do not get name collisions), and let $M = (Q, \delta, \alpha, \beta)$ be the output MBFS. First we show that there is a morphism $f : M_i \rightarrow M$ for $i = 1, 2$. Let R be the equivalence relation on $Q_1 \cup Q_2$ computed by the algorithm. The states of M are the equivalence classes of R , and so we may define functions $\iota_i : M_i \rightarrow M$ for $i = 1, 2$ by $\iota_i : q \mapsto [q]_R$, where $[q]_R$ denotes the equivalence class containing q in R . It is immediate from the construction of the algorithm that both ι_1 and ι_2 satisfy conditions (mor-i) and (mor-iii). Condition (mor-ii) is guaranteed in view of the action of the type meet operation defined in Figure 1.13. Thus, ι_1 and ι_2 are morphisms of MBFS’s, and so $M_i \sqsubseteq M$ for $i = 1, 2$. Thus, the algorithm generates an upper bound of M_1 and M_2 , provided such a bound exists.

To show that M is a *least* upper bound, let $M_3 = (Q_3, \delta_3, \alpha_3, \beta_3)$ be a third MBFS, and let $f_i : M_i \rightarrow M_3$, $i = 1, 2$ be morphisms. In other words, assume that $M_1, M_2 \sqsubseteq M_3$. It suffices to show that $M \sqsubseteq M_3$, *i.e.*, that there is a morphism $h : M \rightarrow M_3$.

Let R be the equivalence relation on $Q_1 \cup Q_2$ computed by the algorithm. Since the states of M are the equivalence classes of R , it suffices to show that if $(q_1, q_2) \in R$ with $q_1 \in Q_i$ and $q_2 \in Q_j$ (here i and j may either be the same or different), then $f_i(q_1) = f_j(q_2)$. The proof proceeds by induction. For the basis, note that for $\ell \in \mathcal{B}$ with $\beta_1(\ell)$ and $\beta_2(\ell)$ defined, we must have that $f_1(\beta_1(\ell)) = \beta_3(\ell) = f_2(\beta_2(\ell))$. Now

```

var  node_pairs: queue;
     newqueue(node_pairs);
     success: Boolean;

procedure unify_MBFS(M1, M2: MBFS)
var  e1, e2: ref label_ref;
begin
  success := true;
  for each e1 in M1.start_nodes and e2 in M2.start_nodes
    with e1.base_label = e2.base_label do
      enqueue(node_pairs, (e1.data, e2.data));
    end foreach;
  main_loop();
end;

procedure main_loop();
var  current_pair: array[1..2] of ref node;
begin
  while not empty(node_pairs) do
    begin
      dequeue(node_pairs, current_pair);
      eq_class_find(current_pair[1], rep1);
      eq_class_find(current_pair[2], rep2);
      if rep1 <> rep2
        then coalesce_nodes(rep1, rep2)
      end
    end
  end;
end;

procedure coalesce_nodes(n1,
                        n2: ref node);
var  which: 1..2;
     root, other: ref node;
     newtype: nodetype;
begin
  newtype := type_meet(n1.type, n2.type);
  eq_class_union(n1, n2, which);
  if which = 1 then
    root := n1; other := n2
  else
    root := n2; other := n1
  end if;
  merge_edge_sets(root, other);
  root.type := newtype;
end;

procedure merge_edge_sets(root,
                          other: ref node);
var  e1, e2: ref edge;
begin
  merge(root.outgoing_edges, other.outgoing_edges, root.outgoing_edges,
        process_duplicates by
          with equal keys e1 in root.outgoing_edges,
                       e2 in other.outgoing_edges do
            include only e1 in merge;
            enqueue(node_pairs, (e1.next_node, e2.next_node));
          end do;
        )
end;

```

Figure 1.12: The main routines of Algorithm 1.

```

function type_meet(a1,
                  a2: nodetype): nodetype;
var a: nodetype;
begin
  case a1.base:
    unlabelled:
      type_meet := a2;
    labelled:
      if a2.base = unlabelled
        then type_meet := a1
      else if a2.base = labelled
        then if a2.label = a1.label
          then type_meet := a1
          else success := false;
          end if;
        end if;
      end if;
  complex:
    begin
      if ((a2.base = unlabelled) or (a2.base = complex))
        then
          a.base := complex;
          type_meet := a;
        else
          success := false;
        endif
    end;
end;

```

Figure 1.13: The type meet operation of Algorithm 1.

suppose that $(q_1, q_2) \in R$ with $f_i(q_1) = f_j(q_2)$ with i, j as defined above. Then, by the very definition of a morphism, for any $e \in \mathcal{E}$, $f_i(\delta_i(q_1, e)) = \delta_3(f_i(q_1), e) = \delta_3(f_j(q_2), e) = f_j(\delta_j(q_2), e)$. Since R is constructed by repeatedly adding pairs of the form $((\delta_i(q_1), e), \delta_j(q_2), e))$ to the initial relation, it follows that $f_i(q_1) = f_j(q_2)$ for all $(q_1, q_2) \in R$ with $q_1 \in Q_i$ and $q_2 \in Q_j$. Hence we may define a morphism $h : M \rightarrow M_3$ by $[q] \mapsto f_i(q)$, where $i = 1$ if $q \in Q_1$ and $i = 2$ if $q \in Q_2$. Thus $M \sqsubseteq M_3$, as was to be shown.

One direction of part (b) has already been established explicitly. Namely, if the algorithm does terminate with **success** true, then we know that the input MBFS's have an upper bound, namely their unification. The other direction has been established implicitly by the proof. Assume that M_1 and M_2 have an upper bound. Then there is an MBFS M_3 and morphisms $f_i : M_i \rightarrow M_3$. We have already shown above that there is then a morphism $M \rightarrow M_3$, provided that the algorithm terminates successfully. Suppose it fails. Then run it to completion anyway, with multiple types on some nodes. Upon completion, there will be some state of M , which, when regarded as a set of states from $Q_1 \cup Q_2$, will contain representatives q_1 and q_2 , and q_1 labelled with an atomic value a_1 and q_2 either labelled with an atomic value a_2 or else with an outgoing edge. Assume that $q_1 \in Q_i$ and $q_2 \in Q_j$, with $i, j \in \{1, 2\}$. Now by the above argument, we must have that $f_i(q_1) = f_j(q_2)$. But this is impossible unless q_1 and q_2 are type compatible, since any atomic label and/or outgoing edge must be carried to M_3 . Hence the unification must have terminated with **success** true, and the unification succeeded. \square

1.19 Theorem — the complexity of the algorithm *Let $M_1 = (Q_1, \delta_1, \alpha_1, \beta_1)$ and $M_2 = (Q_2, \delta_2, \alpha_2, \beta_2)$ be MBFS's, with n the total number of states in M_1 and M_2 and ϵ the total number of distinct edge labels which occur in M_1 and M_2 . Then the worst case time-complexity of the unification algorithm is $O(n \cdot \epsilon \cdot A(n \cdot \epsilon, n))$, where $A(-, -)$ is an inverse Ackermann function which grows incredibly slowly — for all practical purposes, $A(n \cdot \epsilon, n) \leq 4$.*

PROOF: There are two operations which dominate the algorithm. The first is merging edge sets, as identified in the procedure `merge_edge_sets`, and the second is the invoking of union and find in the procedure `main_loop`. Now in the worst case, there will be $n - 1$ unions performed. Each union will result in a merge, which, in the worst case, may involve at least a fixed percentage of all the ϵ edge labels. Thus, the total merge time in the worst case will be on the order of $(n - 1) \cdot \epsilon$, which is $O(n \cdot \epsilon)$. The total number of calls to `eq_class_find` is twice the number of node pairs which are enqueued, as can be seen from the procedure `main_loop`. Enqueued node pairs can arise from two sources. First, the initialization of procedure `unify_MBFS` can result in at most $\lfloor n/2 \rfloor$ enqueued pairs. Second, each call to `merge_edge_sets` can result in ϵ enqueued pairs, and there can be at most $n - 1$ such calls, on for each call to `coalesce_nodes`. Thus, the total number of finds is bounded by $n/2 + 2 \cdot n \cdot \epsilon$, which is $O(n \cdot \epsilon)$. Hence the union-find complexity, by the result of Tarjan [Tar75], is $O(n \cdot \epsilon \cdot A(n \cdot \epsilon, n))$, in which A is an inverse Ackermann function. Thus we have that the entire algorithm is $O(n \cdot \epsilon \cdot A(n \cdot \epsilon, n))$ in the worst case. \square

In practical terms, this means that the algorithm is $O(n \cdot \epsilon)$, since the $A(n \cdot \epsilon, n)$ term grows so slowly as to be insignificant.

It is important to remark that some authors claim that the complexity of the entire algorithm is $O(n \cdot A(n, n))$. However, this assumes that the number of distinct edge labels is bounded (*i.e.*, that \mathcal{E} is a finite set), which we have not done. If this number is bounded, then each merge takes constant time.

2. Multiple-Base Feature Structures with Constraints

In this section, we turn to the issue of describing constraints on MBFS's. Our discussion is broken into three subsections. In the first, we examine constraints and genericity in a general setting, without recourse to logic. In the second, we present a clause-based logic for MBFS's which parallels the logic of Rounds and Kasper. Finally, in the third subsection, we investigate Horn clauses within this logic, and establish the connections between such clauses and genericity.

Extending the Notion of a Multiple-Base Feature Structure to Include Constraints

2.1 An alternate perspective on unification As discussed in the introduction, a feature structure (and an MBFS as well) has two dimensions. On the one

hand, it is a record-like data structure, with (nested) attributes and values for those attributes. This is a *necessity* component, identifying the information which must be true regardless of the final result. In this way, it represents specific, definite knowledge. On the other hand, when viewed as an extensible structure, it provides a description of the possibilities which may evolve as new information becomes available. This is the *possibility* component. Making these ideas more precise, let M be an MBFS, and define $\text{Ext}(M) = \{N \in \text{MBFS}(\mathcal{C}) \mid M \sqsubseteq N\}$. The set $\text{Ext}(M)$ represents the set of possibilities into which M may grow. Notice that $\text{Ext}(M)$ determines M up to isomorphism; indeed, M must be one of the (isomorphic) least elements of $\text{Ext}(M)$ under the ordering \sqsubseteq . We furthermore have the following important characterizations.

2.2 Proposition *Let M_1 and M_2 be MBFS's.*

- (a) $M_1 \sqsubseteq M_2$ iff $\text{Ext}(M_2) \subseteq \text{Ext}(M_1)$.
- (b) $M_1 \sqcup M_2$ exists iff $\text{Ext}(M_1) \cap \text{Ext}(M_2) \neq \emptyset$.
- (c) *In the case that $M_1 \sqcup M_2$ does exist, it is given by any least element of $\text{Ext}(M_1) \cap \text{Ext}(M_2)$ under \sqsubseteq . (All such least elements must be isomorphic.)*

PROOF: Part (a) is immediate. To see (b), it suffices to note that M_1 and M_2 have an upper bound iff $\text{Ext}(M_1) \cap \text{Ext}(M_2) \neq \emptyset$, and then to apply 1.18(b). Since $\text{Ext}(M_1) \cap \text{Ext}(M_2)$ is the set of all upper bounds of both M_1 and M_2 , it follows from 1.18 (a) and (b) that it will have a least upper bound, which is given by their unification. \square

2.3 Extended MBFS's As the previous discussion illustrates, the essence of an MBFS M is recaptured entirely by its set of extensions $\text{Ext}(M)$, since M itself is the least element of this set of extensions. Thus, taking the point of view that an MBFS is a set of possibilities, we may provide an equivalent definition in which an MBFS is a subset of $\text{MBFS}(\mathcal{C})$ of the form $\text{Ext}(M)$. This suggests a natural generalization to MBFS's with constraints – rather than working just with classes of the form $\text{Ext}(M)$, we allow any subclass of $\text{MBFS}(\mathcal{C})$. More precisely, define an *extended MBFS* (or *XMBFS* for short) over \mathcal{C} to be any subclass \mathbf{K} of $\text{MBFS}(\mathcal{C})$ which is closed under isomorphism. We write $\text{XMBFS}(\mathcal{C})$ to denote the class of all extended MBFS's over \mathcal{C} .

Subsumption of extended MBFS's is defined in the way motivated by 2.2(a). More precisely, if \mathbf{K}_1 and \mathbf{K}_2 are extended MBFS's, we say that \mathbf{K}_1 *subsumes* \mathbf{K}_2 , and write $\mathbf{K}_1 \sqsubseteq_x \mathbf{K}_2$, if $\mathbf{K}_2 \subseteq \mathbf{K}_1$. Unification is defined as follows.

$$\mathbf{K}_1 \sqcup_x \mathbf{K}_2 = \begin{cases} \mathbf{K}_1 \cap \mathbf{K}_2 & \text{if } \mathbf{K}_1 \cap \mathbf{K}_2 \neq \emptyset; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

In light of 2.2(b) and 2.2(c) above, this generalizes unification of ordinary MBFS's. Furthermore, since we are just defining subsumption as class containment, it is clear that $\mathbf{K}_1 \sqcup_x \mathbf{K}_2$ is the least upper bound of \mathbf{K}_1 and \mathbf{K}_2 , whenever this bound exists.

2.4 Genericity and stability The preceding definition ignores the issue of a least element. There is no reason to expect, in general, that an arbitrary XMBFS will have a least element (up to isomorphism). In one sense of knowledge representation, this is a desirable property. The lack of a unique least element means that there are minimal alternatives, *i.e.*, that the XMBFS contains positive disjunctive information. Given the great interest and need for disjunction in feature structures, this may, in a knowledge representation sense, be viewed as a distinct advantage. However, it is also a drawback for at least two reasons. Firstly, we lose one of the two properties associated with ordinary feature structures; namely, that in addition to representing a set of possibilities, it also represents a single, definite structure which may be taken to be the result of the processing up to the current point. Secondly, as we shall see shortly, it also implies that unification will be a vastly more complex operation in terms of the time resources required. Therefore, it at least behooves us to ask how far the notion of an MBFS can be extended while retaining the property of being able to extract a least structure. Disjunction, if necessary, may then be built on top of this representation.

Call an extended MBFS \mathbf{K} *weakly generic* if it contains a least element under \sqsubseteq , up to isomorphism. In this case, we write $\text{LeastElt}(\mathbf{K})$ to denote the least element of \mathbf{K} , and may write $\langle \text{LeastElt}(\mathbf{K}), \mathbf{K} \rangle$ to emphasize that we are working with an extended MBFS with a least element. $\text{WeakGen}(\mathcal{C})$ denotes the class of all weakly generic extended MBFS's over \mathcal{C} .

Call a subset $\mathcal{G} \subseteq \text{XMBFS}(\mathcal{C})$ *stable* if it is closed under unification in the sense that if $\mathbf{K}_1, \mathbf{K}_2 \in \mathcal{G}$ and $\mathbf{K}_1 \sqcup_x \mathbf{K}_2$ exists in $\text{XMBFS}(\mathcal{C})$, then it is a member of \mathcal{G} . We wish to identify the largest “reasonable” stable subset of $\text{WeakGen}(\mathcal{C})$. (As we shall discuss shortly, we must add some reasonableness conditions to avoid undesirable anomalies.) It is clear that $\text{WeakGen}(\mathcal{C})$ itself is not stable. Figure 2.1 provides a simple example, in which \mathbf{K}_1 consist of the three structures in the left column, and \mathbf{K}_2 consist of the three structures on the right (plus all isomorphic copies). All are assumed to have the same base label ℓ . It is easy to see that the top element of each column is least in its respective set. Yet the intersection of \mathbf{K}_1 and \mathbf{K}_2 consists of the bottom two elements in each column, and neither subsumes the other. (Note the structure rooted at $x11$ is isomorphic to that rooted at $y11$, as is the pair rooted at $x21$ and $y21$ respectively.) Hence $\mathbf{K}_1 \sqcup_x \mathbf{K}_2$ is not weakly generic.

Weak genericity simply guarantees that the given set of possibilities has a least element; it says nothing about any extensions. We seek a robustness which preserves the least model property under extensions. The appropriate definition is the following. Call an extended MBFS \mathbf{K} *strongly generic* if for any $M \in \text{MBFS}(\mathcal{C})$, the set $\{N \in \mathbf{K} \mid M \sqsubseteq N\}$ is either empty or else weakly generic. We write $\text{StrongGen}(\mathcal{C})$ to denote the class of all strongly generic extended MBFS's over \mathcal{C} . The following lemma provides an alternate way of thinking of strong genericity. The proof is an immediate consequence of the definition.

2.5 Lemma *An extended MBFS \mathbf{K} is strongly generic iff for each MBFS M , if there is an $N \in \mathbf{K}$ such that $M \sqsubseteq N$, then there is a least such N , up to isomorphism. \square*

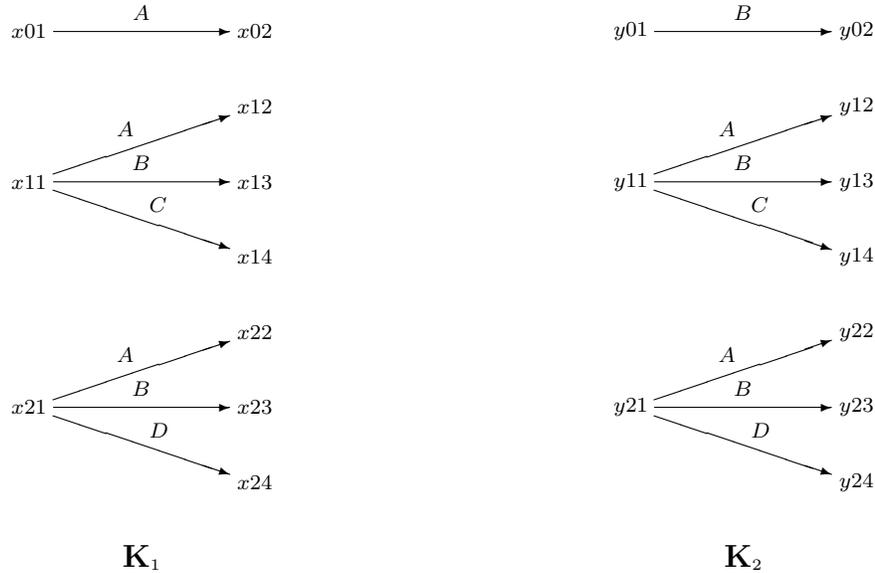


Figure 2.1: Two weakly generic XMBFS's whose unification is not weakly generic.

We can think of the N identified in the above lemma as a sort of completion of M within \mathbf{K} . To make this explicit, if $M \in \text{MBFS}(\mathcal{C})$, we write $\text{Completion}(M, \mathbf{K})$ to denote the least element of \mathbf{K} which M subsumes, when such an element exists. Intuitively, what this says is that whenever a parsing process delivers an MBFS M as an “unconstrained” object, we can always apply the constraints (represented by \mathbf{K}) to obtain a *unique* semantically legal MBFS which is closest to M , and which involves no arbitrary choices.

As it stands, the property of strong genericity is neither necessary nor sufficient to guarantee that a class of XMBFS's is stable. However, it is “close” to being an accurate characterization, modulo certain anomalies which we must rule out. To show the anomaly which prevents it from being necessary, consider the two-element set of XMBFS's consisting of \mathbf{K}_1 of Figure 2.1, together with the set consisting of the single MBFS rooted at $x01$, *i.e.*, the set consisting of only the single MBFS in the upper left-hand corner of Figure 2.1. This set is clearly stable. The problem is that we have not included enough other MBFS's to cause a problem. It turns out that “enough” is very little and very reasonable. Specifically, call an extended MBFS \mathbf{K} *syntactic* if it is of the form $\text{Ext}(M)$ for some MBFS M , and let $\text{SyntXMBFS}(\mathcal{C})$ denote the class of all syntactic XMBFS's. In words, a syntactic XMBFS is just the interpretation of an ordinary MBFS as an XMBFS. It is obvious that every syntactic XMBFS is strongly generic, since it contains all possible extensions of each of its elements. We then have the following.

2.6 Proposition *Let \mathcal{G} be a stable subset of $\text{XMBFS}(\mathcal{C})$ with the property that $\text{SyntXBMFS}(\mathcal{C}) \subseteq \mathcal{G}$. Then $\mathcal{G} \subseteq \text{StrongGen}(\mathcal{C})$.*

PROOF: Assume that \mathcal{G} is stable, and contains all syntactic XMBFS's. Let $\mathbf{K} \in \mathcal{G}$. Then, in particular, for each $M \in \text{MBFS}(\mathcal{C})$, it must be the case that whenever $\mathbf{K} \cap \text{Ext}(M)$ is nonempty, it has a least element. But this is just the property of strong genericity, and so the result is proved. \square

Thus, any stable class which includes all ordinary MBFS's must consist entirely of strongly generic MBFS's, and so under this condition, strong genericity is a necessary condition for stability. Unfortunately, it is not sufficient, as the following example shows.

2.7 Example Let P_i and Q_i be the MBFS's depicted in Figure 2.2. Each

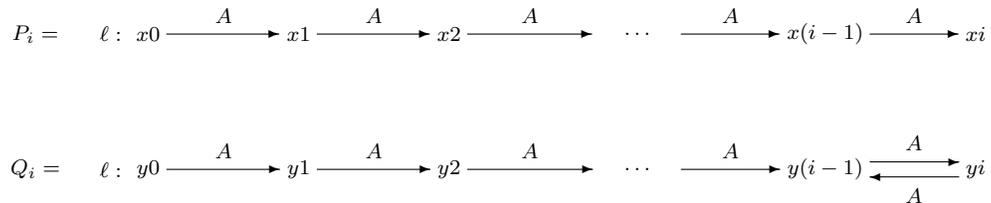


Figure 2.2: The families of XMBFS's for the example of 2.7.

XMBFS P_i has a single loop-free path of length i from its single base node, with each edge along that path labelled with A . The XMBFS Q_i is identical to P_i , save that it has one more edge which is looped back to the source of the previous edge. The following subsumptions are readily verified.

$$\begin{array}{l}
 P_i \sqsubseteq P_j \quad \text{iff } i \leq j \\
 Q_i \sqsubseteq Q_j \quad \text{iff } j \leq i \\
 P_i \sqsubseteq Q_j \quad \text{for all } i, j
 \end{array}$$

Now define $\mathbf{K}_1 = \{P_i \mid i \text{ is odd}\} \cup \{Q_i \mid i \geq 2\}$ and $\mathbf{K}_2 = \{P_i \mid i \text{ is even}\} \cup \{Q_i \mid i \geq 2\}$. Upon noting that any MBFS M which subsumes infinitely many Q_j 's must subsume some P_i , it is easily verified that both \mathbf{K}_1 and \mathbf{K}_2 are strongly generic. Yet $\mathbf{K}_1 \cap \mathbf{K}_2 = \{Q_i \mid i \geq 2\}$ is not strongly generic, since for M_o , the XMBFS with a single node labelled ℓ and no edges, there is no least XMBFS N in $\mathbf{K}_1 \cap \mathbf{K}_2$ with $M_o \sqsubseteq N$. Indeed, $M_o \sqsubseteq Q_i$ for each $i \geq 1$, and since $Q_j \sqsubseteq Q_i$ for $i \leq j$, Q_i cannot be N .

The above example is anomalous in the sense that the unification of \mathbf{K}_1 and \mathbf{K}_2 contains an infinite descending subsumption chain without a least element. To complete our characterization and obtain a sufficient condition for stability, we must introduce a means of avoiding such chains. Our approach is to use logic to characterize XMBFS's. Our final characterization of stability, which is formalized in 2.21, gives precise conditions for a logically defined XMBFS to be stable.

A Logic for Multiple-Base Feature Structures

The property of strong genericity is abstract. To have any hope of manipulating XBMFS's efficiently by algorithm, we need a succinct language for expressing constraints. Logics are precisely such languages. Thus, we seek to identify a logic whose models are XMBFS's. There has already been much related work, the earliest of which was due to Kasper and Rounds [RK86], who introduced a logic for representing ordinary feature structures. More recently, it has been shown that more sophisticated logics can be used to great advantage in manipulating feature structures. For example, Johnson [Joh91] has shown how a certain decidable subset of first-order logic is applicable in this context, and Blackburn [Bla91] has shown how a particular modal logic may be used to advantage. However, for our purposes, a relatively minor modification of the original logic of Kasper and Rounds appears to be most appropriate. Of course, we must provide the (minor) extension necessary to accommodate the node labels required to define multiple bases. The formal definition of syntax is the following.

2.8 Feature formula syntax All of the strings in these definitions are taken over the alphabet $\mathcal{V} \cup \mathcal{E} \cup \mathcal{B} \cup \{[,], (,), :, \top, \perp, \asymp, \vee, \wedge, \neg\}$.

With respect to a fixed context $\mathcal{C} = (\mathcal{E}, \mathcal{V}, \mathcal{B})$, we define the set $\text{Atoms}(\mathcal{C})$ of *feature atoms* to be the smallest set of strings which is closed under the following operations.

- (fa-i) The symbol $\top \in \text{Atoms}(\mathcal{C})$.
- (fa-ii) The symbol $\perp \in \text{Atoms}(\mathcal{C})$.
- (fa-iii) For each $l \in \mathcal{B}$ and each $\omega \in \mathcal{E}^*$, $l[\omega] \in \text{Atoms}(\mathcal{C})$.
- (fa-iv) For each $l \in \mathcal{B}$, each $\omega \in \mathcal{E}^*$, and each $a \in \mathcal{V}$, $l[\omega : a] \in \text{Atoms}(\mathcal{C})$.
- (fa-v) For each $l_1, l_2 \in \mathcal{B}$ and each $\omega_1, \omega_2 \in \mathcal{E}^*$, $l_1[\omega_1] \asymp l_2[\omega_2] \in \text{Atoms}(\mathcal{C})$.

The set $\text{Literals}(\mathcal{C})$ of *feature literals* over \mathcal{C} is the smallest class of strings closed under the following operations.

- (fl-i) $\text{Atoms}(\mathcal{C}) \subseteq \text{Literals}(\mathcal{C})$.
- (fl-ii) For each $\varphi \in \text{Atoms}(\mathcal{C})$, $(\neg\varphi) \in \text{Literals}(\mathcal{C})$.

A feature literal as defined by (fl-i) is termed *positive*, while one defined by (fl-ii) is *negative*. A feature literal of a clause φ is called a *disjunct* of that clause.

The set of $\text{Clauses}(\mathcal{C})$ *feature clauses* over \mathcal{C} is the smallest class of strings closed under the following operations.

- (fc-i) $\text{Literals}(\mathcal{C}) \subseteq \text{Clauses}(\mathcal{C})$.
- (fc-ii) For each $\varphi_1, \varphi_2 \in \text{Clauses}(\mathcal{C})$, $(\varphi_1 \vee \varphi_2) \in \text{Clauses}(\mathcal{C})$.

The set of $\text{Formulas}(\mathcal{C})$ *feature formulas* over \mathcal{C} is the smallest class of strings closed under the following operations.

- (ff-i) $\text{Clauses}(\mathcal{C}) \subseteq \text{Formulas}(\mathcal{C})$.

(ff-ii) For each $\varphi_1 \in \text{Formulas}(\mathcal{C})$, $\varphi_2 \in \text{Clauses}(\mathcal{C})$, $(\varphi_1 \wedge \varphi_2) \in \text{Formulas}(\mathcal{C})$.

A feature clause which is a constituent of a feature formula is called a *conjunct* of that formula.

Unlike most presentations of logical syntax, we have described our formulas directly in clausal form. Indeed, our feature formulas are already in *conjunctive normal form*. There is no essential theoretical need to do this, since clauses could be defined as special forms of a more general class of formulas, and it is well known that any formula in any standard logic may be placed into conjunctive normal form [Dav89, pp. 26–27]. However, since our main focus will be in addressing constraints defined by clauses, it seems most appropriate to restrict our formulas to such forms from the very start.

We will discuss further differences with the Kasper-Rounds logic once we have given the semantics.

We will take the same liberties in dropping parentheses that are typically taken in logics. Thus, for example, $((\neg\varphi_1) \vee (\varphi_2 \vee (\neg\varphi_3)))$ may be abbreviated to $(\neg\varphi_1 \vee \varphi_2 \vee \neg\varphi_3)$.

2.9 Feature formula semantics Let $M = (Q, \delta, \alpha, \beta) \in \text{MBFS}(\mathcal{C})$ and let $\varphi \in \text{Formulas}(\mathcal{C})$. We write $M \models \varphi$ to mean that M is a *model* of the formula φ , or that “ M satisfies φ .” Similarly, we write $M \not\models \varphi$ to mean that M is not a model of φ . The following eight rules provide the formal definition of satisfaction.

- (sat-i) $M \models \top$ always holds.
- (sat-ii) $M \models \perp$ never holds.
- (sat-iii) $M \models \ell[\omega]$ iff $\beta(\ell) \downarrow$ and $\delta^*(\beta(\ell), \omega) \downarrow$.
- (sat-iv) $M \models \ell_1[\omega_1] \succ \ell_2[\omega_2]$ iff $\beta(\ell_1) \downarrow$, $\beta(\ell_2) \downarrow$, $\delta(\beta(\ell_i), \omega_i) \downarrow$, and $\delta(\beta(\ell_1), \omega_1) = \delta(\beta(\ell_2), \omega_2)$, for $i = 1, 2$,
- (sat-v) $M \models \ell[\omega : a]$ iff $\beta(\ell) \downarrow$, $\delta^*(\beta(\ell), \omega) \downarrow$, and $\alpha(\delta^*(\beta(\ell), \omega)) = a$.
- (sat-vi) $M \models (\neg\varphi)$ iff $M \not\models \varphi$.
- (sat-vii) $M \models (\varphi_1 \vee \varphi_2)$ iff $M \models \varphi_1$ or $M \models \varphi_2$.
- (sat-viii) $M \models (\varphi_1 \wedge \varphi_2)$ iff $M \models \varphi_1$ and $M \models \varphi_2$.

If Φ is a set of feature formulas, we write⁴ $M \in \text{Mod}(\Phi)$ to mean that $M \models \varphi$ for each $\varphi \in \Phi$. When Φ is a singleton $\{\varphi\}$, we may write $\text{Mod}(\varphi)$ for $\text{Mod}(\{\varphi\})$. If Ψ is another set of feature formulas, we write $\Phi \models \Psi$ just in case $\text{Mod}(\Phi) \subseteq \text{Mod}(\Psi)$. For a single formula ψ , we may also write $\Phi \models \psi$ for $\Phi \models \{\psi\}$. A set Φ of feature formulas is *consistent* if $\text{Mod}(\Phi)$ is nonempty. Two sets of feature formulas Φ_1 and Φ_2 are *logically equivalent* if both $\Phi_1 \models \Phi_2$ and $\Phi_2 \models \Phi_1$ hold.

In comparison to the logic of Kasper and Rounds [RK86], ours has only minor, inessential differences. Of course, we have had to add a bit of syntax to accommodate the semantics of the base labels. Beyond that, we incorporate two other

⁴As is common practice, we give the symbol \models double duty. In $A \models B$, A may be either a structure or else a set of formulas. Context will always make clear which it is.

economies. We disallow nested path expressions of the form $\ell[\omega_1 : \omega_2 : \dots : \omega_n]$, since such an expression is logically equivalent to $\ell[\omega_1\omega_2\dots\omega_n]$. Our eventual focus is upon algorithms which operate upon such formulas, and so we wish to make formula parsing as easy as possible. Also, we only allow for binary path equivalences, written $\ell_1[\omega_1] \asymp \ell_2[\omega_2]$, whereas Kasper and Rounds allow for arbitrary finite sets of equivalent paths to expressed in one atom. Again, this is merely a syntactic convenience, as we may represent any n -ary equivalence using $n - 1$ binary equivalences. Of course, as noted above, we also restrict our attention to conjunctive normal form, while Kasper and Rounds do not make this restriction.

2.10 Logical extended multiple-base feature structures Our entire purpose for introducing feature formulas is to have a formal means for describing XMBFS's. The idea is simple. Let \mathbf{K} be an extended MBFS. We say that \mathbf{K} is a *logical extended MBFS* (or *LoXMBFS* for short) if $\mathbf{K} = \text{Mod}(\Phi)$ for some consistent finite set Φ of feature clauses. The finiteness is enforced for the purposes of admitting an algorithm, since algorithms can only operate on finite specifications. The symbol $\text{LoXMBFS}(\mathcal{C})$ denotes the class of all LoXMBFS's over the context \mathcal{C} .

2.11 Unification algorithms for LoXMBFS's In the light of 2.2(b), it follows immediately that determining whether or not two arbitrary LoXMBFS's $\mathbf{K}_1 = \text{Mod}(\Phi_1)$ and $\mathbf{K}_2 = \text{Mod}(\Phi_2)$ are unifiable amounts to determining whether or not $\Phi_1 \cup \Phi_2$ has a model; that is, whether or not it is satisfiable. Now Kasper and Rounds have shown that the decision problem for satisfiability in their logic is NP-complete [RK86, Thm. 4]. As their proof puts a formula into conjunctive normal form, which our formulas are in automatically, it takes only a trivial syntactic modification to their proof to show that satisfiability for our feature formula logic has an NP-complete decision problem as well. Thus, the best known algorithm for testing satisfiability is exponential in the size of the input, and so computationally intractable. As we noted in the discussion in the first paragraph of 2.4, there are two reasons for enforcing genericity. The first is that a least element is provided. The other is that a tractable unification is available. The key which we need to show is that strong genericity of LoXMBFS's is equivalent to their being describable by special class of sentences known as Horn clauses. We now turn to the development of this result.

Horn Clauses in the Context of Feature Logic

2.12 The definition of implication Let $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_{k_1}\}$ and $P = \{\rho_1, \rho_2, \dots, \rho_{k_2}\}$ be finite sets of feature atoms. We define

$$(\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_{k_1}) \Rightarrow (\rho_1 \vee \rho_2 \vee \dots \vee \rho_{k_2}) \quad (\text{rf})$$

to be an abbreviation for the clause

$$(\neg\sigma_1 \vee \neg\sigma_2 \vee \dots \vee \neg\sigma_{k_1} \vee \rho_1 \vee \rho_2 \vee \dots \vee \rho_{k_2}), \quad (\text{df})$$

and we call formula (rf) the *rule form* of the clause (df), which is said to be in *disjunctive form*. The set Σ is called the *antecedent set* and P the *consequent set* of the clause (in either form). Formally, a feature clause is *Horn* if in rule form, its set of consequents has at most one element. Equivalently, a clause in disjunctive form is Horn if at most one of its disjuncts is a positive literal. Essentially, then, a Horn clause is one in which there is no disjunction among the consequents. If the antecedents are satisfied, then a definite fact becomes true, rather than one of a set of alternatives. Look back to items (H1) through (H4) of the introduction for a review of the taxonomy of Horn clauses.

2.13 Horn feature clauses and linguistic knowledge Recasting the examples (L1) through (L5) of the introduction in a specific context for the representation of linguistic knowledge would require that we develop an appropriate context, complete with semantics, and is beyond the scope of this work. However, to illustrate the variety of forms which rules may take, let us recast the first four of those five examples as formal rules. (Example (L5) does not add anything interesting, so we omit any further elaboration of it.) In all of these examples, let ℓ be the root node of the context currently under consideration.

(Example L1 continued) Let ω_{sp} be the path defining the person of the subject and let ω_{vp} be the path defining the person of the verb. The corresponding rule then becomes

$$\ell[\omega_{sp}] \wedge \ell[\omega_{vp}] \Rightarrow \ell[\omega_{sp}] \succ \ell[\omega_{vp}].$$

(Example L2 continued) Let ω_{vt} be the path defining the type of the verb, and let ω_{do} be the path defining a direct object. Then the rule defining the constraint is

$$\ell[\omega_{vt} : \text{transitive}] \Rightarrow \ell[\omega_{do}].$$

(Example L3 continued) Let ω_{vt} be the path defining the type of the verb, and let ω_{do} be the path defining a direct object. Then the rule defining the constraint is

$$\ell[\omega_{vt} : \text{intransitive}] \wedge \ell[\omega_{do}] \Rightarrow \perp$$

(Example L4 continued) Let ω_{sr} define the path of the subject referent attribute, and let ω_{or} define the path of the object referent. Let ω_{ppt} define the path of the form of the possessive object pronoun. The rule then takes the following form.

$$\ell[\omega_{sr}] \succ \ell[\omega_{or}] \Rightarrow \ell[\omega_{ppt} : \text{reflexive}]$$

Notice particularly that path equivalences may occur either as antecedents (L2) or as consequents (L4) of rules. Indeed, there are *no* restrictions on the types of atoms which may occur in Horn formulas, nor in whether they may occur as antecedents or as consequents.

The specialization of 2.10 to Horn clauses is the following.

2.14 Horn extended multiple-base feature structures Let \mathbf{K} be an XMBFS. We say that \mathbf{K} is a *Horn extended MBFS* (or *HoXMBFS* for short) if there is a finite set Φ of Horn feature clauses such that $\mathbf{K} = \text{Mod}(\Phi)$. The symbol $\text{HoXMBFS}(\mathcal{C})$ denotes the class of all HoXMBFS's over the context \mathcal{C} .

The Genericity of Horn Clauses

In 2.7, we provided an example which shows that the unification of two strongly generic XMBFS's need not be strongly generic. We now show that this cannot happen if we restrict attention to strong genericity within the context of LoXMBFS's. The main result, given in 2.22 below, states that the HoXMBFS's are precisely the strongly generic LoXMBFS's. Many of the ideas which we use in this part of the paper parallel those of Makowsky in [Mak87, Sec. 1].

2.15 Sets of atoms and generic models For $M \in \text{MBFS}(\mathcal{C})$, define $\text{Facts}(M) = \{p \in \text{Atoms}(\mathcal{C}) \mid M \in \text{Mod}(p)\}$. In words, $\text{Facts}(M)$ is the set of all atoms which are true for the MBFS M .

Let Φ be a set of feature clauses. Define $\text{AtomicCon}(\Phi) = \{p \in \text{Atoms}(\mathcal{C}) \mid \Phi \models p\}$. This set is called the set of *atomic consequences* of Φ .

Let Φ be a set of feature clauses, and let $M \in \text{MBFS}(\mathcal{C})$. We say that M is a *generic model* for Φ if $M \in \text{Mod}(\Phi)$ and $\Phi \models \text{Facts}(M)$.

The following lemma is an immediate consequence of the above definitions.

2.16 Lemma *Let Φ be a finite set of features clauses, and let $M \in \text{MBFS}(\mathcal{C})$. Then M is a generic model for Φ iff M is the least element under \sqsubseteq (up to isomorphism) of $\text{Mod}(\Phi)$. \square*

2.17 Proposition *Let Φ be a finite set of feature clauses, and let $M \in \text{MBFS}(\mathcal{C})$. Then M is a generic model for Φ iff $\text{Facts}(M) = \text{AtomicCon}(\Phi)$.*

PROOF: First suppose that M is a generic model for Φ . Then it is immediate that $\text{Facts}(M) \subseteq \text{AtomicCon}(\Phi)$. On the other hand, by the above lemma, $M \sqsubseteq N$ for each $N \in \text{Mod}(\Phi)$. Thus, for any $p \in \text{AtomicCon}(\Phi)$, we must have that $M \models p$. Hence $\text{AtomicCon}(\Phi) \subseteq \text{Facts}(M)$, and so $\text{Facts}(M) = \text{AtomicCon}(\Phi)$.

Conversely, suppose that $\text{Facts}(M) = \text{AtomicCon}(\Phi)$. Then it is immediate that $M \in \text{Mod}(\Phi)$. And, since $\Phi \models \text{AtomicCon}(\Phi)$, we have that $\Phi \models \text{Facts}(M)$ as well. Hence M is a generic model for Φ . \square

There is a slight problem in working directly with $\text{Facts}(M)$, since if M has a cycle (*i.e.*, a state q and a nonempty string ω for which $\delta^*(q, \omega) = q$), then $\text{Facts}(M)$ will be an infinite set. Since we want definitions which are compatible with algorithms, we seek a finite representation. Fortunately, it suffices to work only with those facts which deal with strings which are no longer than the number of states in M , since the collection of such facts completely characterizes M . We formalize this as follows.

2.18 Definitions — finiteness of MBFS's (a) Let $\varphi \in \text{Atoms}(\mathcal{C})$. Define $\text{PathLength}(\varphi)$ as follows.

- (i) $\text{PathLength}(\top) = \text{PathLength}(\perp) = 0$.
 - (ii) For $\ell \in \mathcal{B}$, $\omega \in \mathcal{E}$, $\text{PathLength}(\ell[\omega]) = \text{Length}(\omega)$. (Here $\text{Length}(\omega)$ denotes the length of ω regarded as a string.)
 - (iii) For $\ell \in \mathcal{B}$, $\omega \in \mathcal{E}$, $a \in \mathcal{V}$, $\text{PathLength}(\ell[\omega : a]) = \text{PathLength}(\ell[\omega])$.
 - (iv) For $\ell_1, \ell_2 \in \mathcal{B}$, $\omega \in \mathcal{E}^*$, $\text{PathLength}(\ell_1[\omega_1] \succ \ell_2[\omega_2]) = \max(\{\text{PathLength}(\ell_1[\omega_1]), \text{PathLength}(\ell_2[\omega_2])\})$.
- (b) Let $M = (Q, \delta, q_o, \alpha) \in \text{MBFS}(\mathcal{C})$. Define $\text{BaseFacts}(M) = \{p \in \text{Facts}(M) \mid \text{PathLength}(p) \leq \text{Card}(Q)\}$, with $\text{Card}(Q)$ denoting the number of states in Q .

2.19 Lemma *Let $M \in \text{MBFS}(\mathcal{C})$. Then $\text{Facts}(M)$ and $\text{BaseFacts}(M)$ are logically equivalent.*

PROOF: The proof is immediate, since from $\text{BaseFacts}(M)$ we can obtain a complete description of M , up to isomorphism. \square

We then have the following strengthening of 2.17.

2.20 Proposition *Let Φ be a finite set of feature clauses, and let $M \in \text{MBFS}(\mathcal{C})$. Then M is a generic model for Φ iff $\text{BaseFacts}(M)$ is logically equivalent to $\text{AtomicCon}(\Phi)$.*

PROOF: Follows immediately from 2.17 and 2.19. \square

2.21 Admitting generic models Let Φ be a finite set of feature clauses. We say that Φ *admits generic models* if for every finite set Δ of feature atoms, if $\Phi \cup \Delta$ is consistent, then it has a generic model. Note in particular that Δ may be of the form $\text{BaseFacts}(M)$ for some MBFS M , so that this property is effectively strong genericity for LoXMBFS's.

2.22 Theorem — characterization of genericity *A finite set Φ of feature clauses admits generic models iff it is logically equivalent to a finite set of Horn clauses.*

PROOF: First of all, assume that Φ is a finite set of Horn clauses. We apply the following procedure. Let A be an initially empty set of feature atoms. Repeatedly apply the following two rules in any order, until no more application is possible.

- (r1) If a clause has an empty antecedent set, then add the consequent literal, if any, to the set A . (If a clause has an empty antecedent set and an empty consequence set, then the set of clauses is inconsistent, and the procedure fails.)
- (r2) If a clause contains an antecedent atom which is a logical consequence of the current set A , then delete that atom from the set of antecedents.

If the procedure succeeds, let M be the least MBFS for which all the atoms in A are true. (Just build the MBFS directly from the specifications of the atoms.) It is easy to see that M is a least model for Φ . Hence, by 2.20, Φ admits generic models.

Conversely, assume that Φ is a finite set of clauses which admits generic models. Let $\varphi \in \Phi$, with $\varphi = (\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_{k_1}) \Rightarrow (\rho_1 \vee \rho_2 \vee \dots \vee \rho_{k_2})$. Without loss of generality, we may assume that if we delete any antecedent or consequent from φ , the resulting clause is no longer a logical consequence of Φ . For if we may perform such a deletion and obtain a logical consequence of Φ , we obtain a clause which is stronger than the original, and so we may replace the original with this new clause. Now let $\Delta = \{\sigma_1, \sigma_2, \dots, \sigma_{k_1}\}$, and let M be the generic model of $\Phi \cup \Delta$. Then it is the case that $M \models \rho_1 \vee \rho_2 \vee \dots \vee \rho_{k_2}$. But then it must be the case that $M \models \rho_i$ for some i . Since M is generic, it must further be the case that $N \models \rho_i$ whenever $M \sqsubseteq N$. Hence, for any N which is subsumed by M , we may replace φ by $(\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_{k_1}) \Rightarrow \rho_i$. On the other hand, for any N which is not subsumed by M , $N \not\models (\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_{k_1})$, and so replacing φ by $(\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_{k_1}) \Rightarrow \rho_i$ will not affect whether or not such an N is in $\text{Mod}(\Phi)$. Hence we may replace φ with a Horn clause. Repeating this procedure for each clause in Φ , we obtain an equivalent set of Horn clauses, as required. \square

Finally, the following two corollaries formalize the result that Horn clauses provide the logical characterization of strong genericity.

2.23 Corollary *Let $\mathbf{K} \in \text{LoXMBFS}(\mathcal{C})$. Then \mathbf{K} is strongly generic iff it is of the form $\text{Mod}(\Phi)$ for some finite set Φ of Horn feature clauses.*

PROOF: Let Φ be a finite set of feature clauses, and let $M \in \text{MBFS}(\mathcal{C})$. It is clear that $\{N \in \text{Mod}\Phi \mid M \subseteq N\} = \text{Mod}(\Phi \cup \text{BaseFacts}(M))$. Since every finite set of atoms is logically equivalent to one of the form $\text{BaseFacts}(N)$ for some MBFS N , it follows that $\text{Mod}(\Phi)$ is strongly generic iff Φ admits generic models. But then the result follows from the above theorem. \square

2.24 Corollary *$\text{HoXMBFS}(\mathcal{C})$ is the largest strongly generic subset of $\text{LoXMBFS}(\mathcal{C})$ which contains $\text{SyntXBMFS}(\mathcal{C})$.*

PROOF: First of all, note that for a given MBFS M , $\text{Ext}(M) = \text{Mod}(\text{Facts}(M))$. Since $\text{BaseFacts}(M)$ is a finite set of Horn feature clauses, it follows that $\text{SyntXBMFS}(\mathcal{C}) \subseteq \text{HoXMBFS}(\mathcal{C})$. Secondly, observe that for any two sets Φ_1 and Φ_2 of Horn feature clauses, $\Phi_1 \cup \Phi_2$ is also a set of Horn feature clauses. Thus, since $\text{Mod}(\Phi_1) \sqcup_x \text{Mod}(\Phi_2) = \text{Mod}(\Phi_1) \cap \text{Mod}(\Phi_2) = \text{Mod}(\Phi_1 \cup \Phi_2)$, it follows that $\text{HoXMBFS}(\mathcal{C})$ is closed under unification. The result now follows from the above corollary. \square

3. The Inference Algorithm for Propositional Logic

Having established the desirability of working with HoXMBFS's, we now turn to the problem of their efficient unification. We have already defined one of the key building blocks of this algorithm, the unification algorithm for ordinary MBFS's of Section 1. The other building block is the fast unification algorithm for propositional Horn logic, which we present in this section. Because this algorithm has already been presented in the paper of Dowling and Gallier [DG84], we restrict our discussion to the components essential to understanding the extended algorithm of the next section. Throughout this paper, we refer to the algorithm of this section as *Algorithm 2*.

3.1 The principal data structures There are two data types which we must define, the atom and the clause, whose definitions are depicted in Figure 3.1. The algorithm maintains a counter for each clause indicating how many more of its antecedent literals must become true before the clause fires, *i.e.*, before its consequents can be declared to be true. This counter is represented by the `antecedent_count` field of the type `clause`, and is initialized to the number of distinct atoms in the antecedent part of the clause. Each clause record also contains a list of references to those atoms which become true when the clause fires. (For efficiency, we allow a clause to have multiple atoms in its consequent set. This is no violation of the theory, as a sentence of the form $(A \wedge B) \Rightarrow (C \wedge D)$ may be regarded as an abbreviation for two clauses, $(A \wedge B) \Rightarrow C$ and $(A \wedge B) \Rightarrow D$.)

The type `atom` contains the `clause_list` field, which lists those clauses which contain the atom in their antecedent sets. The `truth_value` records whether the atom is true or false, and all atoms are initialized to false.

```

type clause = record
  antecedent_count: natural_number; {Initialized to the number of
                                     distinct antecedents.}
  consequent: list_of(ref atom) {Empty list implies right-hand side = false.}
end;

type atom = record
  name: atom_name;
  clause_list: list_of(ref clause);
  truth_value: Boolean {Initialized to false.}
end

```

Figure 3.1: The principal data structures for Algorithm 2.

3.2 The main algorithm The main algorithm is a form of what is known as *forward chaining*. A queue `clause_queue` of clauses to be fired is maintained. Initially, the clauses placed in this queue (by `initialize`) are those with no antecedents, *i.e.*, facts. Then, as a clause fires, those other clauses which contain

the consequents of the fired clause as antecedents are told this information by `report_antecedent_count_decrease`, which decreases their `antecedent_count` field. When this field drops to zero, that clause is placed in the queue. There is no requirement on the order that clauses be dequeued for firing.

```

var queue: clause_queue;
    clauses: list_of(clause);
    consistent: Boolean;

procedure initialize();
var c: clause;
begin
    consistent := true;
    for each c in clauses with c.antecedent_count = 0 do
        enqueue(clause_queue,c)
    end foreach;
end; {initialize}

procedure main();
var c,v: ref clause;
    x: atom;
begin
    initialize();
    while ((not empty(clause_queue)) and consistent) do
        dequeue(clause_queue,c);
        if empty(c.consequent)
            then consistent := false
        else
            for each x in c.consequent
                if x.truth_value = false
                    then
                        x.truth_value = true;
                        for each v in x.clause_list do
                            report_antecedent_count_decrement(v);
                        end foreach;
                    end if;
            end foreach;
        end if;
    end while;
end; {main}

procedure report_antecedent_count_decrement(v: ref clause);
begin
    v.antecedent_count := v.antecedent_count - 1;
    if v.antecedent_count = 0 then
        enqueue(clause_queue,v)
    end if
end; {report_antecedent_count_decrement}

```

Figure 3.2: The main routines of Algorithm 2.

3.3 The complexity of the propositional inference algorithm A full discussion of the complexity may be found in [DG84]. For our purposes, the important thing to note is that the complexity depends upon how the atoms are stored and accessed. If we assume that the proposition names are natural numbers, so that we may arrange all of the objects of type `atom` in an array, then the algorithm runs in

time linear in the length of the input. (That is, the sum of the lengths of all of the clauses.) If the atoms are indexed by names, so we must store them in a symbol table, then the running time rises to $O(n \cdot \log(n))$, where n is the length of the input, since each access to the symbol table will take $O(\log(n))$ time.

It is this $O(n \cdot \log(n))$ complexity which is most significant for our purposes. In extending this algorithm to the logic of MBFS's, we will not make any initial assumptions about the number of distinct atoms which may appear in the formulas describing the constraints on a given system. We now turn to the development of the details of this extension.

4. The Full Unification Algorithm

In this section, we present the unification algorithm for HoXMBFS's. This algorithm will be referred to as *Algorithm 3*. The reader is particularly directed to 4.13 in the final subsection, which provides a detailed example of the unification process. The example is presented in such a way that it may be read before studying the details of the algorithm. It is suggested that the reader at least skim this example before tackling the algorithmic details.

Overview and High-Level Specification

4.1 An overview of the operation of the algorithm Since the details of the unification algorithm are somewhat involved, we begin with an informal description of how things work. In Algorithm 2, each proposition has a truth value associated with it. All propositions start out as false, and as rules fire, they become true. The key idea is that as propositions become true, that change in truth value is reported to clauses which contain those propositions as antecedents. In particular, a major data structure of Algorithm 2 is a vector of truth values for the propositions. In our unification algorithm for Horn feature clauses, we maintain a corresponding data structure. However, the situation is more complex than in the propositional case. For example, for any paths ω_1 and ω_2 and any label ℓ , the term $\ell[\omega_1]$ is always a logical consequence of the term $\ell[\omega_1\omega_2]$, i.e., $\ell[\omega_1\omega_2] \Rightarrow \ell[\omega_1]$ is a tautology. On the other hand, in propositional logic, no formula of the form $A \Rightarrow B$, with A and B distinct atomic propositions, is a tautology. What we need is an extension of Algorithm 2 which implicitly incorporates the semantics of feature logic. Looking back to the proof of 2.22 (which essentially represents a forward-chaining procedure for deduction), we need a way to implement (r2); that is, a way to tell whether or not an antecedent atom is a logical consequence of the currently deduced set of atoms. To accomplish this, we replace the vector of propositions and their truth values of Algorithm 2 with a specially augmented MBFS. More particularly, the "truth state" of the deduction is represented by an MBFS in which each edge, each node, and each atomic node attribute assignment is tagged as either true or false (which we term *actual* and *potential*, respectively). In such a *virtual feature structure*, the actual edges, nodes and node assignments are those which have been determined to

be part of the least feature structure of the final theory. The potential edges, nodes, and assignments are those which occur in atoms of the antecedents of some of the clauses, but which have not yet been determined to be true. When the deductive process is finished, the generic (least) model of the theory is defined by the actual part of the structure.

In Algorithm 2, the number of distinct antecedents is recorded for each clause in the `antecedent_count`. As these antecedents become true, the count is decreased accordingly. When the count reaches zero, the rule fires. In the algorithm of this section, instead of counting the number of antecedent atoms for each clause, for each antecedent set the number of edges and node attributes, as well as the number of certain kinds of nodes themselves, is counted, and duplicates are *not* eliminated. (It would be more trouble to eliminate duplicates than to count them again.) Thus, for example, an atom of the form $\ell[ABCA : a]$ would add 5 to the initial count, since it has four edges and one atomic node attribute a . The base label ℓ is not counted, since it is just a marker. The nodes along the path are not counted, since their existence is mandated by the edges which connect them. However, for a term of the form $\ell_1[\lambda]$, the node labelled by ℓ_1 would be counted, since it has no incident edges to define its existence. Also, if the term $\ell[ABD]$ occurred as an antecedent of the same clause, it would add 3 to the initial count, one for each edge, even though two of the edges are duplicates from the atom $\ell[ABCA : a]$.

The algorithm must also take into account that terms mandating the coalescing of two nodes may occur as both antecedents and as consequents of rules. If the firing of a rule mandates that two nodes be coalesced, a normal unification of the associated virtual feature structure, as described by Algorithm 1, is enabled. When two actual edges are merged, or if an actual edge is merged with a potential one, the resulting edge is actual. It is only when both edges of the merger are potential that the result is potential. A similar rule applies for coalesced nodes.

To handle rules with a path coalescing as an antecedent, we use special pairs of node tags. If we have a term of the form $\ell_1[\omega_1] \asymp \ell_2[\omega_2]$, then we tag the node at the end of each path with the same, unique marker. (The markers are internal to the virtual MBFS, and do not correspond to any part of an ordinary MBFS.) Whenever two nodes are coalesced, we check for matching pairs of such markers. If found, the coalescing is reported back to the appropriate rules. It is important to mention that when we have an antecedent of the above form, in addition to the markers, we also act exactly as if the two feature atoms $\ell_1[\omega_1]$ and $\ell_2[\omega_2]$ occur as antecedents. The reason is that the semantics of $\ell_1[\omega_1] \asymp \ell_2[\omega_2]$ is twofold: (i) both paths exist and (ii) the end points coalesce. If we just checked for the end markers agreeing, we would not know that both paths are actual. Thus, for the example feature atom $\ell_1[ABC] \asymp \ell_2[BCDE]$, the total initial count of items to be made actual is 8, three for the path of the first literal, four for the path of the second, and one for the path equivalence.

In the algorithm, the only potential edges, nodes, and attributes which need to be maintained are those which occur in antecedents of rules. The nodes, edges, and attributes which occur in the consequents of rules need not be added to the virtual MBFS until the corresponding rule actually fires. Of course, they could be added

as potential entities from the very start, but there is no advantage in so doing. By adding them only when necessary, we keep the size of the virtual MBFS as small as possible, which in turn helps keep the computational complexity as low as possible.

4.2 The process queue The structure of the overall process queue is shown in Figure 4.1. At the highest level, the basic operation which is repeated in Algorithm 3 is one of firing a clause. This is represented by a queue entry of type `fire_clause`. Firing a clause, in turn, makes each of its consequents true. If a consequent is of the form $\ell[\omega]$ or $\ell[\omega : a]$, then the type of operation is `actual_path`, which declares that a term is to be made actual in the virtual MBFS. For a consequent of the form $\ell_1[\omega_1] \asymp \ell_2[\omega_2]$, an operation of the form `coalesce_nodes` is required, as well as two operations of the form `make_actual_path`. Finally, in a unification process, we may need to add new data after the process has begun. A process of the type `add_potential_path` serves to support that purpose. It is similar to `make_actual_path`, except that the resulting path is potential instead of actual. As is the case in Algorithms 1 and 2, the process queue need not be a formal queue. A `dequeue` operation may select any element currently enqueued. The monotonicity of the associated operations assures us that the order in which operations are performed does not affect the final result.

```

type process_type = (fire_clause, coalesce_nodes,
                    make_actual_path, add_new_potential_path)

type process_queue_entry = record
  case type: process_type of
    fire_clause:      (clause_id: ref clause);
    coalesce_nodes:   (nodes: array[1..2] of ref node);
    make_actual_path: (path: path_expression);
    add_potential_path: (path: path_expression)
  end;

```

Figure 4.1: Definition of the process queue for Algorithm 3.

4.3 The principal data types The first three definitions of Figure 4.2 give the context, which is identical to that associated with Algorithm 1. The types `path_expression` and `path_equivalence` define the two types of feature atoms, and are largely self explanatory. The type `cond_node_label` may be either empty or a node label, as the end of the corresponding path expression requires.

Figure 4.3 contains the definitions of the most important data types. The type `edge` has the two attributes `label` and `next_node` which it inherited from the definition in Figure 1.9. It also has two new attributes not used in the algorithm of Section 1. The Boolean value `actual` records whether the edge is potential or actual, and `clause_list` is a list of clauses which have counted that edge in their antecedent counter. A clause may appear more than once in this list, since duplicate entries are allowed, as discussed in 4.1.

```

type base_label = <ordered_type>;
type edge_label = <ordered_type>;
type node_label = <ordered_type>;

type path_expression = record
  label:      base_label;
  edge_list:  list_of(edge_label);
  end_label:  cond_node_label;
end record; {path_expression}

type path_equivalence = array[1..2] of path_expression;

type cond_node_label = record
  case defined: Boolean
    false: ()
    true: (label_value: node_label)
  end record; {cond_node_label}

```

Figure 4.2: Context, path, and logical expression definitions for Algorithm 3.

A node has attributes `outgoing_edges`, `type`, and `eq_cl_link`, which serve exactly the same purpose as do the corresponding fields defined for Algorithm 1. `clause_list` is a new attribute, which plays the same rôle for the node that the corresponding attribute in the `edge` definition plays. Namely, it identifies the clauses whose antecedent truth values depend upon the existence and atomic value assignment for the node. The entries in this list are of type `clause_ref_node`, which identify both a clause and a possible atomic attribute value for that node. If an entry of type `clause_ref_node` appears in a `clause_list`, this means that the clause has an antecedent literal which requires that the atomic value identified by the `atomic_type` field be present at the identified node. If such an entry occurs more than once in the `clause_list`, it means that the associated node reference occurred more than once in the set of antecedents for that clause. This is in agreement with our general strategy (as mentioned in 4.1) of treating duplicate references as distinct, rather than attempting to locate and eliminate them.

The attribute `pair_list` serves a similar function, but for clause antecedent feature atoms which involve path coalescence. An entry means that this node is one of two which are the endpoints of a path coalescing feature atom. The type `clause_literal_pair_ref_node` associates a pair identifier with particular literal within a clause. If two entries with the same `literal` field (a value of type `clause_ref_id_type`) are in the `pair_list` for a node, this signals that an antecedent of the form $\ell_1[\omega_1] \succ \ell_2[\omega_2]$ has been satisfied for that clause, because the appropriate nodes have been coalesced. Finally, the `actual` field is a flag which indicates whether the node is actual or potential. In the type `clause_literal_pair_ref_node`, the `literal` field is a unique identifier for a given antecedent literal of the form $\ell_1[\omega_1] \succ \ell_2[\omega_2]$ in a given clause. We use such identifiers, rather than just a reference to the clause, because it is possible that the same path equivalence may occur more than once as the antecedent of the same clause. In this case, distinct entries (with the same `clause_id` field but with dis-

```

type edge = record
  label:      edge_label;
  next_node:  ref node;
  actual:     Boolean;
  clause_list: list_of(ref clause)
end record; {edge}

type node = record
  outgoing_edges:  list_of(ref edge;
                        key: label);
  type:            nodetype;
  eq_cl_link:     ref node;
  clause_list:    list_of(clause_ref_node;
                        key: clause_id);
  pair_list:      list_of(clause_literal_pair_ref_node;
                        key: literal);
  actual:         Boolean; {Used only for group header nodes:}
end record; {node}

type base_nodetype = (unlabelled, labelled, complex);

type nodetype = record
  base:          base_nodetype;
  label:         cond_node_label;
end record; {nodetype}

type label_ref = record
  base_label:   base_label;
  data:         ref node;
end record; {label_ref}

type clause_ref_node = record
  clause_id:    ref clause;
  atomic_type: cond_node_label;
end record; {clause_ref_node}

type clause_literal_pair_ref_node = record
  clause_id:    ref clause;
  literal:     literal_ref_id_type;
end record; {clause_literal_pair_ref_node}

type literal_ref_id_type: natural_number;

type clause = record
  antecedent_count:  natural_number;
  path_consequents: list_of(path_expression);
  coalesce_consequents: list_of(path_equivalence);
end record; {clause}

```

Figure 4.3: The principal data types for Algorithm 3.

tinct `literal` fields) are made in the clause lists of each of the two associated nodes, one for each literal. When the two nodes are coalesced into one, one match will be found for each literal identifier, and the antecedent count for the corresponding clause will be decremented once for each corresponding literal. The values of type `literal_ref_id_type` are *globally* unique, rather than just unique for a given clause, so that a match can only occur once for each label. This is easily accomplished by maintaining a counter, initialized to zero, which is incremented each

time that a new value is needed. Thus, strictly speaking, duplicates do not occur in `pair_list`'s, since the global identifiers distinguish them. However, distinct identifiers may reference identical path expressions.

The type `clause` itself contains the counter `antecedent_count`, which is analogous to the field of the same name in the definition of `clause` in Algorithm 2. And while the definition of Algorithm 2 has only one list of consequents, the definition for Algorithm 3 contains two type of consequents, one for paths and one for node pairings. These correspond to the two types of things which can become true as the result of a rule firing; namely, paths (including end labels) can become actual and nodes can be coalesced.

Finally, the type `label_ref` is the same as in Algorithm 1.

4.4 The core of the algorithm The core of the algorithm consists of four procedures, one for firing a clause, one for coalescing nodes, one for realizing a new path, and one for merging edge sets. These are defined in Figures 4.4 through 4.7, respectively.

The clause-firing procedure in Figure 4.4 is straightforward. The process of firing a clause `c` is twofold. First of all, all of the paths in its `path_consequents` list are made actual. Secondly, all the path equivalences in its `coalesce_consequents` are made true. This latter step includes both making the corresponding paths actual and initiating a unification on the end points of these paths.

```

procedure fire_clause(c: ref clause);
  var p: process_queue_entry;
      q: path_expression;
      n: array[1..2] of ref node;
      i: 1..2;
  begin
    p.type := make_actual_path;
    for each q in c.path_consequents do
      p.path := q;
      enqueue(process_queue,p);
    end foreach;
    for each r in c.coalesce_consequents do
      p.type := make_actual_path;
      for i := 1 to 2 do
        p.path := r[i];
        enqueue(process_queue,p);
        n[i] := end_node(r[i]); {n[i] := the last node in the path r[i].}
      end for;
      p.type := coalesce_nodes;
      p.nodes := n;
      enqueue(process_queue,p);
    end foreach;
  end;

```

Figure 4.4: The clause-firing procedure of Algorithm 3.

The procedure `coalesce_nodes` of Figure 4.5 is based upon the procedure of the same name in Algorithm 1, and is invoked whenever an item type `coalesce_nodes`

is dequeued from the process queue. Since this routine, unlike the one of Algorithm 1, may be given arguments which have already been coalesced, it contains a test to make sure that the arguments `n1` and `n2` have not already been placed in the same equivalence class. Other than that, the principal difference is that calls to `merge_clause_ante_lists`, `merge_check_pair_lists`, and `make_true_node_refs` have been inserted.

The procedure `merge_clause_ante_lists` merges the `clause_list`'s of the lists of the two nodes and makes that merger the clause list of the root node, since the root node of an equivalence class always contains all information about that equivalence class' participation in firing rules. As noted previously, duplicates are included in this merger. If a clause occurs more than once in the `clause_list` of a node, this means that the node occurred more than once on the left-hand side of the given clause in a fashion that needed to be reported. Such a fashion includes both the cases that the node had an explicit `node_label` value assigned to it or that it is the only node in a path of the form $\ell[\lambda]$. The procedure `merge_check_pair_lists` performs a similar function for the `pair_list`'s of the nodes. Because the `literal` field of such an entry is globally unique (as noted in 4.3), duplicate entries occur in such a list only for identical `literal` fields; that is, only for references to the same literal. Noting that `literal` is the key for the `pair_list` of an object of type `node` (see Figure 4.3), a key may occur at most twice, and such a duplication cites that the path equivalence *for a particular literal* has been satisfied. If that path equivalence occurs more than once, even in the same clause, there will be one citation (given by a unique `literal` reference) for each occurrence.

Finally, if the merger resulted in an actual node, then `make_true_node_refs` is called to report any consequences of making that node actual to the appropriate clauses. These three procedures are detailed in Figure 4.7. The procedure `type_meet` is exactly the same as that given in Figure 1.13, and so is not repeated here.

Figure 4.6 details the procedure for building a new path. This procedure is invoked when an item of type `make_actual_path` is dequeued from the process queue. This is a result of the firing of a rule which requires that a new path be made actual. The operations `first` and `rest`, operating on a list, correspond to the `car` and `cdr` operations of Lisp, and retrieve the first element from the list and everything but the first element, respectively. The `find` operation just searches a list for the given element. The `make_edge` routine builds a new edge record to the given specifications, and the `insert` routine inserts an element into a sorted list.

The procedure `merge_edge_sets` of Figure 4.7 is an extension of the corresponding routine in Algorithm 1. The rather complex `merge` call stipulates the following. The merger of the edge sets of `n1` and `n2` first requires that we merge the corresponding clause lists. These lists are merged *without* eliminating duplicates, because the same edge may participate in several feature terms in a given antecedent set. The resulting edge is actual if and only if one of the two participants is actual. If the edge is actual, then this is reported to the appropriate clauses via the routine `report_actual_edge`. Again, we note that if the same edge occurs more than once in a given list, this means that it occurred more than once in an antecedent

```

procedure coalesce_nodes(n1,
                        n2: ref node):
  var which: 1..2;
      root, other, p1, p2: ref node;
      newtype: nodetype;
begin
  newtype := type_meet(n1.type, n2.type);
  eq_cl_find(n1,p1);
  eq_cl_find(n2,p2);
  if p1 <> p2 then
    eq_class_union(p1,p2,which);
    if which = 1 then
      root := p1; other := p2
    else
      root := p2; other := p1
    end if ;
    merge_edge_sets(root, other);
    merge_clause_ante_lists(root, other);
    merge_check_pair_lists(root, other);
    root.actual := n1.actual or n2.actual;
    if root.actual then
      make_true_node_refs(root, newtype);
    end if;
  end if;
end;

```

Figure 4.5: The main procedure of Algorithm 3 for coalescing nodes.

literal for the same clause, and that we treat these as distinct occurrences, rather than attempting to eliminate them. We thus report that the edge became actual to the given clause one time *for each occurrence of that clause* in the corresponding `clause_list`.

Finally, as in the procedure of Algorithm 1, the end points of the two edges are enqueued for coalescing.

4.5 The support routines Figure 4.8 contains descriptions of some of the more important support routines for Algorithm 3. Since the actions of these routines have already been identified in the main discussion above, we will forego detailed descriptions of how they operate. Rather, we just note the point that `make_true_node_refs`, `merge_check_pair_lists`, and `report_actual_edge` all delete the corresponding list entries once they have been reported to the appropriate clauses. This is to prevent such reporting from occurring more than once, and thus resulting in an incorrectly low count and incorrect rule firing. Although duplicates are allowed in some cases, we only want one report for each entry.

4.6 Other details of the algorithm There are a few straightforward routines for which we have not supplied an algorithmic description. The most important is the initialization routine, which functions as follows. First of all, for each clause, an object of type `clause` is constructed. From that, the appropriate set of `path_expression`'s and `path_equivalence`'s are constructed. Then the procedure `build_path` is repeatedly invoked to realize each of the paths, with a separate rou-

```

procedure build_path(source:    ref node;
                    p:        edge_path;
                    end:      nodetype;
                    actual:    Boolean;
                    end_point: ref node;
                    clause     ref clause);
var x:              list;
    e, f:           edge;
    q:              edge_path;
    node, v:        ref node;
begin
    node := source;
    q := p;
    while q <> nil do
        f := first(q);
        q := rest(q);
        r := find(f, node.outgoing_edges);
        if r <> nil then {Find successful -- edge already exists.}
            if (actual and (not r.actual))
                then
                    r.actual := true; {Make hypothetical edge actual.}
                    report_actual_edge(r)
                end if;
            insert(r.clause_list, clause);
            node := r.next_node
        else {Build a new edge.}
            v := make_vertex(actual: actual); {New node for head of edge.}
            new_list(x); insert(x, clause);
            e := make_edge({actual:}      actual, {New edge itself.}
                          {label:}      f,
                          {next_node:}  v,
                          {clause_list:} x,
                          {set_link:}   nil,
                          {block_count:} 1);
            insert(node.outgoing_edges, e);
            node := v
        end if;
    end while;
    node.type := end; {Set the value of last node in path.}
    end_point := node;
end;

```

Figure 4.6: The main procedure of Algorithm 3 for realizing a new path.

tine providing the base labels, as necessary. For each path coalescing antecedent, an appropriate object of type `clause_literal_pair_ref_node` is constructed. And, finally, the process queue is initialized to contain all of the clauses which have no antecedents.

Another important operation is the one which adds a structure during the unification process. As we mentioned previously, one of the hallmarks of the unification-based approach is that new data can be added at any time. But the process of adding new data during the unification process is the same as the initialization process described in the previous paragraph.

Once the algorithm has terminated, we will want to extract the generic model. But this is easy, as that model is defined by all of the actual components of the virtual MBFS. The potential components are discarded because no rule fired which

```

procedure merge_edge_sets(n1,
                          n2: ref node);
var p: process_queue_entry;
begin
  merge(n1.outgoing_edges, n2.outgoing_edges, n1.outgoing_edges,
        process_duplicates by
        with equal keys e1, e2 do
          merge(e1.clause_list, e2.clause_list, e1.clause_list,
                include_duplicates);
          e1.actual := e1.actual or e2.actual;
          if e1.actual then report_actual_edge(e1);
          p.type := coalesce_nodes;
          p.nodes := (e1.next_node, e2.next_node);
          enqueue(process_queue,p)
        end do;
)
end;

```

Figure 4.7: The edge merging procedure of Algorithm 3.

made them actual.

Finally, if we wish to disallow the case that two distinct nodes carry the same atomic label, we must set up a symbol table containing all occurrences of *actual* atomic labels. Whenever a node with an atomic label becomes actual, or whenever an actual node receives a new atomic label, the symbol table is searched for another occurrence of an actual node with that label. If one is found, the two nodes are enqueued in the process queue for coalescing. If no such node is found in the symbol table, then an entry is made for that atomic value with the node with that label. Since all nodes with the same label must be coalesced, there is no need to make more than one entry in the table for any given atomic label.

Complexity

We now turn to determining a bound on the worst-case time complexity of Algorithm 3. Our goal here is only to establish a very rough upper bound to illustrate that the algorithm is tractable.

4.7 Complexity of list operations Algorithm 3 makes use of sorted lists in several places. Operations on such lists include merging, deleting, inserting, and finding an element based upon a key. By maintaining such lists as balanced trees, we can guarantee the following worst-case time complexities. Merging can be performed in time $O(n_1 + n_2)$, where n_1 and n_2 are the lengths of the lists to be merged. Deletion, insertion, and finding can be performed in time $O(\log(n))$, where n denotes the length of the list. A typical data structure for accomplishing this is a balanced tree, such as *red-black trees* [CLR90, Chap. 14] or [MS91, Sec. 3.3]. In these references, it is shown how the operations of deletion, insertion, and find may be performed in time $O(\log(n))$, in the worst case. However, there is no discussion of merging balanced trees in that book, or in any other of which we know. Therefore,

```

procedure merge_clause_ante_lists(root,
                                other: ref node);
begin
  merge(root.clause_list, other.clause_list,
        root.clause_list,
        include_duplicates);
end;

procedure make_true_node_refs(n: ref node;
                              t:  nodetype);
var r: clause_ref_node;
begin
  if ((t.base = labelled) or (t.base = unlabelled))
  then
    for each r in n.clause_list do
      if ((r.atomic_type = t.base) or (r.atomic_type = unlabelled))
      then
        report_antecedent_count_decrement(r.clause_id);
        delete(n.clause_list,r)
      end if;
    end foreach;
  end if;
end;

procedure report_antecedent_count_decrement(v: ref clause);
var p: process_queue_entry;
begin
  v.antecedent_count := v.antecedent_count - 1;
  if v.antecedent_count = 0 then
    p.type := fire_clause;
    p.clause_id := v;
    enqueue(process_queue,p)
  end if;
end;

procedure merge_check_pair_lists(root,
                                 other: ref node);
begin
  merge(root.pair_list, other.pair_list, root.pair_list,
        process_duplicates by
        with equal keys e1, e2 do
          report_antecedent_count_decrement(e1.clause_id);
          delete(root.pair_list,e1); delete(root.pair_list,e2);
        end do;
  )
end

procedure report_actual_edge(e: ref edge);
var c: ref clause;
begin
  for each c in e.clause_list do
    report_antecedent_count_decrement(c);
  end foreach;
  e.clause_list := nil;
end;

```

Figure 4.8: Support procedures for Algorithm 3.

we will briefly sketch how this operation may be performed in time proportional to the number of nodes in the two trees. We assume basic familiarity with the no-

tion of a binary search tree and tree traversal, and refer the reader to either of the above-mentioned references for elaboration. Let us mention, however, for the sake of clarity, that we follow the conventions of [CLR90] and always assume that the leaves of a binary search tree are empty, and that each non-leaf node has both a left and a right subtree. Thus, only interior nodes may contain data.

Firstly, a *red-black* tree is a binary search tree in which each node is classified as either *red* or *black*, subject to the following three rules.

(black rule) The root is black, as is each leaf node.

(red rule) The parent of a red node is always black.

(rank rule) Every path from a node to a descendant leaf has exactly the same number of black nodes.

Now, to build a single red-black tree T from two red-black trees T_1 and T_2 , we proceed as follows. First of all, we build a linear list which consists of the merger of the elements of T_1 and T_2 , sorted by key. This is clearly possible in time $O(n_1 + n_2)$, where n_i is the number of non-leaf nodes in T_i . Indeed, we may obtain the elements of any binary search tree in order in linear time by traversing the tree in inorder [CLR90, 13.1], and the process of merging can surely be performed in time linear in the size of the lists to be merged.

In a binary tree, define the natural index of a node as follows:

- (1.) The natural index of the root is 1.
- (2.) If a node has natural index k , then its left child, if it exists, has natural index $2k$.
- (3.) If a node has natural index k , then its right child, if it exists, has natural index $2k + 1$.

More informally, the natural indices just number the nodes, level-by-level, left-to-right. A binary tree T is *complete* [AKM81] if there is a natural number m such that T consists exactly of those nodes whose natural indices lie between 1 and m , inclusive.

Our next step is to build a complete binary tree with $n_1 + n_2$ *internal* nodes. Now, traverse this tree in inorder, and as non-leaf nodes are visited, assign to them, in order, the elements from the list built by merging the data from T_1 and from T_2 . The result is clearly a binary search tree. Since inorder traversal may be performed in linear time, this operation is linear. To make it a red-black tree, let p be the length of the longest path from the root to a leaf. Since the tree is complete, the shortest path from the root to a leaf must have length at least $p - 1$ [AKM81, 3.2.9]. Now color all nodes black except those interior nodes at level $p - 1$. In other words, color a node black except if it is an interior node, both of whose children are leaves, and for which there is no interior node at a greater distance from the root. It is easy to verify that the resulting search tree satisfies the red-black conditions, and so we have thus constructed a merger of T_1 and T_2 in linear time.

We will assume throughout the analysis that the lists are represented as red-black trees, so that the operations may be performed in the times indicated.

We now take a look at the complexity of each type of operation of the process queue.

4.8 Lemma *The worst-case time complexity for executing the procedure `build_path` is $O(l \cdot \log(\epsilon))$, where l is the length of the path and ϵ is the total number of distinct edge labels in the current virtual MBFS.*

PROOF: For each edge in the path, the only part of the computation which is not constant time is running the find operation to determine if there is already an edge by that name emanating from the designated node along the path. To determine that information, the collection of edges from that node must be searched. But in view of the assumptions regarding list operations identified in 4.6, this find can be done in time $O(\log(\epsilon))$. From this the result follows immediately. \square

Since both of the operation types `make_actual_path` and `add_potential_path` are implemented as calls to `build_path`, we immediately have the following.

4.9 Proposition *The worst-case time complexity of an operation of type `make_actual_path` or `new_virtual_path` is $O(l \cdot \log(\epsilon))$, where l is the length of the path and ϵ is the total number of distinct edge labels in the current virtual MBFS. \square*

Now we turn to the `coalesce_nodes` operation. In analyzing this algorithm, we can obtain a more informative bound by measuring *amortized complexity*, that is, the complexity over all calls of an entire execution of the algorithm from start to finish, rather than focusing on one particular unification. In understanding the statement of the proposition below, note that a call to `coalesce_nodes` can only effect a call to `make_actual_path` as the result of firing a clause whose antecedent set has become true. When we say *for the associated unification* below, we are referring to all of the enqueued processes resulting from calls to `merge_edge-sets`, but *not* to any operations resulting from clause firings. Thus, the only enqueued operations for the associated unification will be of type `coalesce_nodes`.

4.10 Proposition *The worst case amortized time complexity for executing all instances of the procedure `coalesce_nodes` for the associated unification is $O(n \cdot (\tau + \epsilon \cdot A(n \cdot \epsilon, n)))$, where n is the total number of nodes and ϵ is the total number of distinct edge labels in the virtual MBFS, τ is the total number of feature atoms in all of the clauses with duplicate feature atoms counted once for each occurrence, and $A(-, -)$ is an inverse Ackermann function.*

PROOF: First of all, note that the *total* number of unions, over the entire life of the algorithm operating on one set of clauses, is bounded by $n - 1$, just as in Algorithm 1. Furthermore, the number of finds which are the result of coalescing node pairs, as opposed to finds mandated by the firing of rules which add expressions of the form $\ell_1[\omega_1] \asymp \ell_2[\omega_2]$ to the set of true atoms, is $O(n \cdot \epsilon)$, just as in Algorithm 1. Thus, the running time of the part of Algorithm 3 encompassing just the is $O(n \cdot \epsilon \cdot A(n \cdot \epsilon, n))$, as shown in 1.19.

Now consider how Algorithm 3 differs from Algorithm 1 in the procedure `merge_edge_sets` and the routines which it calls. Each call to `merge_edge_sets` results in not only a merger of the two edge sets (as in Algorithm 1), but also in a second merger of the corresponding clause lists for each pair of edges which are combined in the merge. Since there are at most ϵ distinct edge labels, there can be up to ϵ secondary mergers of clause lists for each primary merger of edge sets. But each of the τ feature atoms can occur in at most one of the secondary mergers, and so the amortized complexity of the secondary mergers over the (at most) ϵ primary mergers is $O(\tau)$. Since there are at most $n - 1$ primary mergers (one for each union operation), the amortized complexity of all mergers is $O(n \cdot \epsilon) + O(n \cdot \tau)$. Now the procedure `merge_clause_ante_lists` merges the antecedent lists for the two nodes. But since only the end node of a path needs to be identified in an antecedent count, there is at most one entry per feature atom. In the worst case, a merge may always involve a fixed percentage of all of the atoms. Therefore, the worst case time for a particular call is $O(\tau)$. Since this procedure may be called once for each of the at most $n - 1$ calls to the procedure `coalesce_nodes` in which n_1 and n_2 are in distinct equivalence classes, the total amortized complexity is $O(n \cdot \tau)$. Finally, note that a call to `report_antecedent_count_decrement` takes but constant time, so a similar bound may be established for the procedure `merge_check_pair_lists`. The procedure `make_true_node_refs` runs in constant time. Hence the entire amortized complexity is $O(n \cdot \epsilon \cdot A(n \cdot \epsilon, n)) + O(n \cdot \epsilon) + O(n \cdot \tau) = O(n \cdot (\tau + \epsilon \cdot A(n \cdot \epsilon, n)))$, as advertised. \square

Finally, consider the time it takes to build the initial virtual MBFS from a list of clauses. Define the *length* of a clause to be the total number of symbols in these string which defines it (and not just the number of feature atoms).

4.11 Proposition *Let Φ be a set of feature clauses. Then the virtual MBFS associated with Φ may be constructed in worst-case time $O(L \cdot \log(\epsilon))$, where L is the sum of the lengths of all of the clauses in Φ , and ϵ is the number of distinct edge labels in the associated virtual MBFS.*

PROOF: The construction technique is completely straightforward, repeatedly invoking the procedure `build_path`. The result then follows from 4.7. \square

Finally, we can piece all of these results together for the final result.

4.12 Theorem *Let Φ be a set of feature clauses, and let M be the initial virtual MBFS associated with Φ . Define the following numbers.*

L = total length of Φ .

n = total number of nodes in M .

ϵ = total number of distinct edge labels in M .

τ = total number of feature atoms in all of the clauses, with duplicates counted once for each occurrence.

Then a worst case time-complexity bound on the running time of Algorithm 3 is $O(L \cdot \log(\epsilon) + n \cdot (\tau + \epsilon \cdot A(n \cdot \epsilon, n)))$, where $A(-, -)$ is an inverse Ackermann function.

PROOF: Follows from 4.9, 4.10, and 4.11. \square

It is remarkable how little the asymptotic computational complexity has increased over that of Algorithm 1. Indeed, the $O(L \cdot \log(\epsilon))$ term represents the minimal time necessary to build an MBFS from the clauses, and we cannot possibly hope for anything better. The $O(n \cdot \epsilon \cdot A(n \cdot \epsilon, n))$ term of 1.19 has been replaced by $O(n \cdot (\tau + \epsilon \cdot A(n \cdot \epsilon, n)))$, which is biased by the number of atoms in the clauses. In effect, the second term jumps from $O(n \cdot \epsilon)$ to $O(n \cdot (\tau + \epsilon))$, the latter being no worse than quadratic in the size of the input.

An Annotated Example

To close, we present an example of the unification of HoXMBFS's. This example is adapted from the one we presented in [Heg91]. This example is designed to be read before and while studying Algorithm 3 as well as after; therefore, we have tried to make enough of the explanation independent of the details of the notation of that algorithm.

4.13 An example theory and extended feature graphs The set Ξ_1 contains the following ten Horn feature clauses.

$$(\xi_1) \ell_1[AA : a].$$

$$(\xi_2) \ell_1[B : a].$$

$$(\xi_3) \ell_1[AA : a] \wedge \ell_1[B : a] \Rightarrow \ell_1[CCDDG : t].$$

$$(\xi_4) \ell_1[A] \wedge \ell_1[C] \Rightarrow \ell_1[ABDDG] \wedge \ell_1[B] \succ \ell_1[AA].$$

$$(\xi_5) \ell_1[AA] \succ \ell_1[B] \wedge \ell_1[ABDDG] \Rightarrow \ell_1[ABDDEF].$$

$$(\xi_6) \ell_1[ABDD] \wedge \ell_1[B] \Rightarrow \ell_1[CCD] \succ \ell_1[ABD].$$

$$(\xi_7) \ell_1[CCDD] \succ \ell_1[ABDD] \Rightarrow \ell_1[AC].$$

$$(\xi_8) \ell_1[ACD] \Rightarrow \ell_1[ACC : t].$$

$$(\xi_9) \ell_1[CCAB] \wedge \ell_1[CCCD] \Rightarrow \ell_1[CCABC] \succ \ell_1[CCCDE].$$

$$(\xi_{10}) \ell_1[B : b] \Rightarrow \perp.$$

This is our initial set of Horn feature clauses. The corresponding virtual MBFS (called an *extended feature graph* in [Heg91]) is shown in Figure 4.9. Notice that only the paths which occur on the left-hand sides of the rules are included in the initial virtual MBFS. While it is not a problem for correctness to include those which occur on the right-hand sides (as we did in [Heg91]), there is no advantage in so doing. Rather, it is advantageous to keep the virtual MBFS as small as possible, and only add edges and nodes as they are needed. Initially, all nodes and

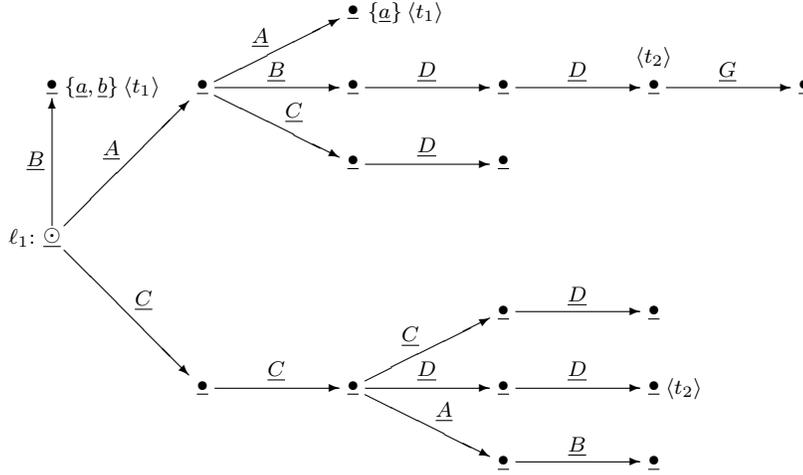


Figure 4.9: The initial virtual MBFS for the example of 4.13.

edges are potential. As a notational convention we mark the labels of potential edges and potential atomic node labels with an underscore, and the bullets marking potential nodes are themselves underscored. Base-labelled nodes are represented by an encircled dot. Nodes which have one or more potential atomic labels have that set of labels representing them next to the bullet. Note that a node may have more than one potential atomic label, as illustrated by the node at the end of the path $\ell_1[B]$, which has a label set consisting of two elements: $\{\underline{a}, \underline{b}\}$. There is no contradiction here, as a node may have many potential atomic labels. On the other hand, by the definition of an MBFS, it may have at most one actual atomic label.

The elements in angle brackets, *e.g.*, $\langle t_1 \rangle$ and $\langle t_2 \rangle$, are coalesce tags, and are entered in the pair list for that node. When two nodes are coalesced, a check is made for matching tags, which indicates that a clause antecedent defining a path equivalence has been satisfied. (These tags are formally part of objects of type `clause_literal_pair_ref_node`.)

Also note that clause ξ_4 has two consequents. In effect, we have represented two clauses in one. This is a convenience which not only provides for compact notation, but also improves the efficiency of the algorithm (although not the asymptotic complexity). Formally, ξ_3 is equivalent to the two clauses $\ell_1[A] \wedge \ell_1[C] \Rightarrow \ell_1[ABDDG]$ and $\ell_1[A] \wedge \ell_1[C] \Rightarrow \ell_1[B] \asymp \ell_1[AA]$.

Initially, entries for firing clauses ξ_1 and ξ_2 are entered in the process queue, since these are the two clauses whose antecedent sets are trivially satisfied. These two may fire in any order; there is no ordering at all implied for the process queue operations. After both fire, both antecedents of clause ξ_3 become true, and so a fire-clause entry for ξ_3 is entered into the process queue. After clause ξ_3 fires, its consequent $\ell_1[CCDDG : t]$ becomes true with the resulting virtual MBFS shown in Figure 4.10. In this depiction, in addition to removing the underscores on objects which have become actual, we have used bold lines for those edges which are actual.

Notice that the node at the end of the path has one actual atomic label, a , and one

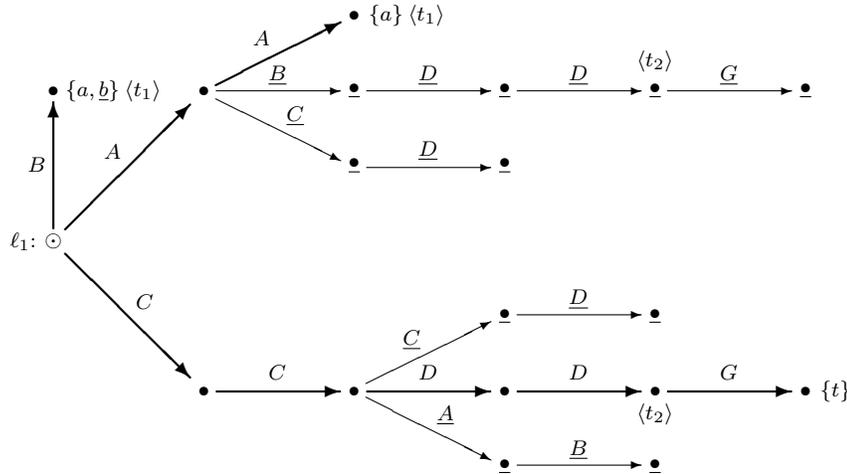


Figure 4.10: The virtual MBFS after clauses ξ_1 , ξ_2 , and ξ_3 have fired.

potential label, b . If the latter label were to become actual as well, we would have a contradiction, and hence an inconsistent set of clauses. Also note that the paths $\ell_1[AA]$ and $\ell_1[B]$ are not coalesced, even though both have the same actual label. We do not enforce the condition that no atomic label be used more than once, as we did in [Heg91] (although we could with only a minor alteration of the algorithm).

At this point, clause ξ_4 may fire, and so is entered into the process queue. Upon its firing, we implement two actions, the first making the path $\ell_1[ABDDG]$ actual and the second coalescing the paths $\ell_1[B]$ and $\ell_1[AA]$. Now upon coalescing the paths $\ell_1[AA]$ and $\ell_1[B]$, the two t_1 tags are matched. This signals that the feature atom $\ell_1[AA] \asymp \ell_1[B]$ is now true, and the false antecedent count of ξ_5 drops to zero. Thus, ξ_5 is enqueued to fire in the process queue. After ξ_5 does fire, the virtual MBFS is as shown in Figure 4.11. Notice that two new edges were added to realize the path $\ell_1[ABDDEF]$. At this point, the antecedents of clause ξ_6 are satisfied, so an entry for firing this clause is entered into the process queue.

Now suppose that the parsing process delivers a new MBFS, which we combine with the existing one. The clauses of this new MBFS are given by the set $\Xi_2 = \{\xi_{11}, \xi_{12}, \xi_{13}\}$, defined as follows.

$$(\xi_{11}) \ell_2[ABC : s].$$

$$(\xi_{12}) \ell_2[AB] \wedge \ell_2[DD] \Rightarrow \ell_2[AB] \asymp \ell_2[CD] \wedge \ell_2[CDE].$$

$$(\xi_{13}) \ell_2[CDE] \Rightarrow \ell_2[CDE : s].$$

The resulting MBFS, now with two distinct bases, is depicted in Figure 4.12. This is the first instance in this example in which we have an MBFS which is not representable as an ordinary feature structure.

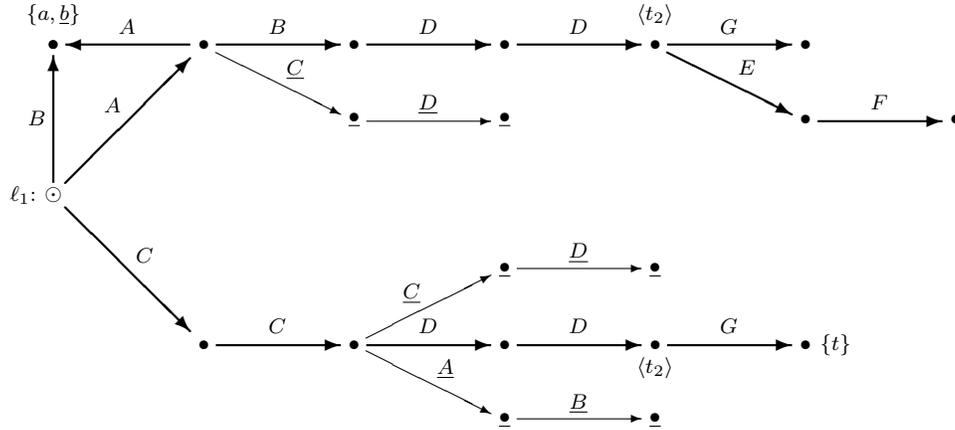


Figure 4.11: The virtual MBFS after clauses ξ_1 through ξ_5 have fired.

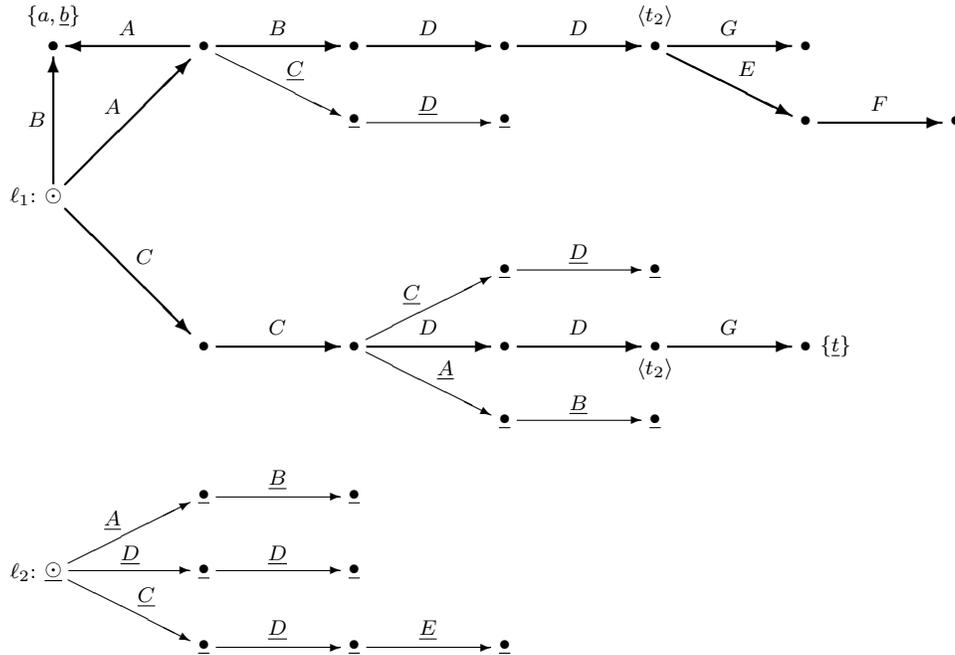


Figure 4.12: The virtual MBFS after the addition of ξ_{11} through ξ_{13} (but before their firing).

At this point, an entry to fire clause ξ_{11} is entered into the process queue, because that clause has a trivial antecedent set. Suppose further that whatever process is controlling the unification issues the clause

$$(\xi_{14}) \quad l_1[CC] \succ l_2[\lambda].$$

- (3) We will examine the possibility of implementing techniques for handling disjunction on top of Horn extended MBFS's, by still requiring all constraints to be clauses (which is not a theoretical limitation at all), and viewing such clauses as rules with disjunctive consequents, as illustrated by the form identified in 2.12.

$$(\sigma_1 \wedge \sigma_2 \wedge \dots \wedge \sigma_{k_1}) \Rightarrow (\rho_1 \vee \rho_2 \vee \dots \vee \rho_{k_2}) \quad (\text{rf})$$

Particularly, it is our conjecture that many of the techniques identified in [Kas87], [ED88], [DE90], and [Str91] may be directly extended to the clausal framework. A great potential advantage of such an approach is that Horn constraints would be handled by our Algorithm 3, while only the disjunctive components would be handled specially. This would guarantee a tractable unification on the widest possible class of structures, with the best heuristics at work on those components which cannot be made uniformly tractable.

- (4) Another formalism which has recently gained favor is that of describing feature structures using terms, rather than finite automata. In this representation, node names become (in one way or another) part of the logic used to describe the structures. The work of Johnson [Joh92] is representative of this approach. We plan to examine how our unification algorithm may be re-expressed within this framework, making use of the fast congruence closure algorithms [NO80] which parallel the ideas of Algorithm 1 of this paper.
- (5) Another mathematical tool which has received relatively little attention within the domain of feature structure unification is that of assigning type to structures. It is our conjecture that using this tool some of the critical operations needed in unification may be “farmed out” to more efficient routines, just as Ait-Kaci and Nasr [AN86] have shown that Prolog may be made more efficient by incorporating type unification. The paper of Carpenter [Car90] provides some beginnings on typed feature structures, upon which we plan to build.

6. Acknowledgments

The author's introduction to unification algorithms on feature structures occurred during his sabbatical visit at the COSMOS computational linguistics group at the University of Oslo from August 1989 through August 1990. This paper is a (regrettably, much delayed) formal presentation of the ideas which saw their origins during that visit. He wishes to express his gratitude to the entire COSMOS group of that time for providing the stimulating environment in which these ideas could be developed, and particularly to Jan Tore Lønning for extending the invitation to include this paper in the technical report series of the University of Oslo computational linguistics group.

Special thanks are due Tore Langholm for carefully reading numerous versions of this report and providing many suggestions for improving the presentation. Although the responsibility for any remaining errors and opaqueness of presentation lies solely with the author, the present quality of presentation could not have been achieved without Tore's detailed and insightful comments.

Finally, thanks are due to Manicka Surendhar who gave the algorithms a very careful reading and pointed out several errors in the pseudocode, and to Liv Janne Nergaard who pointed out errors in the complexity characterizations of Algorithms 1 and 3 and in the running example of Section 4.

References

- [AN86] Ait-Kaci, H. and Nasr, R., "Logic and inheritance," in: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 219–228, 1986.
- [AKM81] Arbib, M. A., Kfoury, A. J., and Moll, R. N., *A Basis for Theoretical Computer Science*, Springer-Verlag, 1981.
- [Bla91] Blackburn, P., "Modal logic and attribute value structures," in: Rosner, M., Rupp, C. J., and Johnson, R., eds., *Constraint Propagation, Linguistic Description, and Computation, Draft Proceedings of the Workshop Held 18-20th September 1991*, pp. 1–19, IDSIA, 1991.
- [Car90] Carpenter, B., "Typed feature structures: Inheritance (in)equality, and extensionality," in: Daelemans, W. and Gazdar, G., eds., *Inheritance in Natural Language Processing: Workshop Proceedings, 1990*, pp. 9–18, Institute for Language Technology and AI, 1990.
- [CL89] Carroll, J. and Long, D., *Theory of Finite Automata with an Introduction to Formal Languages*, Prentice-Hall, 1989.
- [CGT90] Ceri, S., Gottlob, G., and Tanca, L., *Logic Programming and Databases*, Springer-Verlag, 1990.
- [CLR90] Cormen, T. H., Leiserson, C. E., and Rivest, R., *Introduction to Algorithms*, MIT Press, 1990.
- [Dav89] Davis, R. E., *Truth, Deduction, and Computation: Logic and Semantics for Computer Science*, Computer Science Press, 1989.
- [DE90] Dörre, J. and Eisele, A., "Feature logic with disjunctive unification," in: *Proceedings of the COLING 90, Volume 2*, pp. 100–105, 1990.
- [DG84] Dowling, W. F. and Gallier, J. H., "Linear-time algorithms for testing the satisfiability of propositional Horn clauses," *J. Logic Programming*, **3**(1984), pp. 267–284.

- [ED88] Eisele, A. and Dörre, J., “Unification of disjunctive feature descriptions,” in: *Proceedings of the 26th Annual Meeting of the ACL*, pp. 286–294, 1988.
- [FLV89] Fenstad, J. E., Langholm, T., and Vestre, E., “Representations and interpretations,” COSMOS Report No. 09, University of Oslo, Department of Mathematics, 1989, To appear in *Proceedings of the Workshop on Computational Linguistics and Formal Semantics, Lugano, August-September 1988*.
- [Heg91] Hegner, S. J., “Horn extended feature structures: Fast unification with negation and limited disjunction,” in: *Proceedings of the Fifth Conference of the European Chapter of the ACL*, pp. 33–38, 1991.
- [HS73] Herrlich, H. and Strecker, G. E., *Category Theory*, Allyn and Bacon, 1973.
- [Joh91] Johnson, M., “Features and formulas,” *Computational Linguistics*, **17**(1991), pp. 131–151.
- [Joh92] Johnson, M., “Attribute-value logic and natural language processing,” in: Wedekind, J. and Rohrer, C., eds., *Studies in Unification Grammar*, pp. ??–??, MIT Press, to appear, 1992.
- [Kas87] Kasper, R. T., “A unification method for disjunctive feature descriptions,” in: *Proceedings of the 25th Annual Meeting of the ACL*, pp. 235–242, 1987.
- [Kas88] Kasper, R. T., “Conditional descriptions in functional unification grammar,” in: *Proceedings of the 26th Annual Meeting of the ACL*, pp. 233–240, 1988.
- [Lan89] Langholm, T., “How to say no with feature structures,” COSMOS Report No. 13, University of Oslo, Department of Mathematics, 1989.
- [Mak87] Makowsky, J. A., “Why Horn formulas matter in computer science: Initial structures and generic examples,” *J. Comput. System Sci.*, **34**(1987), pp. 266–292.
- [MS91] Moret, B. M. E. and Shapiro, H. D., *Algorithms from P to NP: Volume 1, Design and Efficiency*, Benjamin Cummings, 1991.
- [NO80] Nelson, G. and Oppen, D. C., “Fast decision procedures based upon congruence closure,” *J. Assoc. Comp. Mach.*, **27**(1980), pp. 356–364.
- [Per87] Pereira, F. C. N., “Grammars and logics of partial information,” in: *Proceedings of the International Conference on Logic Programming, Melbourne, Australia*, pp. 990–1013, 1987.

- [Rea91] Reape, M., “An introduction to the semantics of unification-based grammar formalisms,” DYANA Deliverable R3.2.A, University of Edinburgh, 1991.
- [RK86] Rounds, W. C. and Kasper, R., “A complete logical calculus for record structures representing linguistic information,” in: *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pp. 38–43, 1986.
- [SSVV90] Sem, H. F., Sæbø, K. J., Verne, G. B., and Vestre, E. J., “Bound pronouns and absorbed parameters,” COSMOS Report No. 17, University of Oslo, Department of Mathematics, 1990.
- [Shi86] Shieber, S. M., *An Introduction to Unification-Based Approaches to Grammar*, University of Chicago Press, 1986.
- [SS86] Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, 1986.
- [Str91] Strömbäck, L., “Unifying disjunctive feature structures,” Research Report No. LiTH-IDA-R-91-35, Department of Computer and Information Science, Linköping University, 1991.
- [Tar75] Tarjan, R. E., “Efficiency of a good but not linear set union algorithm,” *J. Assoc. Comp. Mach.*, **22**(1975), pp. 215–225.
- [Wed90] Wedekind, J., “A survey of linguistically motivated extensions to unification-based formalisms,” Deliverable R3.1.A, DYANA, 1990.