Unification: A Multidisciplinary Survey

KEVIN KNIGHT

Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213-3890

The unification problem and several variants are presented. Various algorithms and data structures are discussed. Research on unification arising in several areas of computer science is surveyed; these areas include theorem proving, logic programming, and natural language processing. Sections of the paper include examples that highlight particular uses of unification and the special problems encountered. Other topics covered are resolution, higher order logic, the occur check, infinite terms, feature structures, equational theories, inheritance, parallel algorithms, generalization, lattices, and other applications of unification. The paper is intended for readers with a general computer science background—no specific knowledge of any of the above topics is assumed.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Graphs; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on discrete structures*, *Pattern matching*; I.1.3 [**Algebraic Manipulation**]: Languages and Systems; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving; I.2.7 [**Artificial Intelligence**]: Natural Language Processing

General Terms: Algorithms

Additional Key Words and Phrases: Artificial intelligence, computational complexity, equational theories, feature structures, generalization, higher order logic, inheritance, lattices, logic programming, natural language processing, occur check, parallel algorithms, Prolog, resolution, theorem proving, type inference, unification

INTRODUCTION

The unification problem has been studied in a number of fields of computer science, including theorem proving, logic programming, natural language processing, computational complexity, and computability theory. Often, researchers in a particular field have been unaware of work outside their specialty. As a result, the problem has been formulated and studied in a variety of ways, and there is a need for a general presentation. In this paper, I will elucidate the relationships among the various conceptions of unification.

The sections are organized as follows. The three sections The Unification Problem, Unification and Computational Complexity, and Unification: Data Structures and Algorithms introduce unification. Definitions are made, basic research is reviewed, and two unification algorithms, along with the data structures they require, are presented. Next, there are four sections on applications of unification. These applications are theorem proving, logic programming, higher order logic, and natural language processing. The section Unification and Feature Structures is placed before the section on natural language processing and is required for understanding that section. Following are four sections covering various topics of interest. Unification and Equational Theories presents abstract

CONTENTS

work on unification as equation solving. Parallel Algorithms for Unification reviews recent literature on unification and parallelism. Unification, Generalization, and Lattices discusses unification and its dual operation, generalization, in another abstract setting. Finally, in Other Applications of Unification, research that connects unification to research in abstract data types, programming languages, and machine learning is briefly surveyed. The conclusion gives an abstract of some general properties of unification and projects research trends into the future.

Dividing this paper into sections was a difficult task. Some research involving unification straddles two fields. Take, for example, higher order logic and theorem proving—should this work be classified as Unification and Higher Order Logic or Unification and Theorem Proving? Or consider ideas from logic programming put to work in natural language processing—is it Unification and Logic Programming or Unification and Natural Language Processing? Likewise, some researchers work in multiple fields, making their work hard to classify. Finally, it is sometimes difficult to define the boundary between two disciplines; for example, logic programming grew out of classical theorem proving, and the origin of a particular contribution to unification is not always clear. I have provided a balanced classification, giving cross references when useful.

Finally, the literature on unification is enormous. I present all of the major results, if not the methods used to reach them.

1. THE UNIFICATION PROBLEM

Abstractly, the *unification problem* is the following: Given two descriptions x and y, can we find an object z that fits both descriptions?

The unification problem is most often stated in the following context: Given two terms of logic built up from function symbols, variables, and constants, is there a substitution of terms for variables that will make the two terms identical? As an example consider the two terms $f(x, y)$ and $f(g(y, a), h(a))$. They are said to be *unifiable*, since replacing x by $g(h(a), a)$ and y by $h(a)$ will make both terms look like $f(g(h(a), a), h(a))$. The nature of such unifying substitutions, as well as the means of computing them, makes up the study of unification.

In order to ensure the consistency, conciseness, and independence of the various sections of this paper, I introduce a few formal definitions:

Definition 1.1

A *variable symbol* is one of $\{x, y, z, \ldots\}$. A *constant symbol* is one of $\{a, b, c, \ldots\}$. A *function symbol* is one of $\{f, g, h, \ldots\}$.

Definition 1.2

A *term* is either a variable symbol, a constant symbol, or a function symbol followed by a series of *terms*, separated by commas, enclosed in parentheses. Sample terms are a, x, $f(x, y)$, $f(g(x, y), h(z))$. Terms are denoted by the symbols $\{s, t, u, \ldots\}$.[1]

Definition 1.3

A *substitution* is first a function from variables into terms. Substitutions are denoted by the symbols $\{\alpha, \tau, \theta, \ldots\}$. A substitution σ in which $\sigma(x)$ is $g(h(a), a)$ and $\sigma(y)$ is $h(a)$ can be written as a set of bindings enclosed in curly braces; that is, $\{x \leftarrow g(h(a), a), y \leftarrow h(a)\}$. This set of bindings is usually finite. A substitution is *also* a function from terms to terms via its *application*. The application of substitution σ to term t is also written $\sigma(t)$ and denotes the term in which each variable x_i in t is replaced by $\sigma(x_i)$.

Substitutions can be composed: $\sigma\theta(t)$ denotes the term t after the application of the substitutions of θ followed by the application of the substitutions of σ. The substitution $\sigma\theta$ is a new substitution built from old substitutions σ and θ by (1) modifying θ by applying σ to its terms, then (2) adding variable-term pairs in σ not found in θ. Composition of substitutions is associative, that is, $(\sigma\theta)\tau(s) = \sigma(\theta\tau)(s)$ but is not in general commutative, that is, $\sigma\theta(s) \neq \theta\sigma(s)$.

Definition 1.4

Two terms s and t are *unifiable* if there exists a substitution σ such that $\sigma(s) = \sigma(t)$. In such a case, σ is called a *unifier* of s and t, and $\sigma(s)$ is called a *unification* of s and t.

Definition 1.5

A unifier σ of terms s and t is called a *most general unifier* (MGU) of s and t if for any other unifier θ, there is a substitution τ such that $\tau\sigma = \theta$. Consider, for example, the two terms $f(x)$ and $f(g(y))$. The MGU is $\{x \leftarrow g(y)\}$, but there are many non-MGU unifiers such as $\{x \leftarrow g(a), y \leftarrow a\}$. Intuitively, the MGU of two terms is the simplest of all their unifiers.

Definition 1.6

Two terms s and t are said to *match* if there is a substitution σ such that $\sigma(s) = t$. Matching is an important variant of unification.

Definition 1.7

Two terms s and t are *infinitely unifiable* if there is a substitution, possibly containing infinitely long terms, that is a unifier of s and t. For example, x and $f(x)$ are not unifiable, but they are infinitely unifiable under the substitution $\sigma = \{x \leftarrow f(f(f(\ldots)))\}$, since $\sigma(x)$ and $\sigma(f(x))$ are both equal to $f(f(f(\ldots)))$. An infinitely long term is essentially one whose string representation (to be discussed in the next section) requires an infinite number of symbols—for a formal discussion, see Courcelle [1983].

These general definitions will be used often throughout the paper. Concepts specific to particular areas of study will be defined in their respective sections.

2. UNIFICATION AND COMPUTATIONAL COMPLEXITY

In his 1930 thesis, Herbrand [1971] presented a nondeterministic algorithm to compute a unifier of two terms. This work was motivated by Herbrand's interest in equation solving. The modern utility and notation of unification, however, originated with Robinson. In his pioneering paper Robinson [1965] introduced a method of

[1] I will often use the word "term" to describe the mathematical object just defined. Do not confuse these terms with terms of predicate logic, which include things like $\forall(x)\forall(y):(f(x) \rightarrow f(y)) \leftrightarrow (\neg f(y) \rightarrow \neg f(x))$.

theorem proving based on resolution, a powerful inference rule. Central to the method was unification of first-order terms. Robinson proved that two first-order terms, if unifiable, have a *unique* most general unifier. He gave an algorithm for computing the MGU and proved it correct.

Guard [1964] independently studied the unification problem under the name of matching. Five years later, Reynolds [1970] discussed first-order terms using lattice theory and showed that there is also a unique "most specific generalization" of any two terms. See Section 11 for more details.

Robinson's original algorithm was inefficient, requiring exponential time and space. A great deal of effort has gone into improving the efficiency of unification. The remainder of this section will review that effort. The next section will discuss basic algorithmic and representational issues in detail.

Robinson himself began the research on more efficient unification. He wrote a note [Robinson 1971] about unification in which he argued that a more concise representation for terms was needed. His formulation greatly improved the space efficiency of unification. Boyer and Moore [1972] gave a unification algorithm that shares structure; it was also space efficient but was still exponential in time complexity.

In 1975, Venturini-Zilli [1975] introduced a marking scheme that reduced the complexity of Robinson's algorithm to quadratic time.

Huet's [1976] work on higher order unification (see also Section 6) led to an improved time bound. His algorithm is based on maintaining equivalence classes of subterms and runs in $O(n\alpha(n))$ time, where $\alpha(n)$ is an extremely slow-growing function. We call this an *almost linear* algorithm. Robinson also discovered this algorithm, but it was not published in accessible form until Vitter and Simons' [1984, 1986] study of parallel unification. The algorithm is a variant of the algorithm used to test the equivalence of two finite automata. Aït-Kaci [1984] came up with a similar, more general algorithm in his dissertation. Baxter [1973] also discovered an almost-linear algorithm. He presented the algorithm in his thesis [Baxter 1976], in which he also discussed restricted and higher order versions of unification.

In 1976, Paterson and Wegman [1976, 1978] gave a *truly linear* algorithm for unification. Their method depends on a careful propagation of the equivalence-class relation of Huet's algorithm. The papers of Paterson and Wegman are rather brief; de Champeaux [1986] helps to clarify the issues involved.

Martelli and Montanari [1976] independently discovered another linear algorithm for unification. They further improved efficiency [Martelli and Montanari 1977] by updating Boyer and Moore's structure-sharing approach. In 1982, they gave a thorough description of an efficient unification algorithm [Martelli and Montanari 1982]. This last algorithm is no longer truly linear but runs in time $O(n + m \log m)$, where m is the number of *distinct* variables in the terms. The paper includes a practical comparison of this algorithm with those of Huet and Paterson and Wegman. Martelli and Montanari also cite a study by Trum and Winterstein [1978], who implemented several unification algorithms in Pascal in order to compare actual running times.

Kapur et al. [1982] reported a new linear algorithm for unification. They also related the unification problem to the connected components problem (graph theory) and the online/offline equivalence problems (automata theory). Unfortunately, their proof contained an error, and the running time is actually nonlinear (P. Narendran, personal communication, 1988).

Corbin and Bidoit [1983] "rehabilitated" Robinson's original unification algorithm by using new data structures. They reduced the exponential time complexity of the algorithm to $O(n^2)$ and claimed that the algorithm is simpler than Martelli and Montanari's and superior in practice.

The problem of unifying with infinite terms has received some attention. (Recall the definition of *infinitely unifiable*.) In his thesis, Huet [1976] showed that in the case of infinite unification, there still exists a single MGU—he gave an almost-linear algorithm for computing it. Colmerauer [1982b] gave two unification algorithms for

```
function UNIFY(t1, t2) ⇒ (unifiable: Boolean, σ: substitution)
begin
  if t1 or t2 is a variable then
    begin
      let x be the variable, and let t be the other term
      if x = t, then (unifiable, σ) ← (true, ∅)
      else if occur(x, t) then unifiable ← false
      else(unifiable, σ) ← (true, {x ← t})
    end
  else
    begin
      assume t1 = f(x1, . . . , xn) and t2 = g(y1, . . . , ym)
      if f ≠ g or m ≠ n then unifiable ← false
      else
        begin
          k ← 0
          unifiable ← true
          σ ← nil
          while k < m and unifiable do
            begin
              k ← k + 1
              (unifiable, τ) ← UNIFY(σ(xk), σ(yk))
              if unifiable then σ ← compose(τ, σ)
            end
        end
    end
  return (unifiable, σ)
end
```

Figure 1. A version of Robinson's original unification algorithm.

infinite terms, one "theoretical" and one "practical": The practical version is faster but may not terminate. Mukai [1983] gave a new variation of Colmerauer's practical algorithm and supplied a termination proof. Jaffar [1984] presented another algorithm, designed along the lines of Martelli and Montanari. Jaffar claimed that his algorithm is simple and practical, even for finite terms; he discussed the relative merits of several unification algorithms, including Colmerauer [1982b], Corbin and Bidoit [1983], Martelli and Montanari [1977], Mukai [1983], and Paterson and Wegman [1976]. None of these algorithms is truly linear: The theoretical question of detecting *infinite unifiability* in linear time was left open by Paterson and Wegman and has yet to be solved.

3. UNIFICATION: DATA STRUCTURES AND ALGORITHMS

As mentioned in the previous section, Robinson [1965] published the first modern unification algorithm. A version of that algorithm appears in Figure 1. (This version is borrowed in large part from Corbin and Bidoit [1983].) The function UNIFY takes two terms ($t1$ and $t2$) as arguments and returns two items: (1) the Boolean-valued *unifiable*, which is *true* if and only if the two terms unify and (2) σ, the unifying substitution. The algorithm proceeds from left to right, making substitutions when necessary. The routines "occur" and "compose" need some explanation. The "occur" function reports true if the first argument (a variable) occurs anywhere in the second argument (a term). This function call is known as the *occur check* and is discussed in more detail in Section 5. The "compose" operator composes two substitutions according to the definition given in the Introduction of this paper.

Robinson's original algorithm was exponential in time and space complexity. The major problem is one of representation. Efficient algorithms often turn on special data structures; such is the case with unification. This paper will now discuss the various data structures that have been proposed.

First-order terms have one obvious representation, namely, the sequence of symbols we have been using to write them

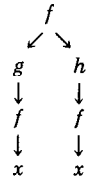

Figure 2. Tree representation of the term $f(g(f(x)), h(f(x)))$.

down. (Recall the inductive definition given at the beginning of this paper.) In other words, a term can be represented as a linear array whose elements are taken from function symbols, variables, constants, commas, and parentheses. We call this the *string representation* of a term. (As an aside, it is a linguistic fact that commas and parentheses may be eliminated from a term without introducing any ambiguity.) Robinson's [1965] unification algorithm (Figure 1) uses this representation.

The string representation is equivalent to the *tree representation*, in which a function symbol stands for the root of a subtree whose children are represented by that function's arguments. Variables and constants end up as the leaves of such a tree. Figure 2 shows the tree representation of the term $f(g(f(x)), h(f(x)))$.

The string and tree representations are acceptable methods for writing down terms, and they find applications in areas in which the terms are not very complicated, for example, Tomita and Knight [1988]. They have drawbacks, however, when used in general unification algorithms. The problem concerns structure sharing.

Consider the terms $f(g(f(x)), h(f(x)))$ and $f(g(f(a)), h(f(a)))$. Any unification algorithm will ensure that the function symbols match and ensure that corresponding arguments to the functions are unifiable. In our case, after processing the subterms $f(x)$ and $f(a)$, the substitution $\{x \leftarrow a\}$ will be made. There is, however, no need to process the second occurrences of $f(x)$ and $f(a)$, since we will just be doing the same work over again. What is needed is a more concise representation for the terms: a *graph representation*. Figure 3 shows graphs for this pair of terms.

The subterm $f(x)$ is shared; the work to unify it with another subterm need be done only once. If $f(x)$ were a much larger structure, the duplication of effort would be more serious of course. In fact, if subterms are not shared, it may be necessary to generate exponentially large structures during unification (see Corbin and Bidoit [1983], de Champeaux [1986], and Huet [1976]).

The algorithms of Huet [1976], Baxter [1973, 1976], and Jaffar [1984] all use the graph representation. Paterson and Wegman's linear algorithm [1976, 1978] uses a graph representation modified to include *parent pointers* so that leaves contain information about their parents, grandparents, and so on. The Paterson–Wegman algorithm is still linear for terms represented as strings, since terms can be converted from string representation to graph representation (and vice versa) in linear time [de Champeaux 1986]. Corbin and Bidoit [1983] also use a graph representation that includes parent pointers. The algorithms of Martelli and Montanari [1986] and of Kapur et al. [1982] both deal directly with sets of equations produced by a unification problem. Their representations also share structure.

What I have called the *graph representation* is known in combinatorics as a *directed acyclic graph*, or *dag*, in which all vertices of the graph are labeled. If we allow our graphs to contain cycles, we can model infinitely long terms, and unification may product substitutions involving such terms. This matter is discussed in connection with the *occur check* in Section 5.

Before closing this section, I will present a unification algorithm more efficient than the one in Figure 1. I have chosen a variant of Huet's algorithm, since it is rather simple

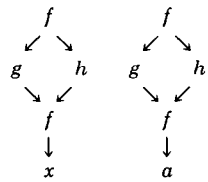

Figure 3. Term graphs for $f(g(f(x)), h(f(x)))$ and $f(g(f(a)), h(f(a)))$.

```
function UNIFY(t1, t2) ⇒ (unifiable: Boolean, σ: substitution)
begin
  pairs-to-unify ← {(t1, t2)}
  for each node z in t1 and t2,
    z.class ← z
  while pairs-to-unify ≠ ∅ do
    begin
      (x, y) ← pop (pairs-to-unify)
      u ← FIND(x)
      v ← FIND(y)
      if u ≠ v then
        if u and v are not variables, and u.symbol ≠ v.symbol or
          numberof(u.subnodes) ≠ numberof(v.subnodes) then
            return (false, nil)
        else
          begin
            w ← UNION(u, v)
            if w = v and u is a variable then
              u.class ← v
            if neither v nor u is a variable then
              begin
                let (u1, . . . , un) be u.subnodes
                let (v1, . . . , vn) be v.subnodes
                for i ← 1 to n do
                  push ((ui, vi), pairs-to-unify)
              end
          end
    end
  Form a new graph composed of the root nodes of the equivalence classes.
  This graph is the result of the unification.
  If the graph has a cycle, return (false, nil), but the terms are infinitely unifiable.
  If the graph is acyclic, return (true, σ), where σ is a substitution in which any variable x is mapped on to the
      root of its equivalence class, that is, FIND(x).
end
```

Figure 4. A version of Huet's unification algorithm.

and uses the graph representation of terms. The algorithm is shown in Figure 4.

Terms are represented as graphs whose nodes have the following structure:

```
type = node
  symbol: a function, variable, or constant
    symbol
  subnodes: a list of nodes that are children of
    this node
  class: a node that represents this node's
    equivalence class
end
```

Huet's algorithm maintains a set of equivalence classes of nodes using the FIND and UNION operations for merging disjoint sets (for an explanation of these operations, see Aho et al. [1974]). Each node starts off in its own class, but as the unification proceeds, classes are merged together. If nodes with different function symbols are merged, failure occurs. At the end, the equivalence classes over the nodes in the two-term graphs form a new graph, namely, the result of the unification. If this

graph contains a cycle, the terms are *infinitely unifiable*; if the graph is acyclic, the terms are *unifiable*. Omitting the acyclicity test corresponds to omitting the occur check discussed in Section 5.

The running time of Huet's algorithm is $O(n\alpha(n))$, where n is the number of nodes and $\alpha(n)$ is an extremely slow-growing function. $\alpha(n)$ never exceeds 5 in practice, so the algorithm is said to be *almost linear*. Note also that it is nonrecursive, which may increase efficiency. For more details, see Huet [1976] and Vitter and Simons [1984, 1986]. A version of this algorithm, which includes inheritance-based information (see Section 7), can be found in Aït-Kaci [1984] and Aït-Kaci and Nasr [1986].

4. UNIFICATION AND THEOREM PROVING

Robinson's [1965] work on resolution theorem proving served to introduce the unification problem into computer science. Automatic theorem proving (a.k.a. computational logic or automated deduction) became an area of concentrated research, and unification suddenly became a topic of interest. In this section, I will only discuss first-order theorem proving, although there are theorem-proving systems for higher order logic. Higher order *unification* will be discussed in a later section.

4.1 The Resolution Rule

I will briefly demonstrate, using an example, how unification is used in resolution theorem proving. In simplest terms, resolution is a rule of inference that allows one to conclude from "A or B" and "not-A or C" that "B or C." In real theorem proving, resolution is more complex. For example, from the two facts

(1) *Advisors get angry when students don't take their advice.*

(2) *If someone is angry, then he doesn't take advice.*

We want to infer the third fact:

(3) *If a student is angry, then so is his advisor.*

The first job is to put these three statements into logical form:

(1a) $\forall z : student(z) \rightarrow [\neg takesadvice(z, advisor(z)) \rightarrow angry(advisor(z))]$

(2a) $\forall x, y : angry(x) \rightarrow \neg takesadvice(x, y)$

(3a) $\forall w : student(w) \rightarrow [angry(w) \rightarrow angry(advisor(w))]$

Next, we drop the universal quantifiers and remove the implication symbols ($x \rightarrow y$ is equivalent to $\neg x \vee y$):

(1b) $\neg student(z) \vee takesadvice(z, advisor(z)) \vee angry(advisor(z))$

(2b) $\neg angry(x) \vee \neg takesadvice(x, y)$

(3b) $\neg student(w) \vee \neg angry(w) \vee angry(advisor(w))$

(1b), (2b), and (3b) are said to be in *clausal* form. Resolution is a rule of inference that will allow us to conclude the last clause from the first two. Here it is in its simplest form:

The Resolution Rule

If clause A contains some term s and clause B contains the *negation* of some term t and if s and t are *unifiable* by a substitution σ, then a *resolvent* of A and B is generated by (1) combining the clauses from A and B, (2) removing terms s and t, and (3) applying the substitution σ to the remaining terms. *If clauses A and B have a resolvent C, then C may be inferred from A and B.*

In our example, let s be $takesadvice(z, advisor(z))$ and let t be $takesadvice(x, y)$. (1b) contains s, and (2b) contains the negation of t. Terms s and t are unifiable under the substitution $\{x \leftarrow z, y \leftarrow advisor(z)\}$. Removing s from (1b) and the negation of t from (2b) and applying the substitution to the rest of the terms in (1b) and (2b), we get the resolvent:

(4) $\neg student(z) \vee angry(advisor(z)) \vee \neg angry(z)$

Expression (4) is the same as (3b), subject to disjunct reordering and renaming of the unbound variable, and therefore resolution has made the inference we intended.

Resolution is a powerful inference rule—so powerful that it is the *only* rule needed for a sound and complete system of logic. The simple example here is intended only to illustrate the use of unification in the resolution method; for more details about resolution, see Robinson's paper [1965] and Chapter 4 of Nilsson's book [1980].

4.2 Research

Much of the theorem-proving community has recently moved into the field of logic programming, while a few others have moved into higher order theorem proving. I will present the work of these two groups in different sections: Unification and Higher Order Logic and Unification and Logic Programming.

5. UNIFICATION AND LOGIC PROGRAMMING

The idea of "programming in logic" came directly out of Robinson's work: The original (and still most popular) logic programming language, Prolog, was at first a tightly constrained resolution theorem prover. Colmerauer and Roussel turned it into a useful language [Battani and Meloni 1973; Colmerauer et al. 1973; Roussel 1975], and van Emden and Kowalski [1976] provided an elegant theoretical model based on Horn clauses. Warren et al. [1977] presented an accessible early description of Prolog.

Through its use of resolution, Prolog inherited unification as a central operation. A great deal of research in logic programming focuses on efficient implementation, and unification has therefore received some attention. I will first give an example of how unification is used in Prolog; then I will review the literature on unification and Prolog, finishing with a discussion of the logic programming languages Concurrent Prolog, LOGIN, and CIL.

5.1 Example of Unification in Prolog

I turn now to unification as it is used in Prolog. Consider a set of four Prolog assertions (stored in a "database") followed by a query. (One example is taken from Clocksin and Mellish [1981].)

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).

?- likes(mary, X), likes(john, X).
```

The query at the end asks, Does Mary like something that John likes? Prolog takes the first term of the query *likes*(*mary*, *X*) and tries to *unify* it with some assertion in the database. It succeeds in unifying the terms *likes*(*mary*, *X*) and *likes*(*mary*, *food*) by generating the substitution $\{X \leftarrow food\}$. Prolog applies this substitution to *every* term in the query. Prolog then moves on to the second term, which is now *likes*(*john*, *food*). This term fails to unify with any other term in the database.

Upon failure, Prolog backtracks. That is, it "undoes" a previous unification, in this case, the unification of *likes*(*mary*, *X*) with *likes*(*mary*, *food*). It attempts to unify the first query term with *another* term in the database: the only other choice is *likes*(*mary*, *wine*). The terms unify under the substitution $\{X \leftarrow wine\}$, which is also applied to the second query term *likes*(*john*, *X*) to give *likes*(*john*, *wine*). This term can now unify with a term in the database, namely, the third assertion, *likes*(*john*, *wine*). Done with all query terms, Prolog outputs the substitutions it has found; in this case, "$X = wine$."

I used such a simple example only to be clear. The example shows how Prolog makes extensive use of unification as a pattern-matching facility to retrieve relevant facts from a database. Unification itself becomes nontrivial when the terms are more complicated.

5.2 Research

Colmerauer's original implementation of unification differed from Robinson's in one important respect: Colmerauer deliberately left out the "occur check," allowing Prolog to attempt unification of a variable with a term already containing that variable.

```
foo(X, X).
?- foo(f(Y), Y).
Y = f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f( . . .
```

Figure 5. Infinite unification in Prolog.

```
Version 1.
  less(X, s(X)).  /* any number is less than its successor */
  ?- less(s(Y), Y).  /* is there a Y whose successor is less than Y? */
  Y = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(
  s(s(s(s(s ...
Version 2.
  less(X, s(X)).
  foo :- less(s(Y), Y).  /* we don't care about the value of Y */
  ?- foo.  /* is there a number whose successor is less than it? */
  yes.  /* there is?! */
```

Figure 6. Prolog draws the wrong conclusion because it lacks an occur check.

For example, Prolog will unify the terms $x$and $f(x)$, using the substitution $\{x \leftarrow f(f(f(\ldots)))\}$; Robinson's occur check, on the other hand, would instantly report nonunifiability.

Discarding the occur check corresponds to moving from *unification* to *infinite unification.* Figure 5 shows Prolog's infinite unification in action. In unifying $f(Y)$ and Y, Prolog comes up with an infinite substitution.

Omitting the occur check results in efficiency gains that outweigh the risk incurred. For example, the concatentation of two lists, a linear-time operation without the occur check, becomes an $O(n^2)$ time operation with the occur check [Colmerauer 1982b]. For this reason, most current Prolog interpreters omit the occur check.

But since its unification is really infinite unification, in this respect, Prolog can be said to be "unsound." Figure 6 (adapted from Plaisted [1984]) shows a case in which Prolog draws an unexpected conclusion, one that would be blocked by an occur check.

Colmerauer [1982b, 1983] studied infinite terms in an effort to provide a theoretical model for real Prolog implementations that lack the occur check. He examined the termination problem and produced a unification algorithm for infinite terms. Plaisted [1984] presented a preprocessor for Prolog programs that detects potential problems due to Prolog's lack of an occur check. See the end of Section 2 for a discussion of algorithms for infinite unification.

As noted above, Prolog is a search engine, and when a particular search path fails, backtracking is necessary. As a result, some of the unifications performed along the way must be "undone." Studies of efficient backtracking can be found in Bruynooghe and Pereira [1984], Chen et al. [1986], Cox [1984], Matwin and Pietrzykowski [1982], and Pietrzykowski and Matwin [1982]. Mannila and Ukkonen [1986] viewed Prolog execution as a sequence of unification and deunifications, relating it to the UNION/FIND (disjoint set union) problem. They showed how standard (fast) unification algorithms can have poor asymptotic performance when we consider deunifications; algorithms with better performance are given.

Concurrent Prolog [Shapiro 1983] extends Prolog to concurrent programming

and parallel execution. In this model, unification may have one of three possible outcomes: *succeed*, *fail*, or *suspend*. Unification suspends when a special "read-only variable" is encountered—execution of the unification call is restarted when that variable takes on some value. It is not possible to rely on backtracking in the Concurrent Prolog model because all computations run independently and in parallel. Thus, unifications cannot simply be "undone" as in Prolog, and local copies of structures must be maintained at each unification. Strategies for minimizing these copies are discussed in Levy [1983].

Aït-Kaci and Nasr [1986] present a new logic programming language called LOGIN, which integrates inheritance-based reasoning directly into the unification process. The language is based on ideas from Aït-Kaci's dissertation [1984], described in more detail in Section 7.

Mukai [1985a, 1985b] and Mukai and Yasukawa [1985] introduce a variant of Prolog called CIL aimed at natural language applications, particularly those involving situation semantics [Barwise and Perry 1983]. CIL is based on an extended unification that uses conditional statements, roles, and types and contains many of the ideas also present in LOGIN. Hasida [1986] presents another variant of unification that handles conditions on patterns to be unified.

6. UNIFICATION AND HIGHER ORDER LOGIC

First-order logic is sometimes too limited or too unwieldy for a given problem. Second-order logic, which allows variables to range over functions (and predicates) as well as constants, can be more useful. Consider the statement "Cats have the same annoying properties as dogs." We can express this in second-order logic as

$$\forall(x)\forall(y)\forall(f):$$
$$[cat(x) \wedge dog(y) \wedge annoying(f)]$$
$$\rightarrow [f(x) \leftrightarrow f(y)]$$

Note that the variable f ranges over *predicates*, not constant values. In a more mathematical vein, here is a statement of the induction property of natural numbers:

$$\forall(f):[(f(0) \wedge \forall(x)[f(x) \rightarrow f(x+1)])$$
$$\rightarrow \forall(x)f(x)]$$

That is, for any predicate f, if f holds for 0 and if $f(x)$ implies $f(x+1)$, then f holds for all natural numbers.

For a theorem prover to deal with such statements, it is natural to work with axioms and inference rules of a higher order logic. In such a case, unification of higher order terms has great utility.

Unification in higher order logic requires some special notation for writing down higher order terms. The typed λ-calculus [Church 1940; Henkin 1950] is one commonly used method. I will not introduce any formal definitions, but here is an example: "$\lambda(u, v)(u)$" stands for a function of two arguments, u and v, whose value is always equal to the first argument, u. Unification of higher order terms involves λ-conversion in conjunction with application of substitutions.

Finally, we must distinguish between *function constants* (denoted $A, B, C, \ldots$) and *function variables* (denoted $f, g, h, \ldots$), which range over those constants. Now we can look at unification in the higher order realm.

6.1 Example of Second-Order Unification

Consider the two terms $f(x, b)$ and $A(y)$. Looking for a unifying substitution for variables f, x, and y, we find

$$f \leftarrow \lambda(u, v)(u)$$
$$x \leftarrow A(y)$$
$$y \leftarrow y$$

This substitution produces the unification $A(y)$. But if we keep looking, we find another unifier:

$$f \leftarrow \lambda(u, v)A(g(u, v))$$
$$x \leftarrow x$$
$$y \leftarrow g(x, b)$$

In this case, the unification is $A(g(x, b))$. Notice that neither of the two unifiers is

more general than the other! A pair of higher order terms, then, may have more than one "most" general unifier. Clearly, the situation is more complex than that of first-order unification.

6.2 Research

Gould [1966a, 1966b] was the first to investigate higher order unification in his dissertation.[2] He showed that some pairs of higher order terms possess many general unifiers, as opposed to the first-order case, where terms always (if unifiable) have a *unique* MGU. Gould gave an algorithm to compute general unifiers, but his algorithm did not return a complete set of such unifiers. He conjectured that the unifiability problem for higher order logic was solvable.

Robinson [1968], urged that research in automatic theorem proving be moved into the higher order realm. Robinson was nonplussed by Gödel's incompleteness results and maintained that Henkin's [1950] interpretation of logical validity would provide the completeness needed for a mechanization of higher order logic. Robinson [1969, 1970] followed up these ideas. His theorem provers, however, were not completely mechanized; they worked in interactive mode, requiring a human to guide the search for a proof.

Andrews [1971] took Robinson's original resolution idea and applied it rigorously to higher order logic. But, like Robinson, he only achieved partial mechanization: Substitution for predicate variables could not be done automatically. Andrews describes an interactive higher order theorem prover based on generalized resolution in Andrews and Cohen [1977] and Andrews et al. [1984].

Darlington [1968, 1971] chose a different route. He used a subset of second-order logic that allowed him only a little more expressive power than first-order logic. He was able to mechanize this logic *completely* using a new unification algorithm (first called *f-matching*). To repeat, he was not using full second-order logic; his algorithm could only unify a predicate variable with a predicate constant of greater arity.

[2] The above example is taken from that dissertation.

Huet [1972] in his dissertation presented a new refutational system for higher order logic based on a simple unification algorithm for higher order terms. A year later, part of that dissertation was published as Huet [1973b]. It showed that the unification problem for third-order logic was *undecidable*; that is, there exists no effective procedure to determine whether two third- (or higher) order terms are unifiable. Lucchesi [1972] independently made this same discovery.

Baxter [1978] extended Huet's and Lucchesi's undecidability results to third-order *dyadic* unification. Three years later, Goldfarb [1981] demonstrated the undecidability of the unification problem for *second-order* terms by a reduction of Hilbert's Tenth Problem to it.

Huet's refutational system is also described in Huet [1973a], and his higher order unification algorithm can be found in Huet [1975]. Since ω-order unification is in general undecidable, Huet's algorithm may not halt if the terms are not unifiable. The algorithm first searches for the *existence* of a unifier, and if one is shown to exist, a very general *preunifier* is returned.

Huet's system was based on the typed λ-calculus introduced by Church [1940]. The λ-calculus was the basis for Henkin's [1950] work on higher order completeness mentioned above in connection with Robinson [1968]. Various types of mathematical theorems can be expressed naturally and compactly in the typed λ-calculus, and for this reason almost all work on higher order unification uses this formalism.

Pietrzykowski and Jensen also studied higher order theorem proving. Pietrzykowski [1973] gave a complete mechanization of second-order logic, which he and Jensen extended to ω-order logic in Jensen and Pietrzykowski [1976] and Pietrzykowski and Jensen [1972, 1973]. Like Huet, they based their mechanization on higher order unification. Unlike Huet, however, they presented an algorithm that computes a complete set of general unifiers.[3] The algorithm is more complicated than Huet's,

[3] Of course, the algorithm still may not terminate if the terms are nonunifiable.

using five basic types of substitutions as opposed to Huet's two, and is generally redundant.

Winterstein [1976] reported that even for the simple case of *monadic*(one argument per function) second-order terms unification could require exponential time under the algorithms proposed by Huet, Pietrzykowski, and Jensen. Winterstein and Huet independently arrived at better algorithms for monadic second-order unification.

Darlington [1977] attempted to improve the efficiency of higher order unification by sacrificing completeness. If two terms unify, Darlington's algorithm may not report this fact, but he claims that in the usual case unification runs more swiftly.

As mentioned above, Goldfarb [1981] showed that higher order unification is in general undecidable. There are, however, variants that *are* decidable. Farmer [1987] shows that second-order monadic unification (discussed two paragraphs ago) is actually decidable. Siekmann [1984] suggests that higher order unification *under some theory T* might be solvable (see Section 9), but there has been no proof to this effect.

7. UNIFICATION AND FEATURE STRUCTURES

First-order terms turn out to be too limited for many applications. In this section, I will present structures of a somewhat more general nature. The popularity of these structures has resulted in a line of research that has split away from mainstream work in theorem proving and logic programming. An understanding of these structures is necessary for an understanding of much research on unification.

Kay [1979] introduced the idea of using unification for manipulating the syntactic structures of natural languages (e.g., English and Japanese). He formalized the linguistic notion of "features" in what are now known as "feature structures."

Feature structures resemble first-order terms but have several restrictions lifted:

- Substructures are labeled symbolically, not inferred by argument position.
- Fixed arity is not required.

$$\begin{bmatrix} \textit{type}: & \textit{person} & \\ \textit{name}: & \textit{john} & \\ \textit{age}: & 23 & \\ \textit{height}: & 70 & \\ \textit{spouse}: & \begin{bmatrix} \textit{name}: & \textit{mary} \\ \textit{age}: & 25 \end{bmatrix} & \end{bmatrix}$$

Figure 7. A feature structure.

- The distinction between function and argument is removed.
- Variables and coreference are treated separately.

I will now discuss each of these differences.

Substructure labeling. Features are labeled *symbolically*, not inferred by argument position within a term. For example, we might want the term "*person*(*john*, 23, 70, *spouse*(*mary*, 25))" to mean "There is a person named John, who is 23 years old, 70 inches tall, and who has a 25-year-old spouse named Mary." If so, we must remember the convention that the *third* argument of such a term represents a person's *height*. An equivalent feature structure, using explicit labels for substructure, is shown in Figure 7. (I have used a standard, but not unique, representation for writing down feature structures.)

Fixed arity. Feature structures do not have the *fixed arity* of terms. Traditionally, feature structures have been used to represent *partial information*—unification combines partial structures into larger structures, assuming no conflict occurs. In other words, unification can result in new structures that are *wider* as well as deeper than the old structures. Figure 8 shows an example (the square cup stands for unification).

Notice how all of the information contained in the first two structures ends up in the third structure. If the *type* features of the two structures had held different values, however, unification would have failed.

The demoted functor. In a term, the *functor* (the function symbol that begins the term) has a special place. In feature structures, all information has equal status. Sometimes one feature is "primary" for a

$$\begin{bmatrix} type: & person \\ name: & john \end{bmatrix} \sqcup \begin{bmatrix} type: & person \\ age: & 23 \end{bmatrix} \Rightarrow \begin{bmatrix} type: & person \\ name: & john \\ age: & 23 \end{bmatrix}$$

Figure 8. Unification of two feature structures.

$$\begin{bmatrix} type: & person & \\ name: & john & \\ age: & [\,] & \\ height: & [\,] & \\ spouse: & \boxed{1} & \begin{bmatrix} name: & [\,] \\ age: & [\,] \end{bmatrix} \\ bestfriend: & \boxed{1} & \end{bmatrix}$$

Figure 9. A feature structure with variables and internal coreference constraints.

particular use, but this is not built into the syntax.

Variables and coreference. Variables in a term serve two purposes: (1) They are place holders for future instantiation, and (2) they enforce equality constraints among different parts of the term. For example, the term *person*(*john*, *Y*, *Z*, *spouse*(*W*, *Y*)) expresses the idea "John can marry anyone exactly as old as he is" (because the variable *Y* appears in two different places). Note, however, that equality constraints are restricted to the *leaves* of a term.[4] Feature structures allow within-term coreferences to be expressed by separating the two roles of variables.

If we wished, for example, to express the constraint that John must marry his best friend, we might start with a term such as

person(*john*, *Y*, *Z*, *spouse*(*W*, *X*),
bestfriend(*W*, *X*))

This is awkward and unclear, however; it is tedious to introduce new variables for each possible leaf, and moreover, the term seems to imply that John can marry anyone as long as she has the same name and age as his best friend. What we really want is a "variable" to equate the fourth and fifth arguments without concern for their internal structures. Figure 9 shows the feature structure formalization.

The boxed number indicates coreference. Features with values marked by the same coreference label share the *same* value (in the sense of LISP's EQ, not EQUAL). The value itself can be placed after any one of the coreference labels, the choice of which is arbitrary. The symbol [ ] indicates a variable. A variable [ ] can unify with any other feature structure *s*.

Like terms, feature structures can also be represented as directed graphs (see Section 3). This representation is often more useful for implementation or even presentation. Whereas term graphs are labeled on *vertices*, feature structure graphs have labels on *arcs* and *leaves*. Figure 10 shows the graph version of the feature structure of Figure 7. The graph of Figure 10, of course, is just a tree. In order to express something such as "John must marry a woman exactly as old as he is," we need coreference, indicated by the graph in Figure 11.

This ends the discussion of feature structures and terms. At this point, a few other structures are worth mentioning: *frames*, ψ-*terms*, and LISP functions. Frames [Minsky 1975], a popular representation in artifical intelligence, can be modeled with feature structures. We view a feature as a slot: Feature labels become *slot names* and feature values become *slot values*. As in the frame conception, a value may be either atomic or complex, perhaps having substructure of its own.

The ψ-terms of Aït-Kaci [1984, 1986] and Aït-Kaci and Nasr [1986] are very similar to feature structures. Subterms are labeled symbolically, and fixed arity is not required, as in feature structures, but

[4] This is construed in the sense of the term's *graph representation.*

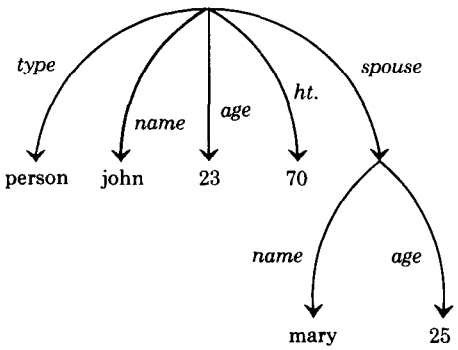

Figure 10. Graph representation of the feature structure shown in Figure 7.

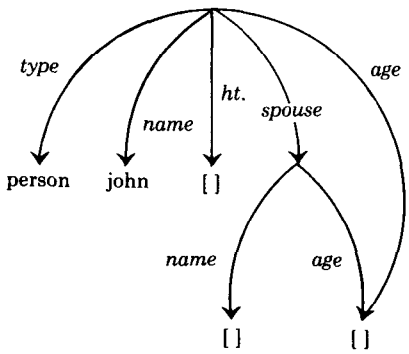

Figure 11. Graph representation of a feature structure with variables and coreference.

the functor is retained. And like feature structures, variables and coreference are handled separately. Although feature structures predate ψ-terms, Aït-Kaci's discussion of arity, variables, and coreference is superior to those found in the feature structure literature.

The novel aspect of ψ-terms is the use of type inheritance information. Aït-Kaci views a term's functions and variables as *filters* under unification, since two structures with different functors can *never* unify, whereas variables can unify with *anything*. He questions and relaxes this "open/closed" behavior by allowing *type information* to be attached to functions and variables. Unification uses information from a taxonomic hierarchy to achieve a more gradual filtering.

As an example, suppose we have the following inheritance information: Birds and fish are animals, a fish-eater is an animal, a trout is a fish, and a pelican is both a bird and a fish-eater. Then unifying the following two ψ-terms,

$$\textit{fish-eater}\ (\textit{likes} \Rightarrow \textit{trout})$$

$$\textit{bird}\ (\textit{color} \Rightarrow \textit{brown};\ \textit{likes} \Rightarrow \textit{fish})$$

yields the new ψ-term

$$\textit{pelican}\ (\textit{color} \Rightarrow \textit{brown};\ \textit{likes} \Rightarrow \textit{trout})$$

Unification does not fail when it sees conflicts between "fish-eater" and "bird" or between "trout" and "fish." Instead, it resolves the conflict by finding the *greatest lower bound* on the two items in the taxonomic hierarchy, in this case "pelican" and "trout," respectively. In this way, Aït-Kaci's system naturally extends the information-merging nature of unification.

As a third structure, I turn to the *function* in the programming language LISP. Functions are not directly related to the feature structures, but some interesting parallels

can be drawn. Older dialects of LISP required that parameters to functions be passed in an arbitrary, specified order. Common LISP [Steele 1984], however, includes the notion of *keyword parameter*. This allows parameters to be passed along with symbolic names; for example,

```
(make-node :state '( 1 2 3 ):
    score 12 :successors nil)
```

which is equivalent to

```
(make-node :score 12 :state '( 1 2 3 ):
    successors nil)
```

Keyword parameters correspond to the explicit labeling of substructure discussed above. Note that Common LISP functions are not *required* to use the keyword parameter format. Common LISP also allows for optional parameters to a function, which seems to correspond to the lack of fixed arity in feature structures. Common LISP, however, is not quite as flexible, since all possible optional parameters must be specified in advance.

Finally, feature structures may contain variables. LISP is a functional language, and that means *a function's parameters must be fully instantiated before the function can begin its computation.* In other words, variable binding occurs in only one direction, from parameters to their values. Contrast this behavior with that of Prolog, in which parameter values may be temporarily unspecified.

8. UNIFICATION AND NATURAL LANGUAGE PROCESSING

This section makes extensive use of the ideas covered in Section 7. Only with this background am I able to present examples and literature on unification in natural language processing.

8.1 Parsing with a Unification-Based Grammar

Unification-based parsing systems typically contain grammar rules and lexical entries. Lexical entries define words, and grammar rules define ways in which words may be combined with one another to form larger units of language called *constituents*. Sample constituent types are the *noun phrase* (e.g., "the man") and the *verb phrase* (e.g., "kills bugs"). Grammar rules in natural language systems share a common purpose with grammar rules found in formal language theory and compiler theory: The description of how smaller constituents can be put together into larger ones.

Figure 12 shows a rule (called an augmented context-free grammar rule) taken from a sample unification-based grammar. It is called an augmented context-free grammar rule because at its core is the simple context-free rule $S \Rightarrow NP\ VP$, meaning that we can build a sentence (S) out of a noun phrase (NP) and a verb phrase (VP). The equations (the augmentation) serve two purposes: (1) to block applications of the rule in unfavorable circumstances and (2) to specify structures that should be created when the rule is applied.

The idea is this: The rule builds a feature structure called $X0$ using (already existent) feature structures $X1$ and $X2$. Each feature structure, in this case, has two substructures, one labeled *category*, the other labeled *head*. *Category* tells what general type of syntactic object a structure is, and *head* stores more information.

Let us walk through the equations. The first equation states that the feature structure $X0$ will have the *category* of S (sentence). The next two equations state that the rule only applies if the categories of $X1$ and $X2$ are NP and VP, respectively.

The fourth equation states that the number *agreement* feature of the *head* of $X1$ must be the same as the number *agreement* feature of the *head* of $X2$. Thus, this rule will not apply to create a sentence like "John swim." The fifth equation states that the *subject* feature of the *head* of $X0$ must be equal to $X1$; that is, after the rule has applied, the *head* of $X0$ should be a structure with at least one feature, labeled *subject*, and the value of that feature should be the *head* of the structure $X1$.

The sixth equation states that the *head* of $X0$ should contain all of the features of

$X0 \Rightarrow X1\ X2$
$\langle X0\ category \rangle = S$
$\langle X1\ category \rangle = NP$
$\langle X2\ category \rangle = VP$
$\langle X1\ head\ agreement \rangle = \langle X2\ head\ agreement \rangle$
$\langle X0\ head\ subject \rangle = \langle X1\ head \rangle$
$\langle X0\ head \rangle = \langle X2\ head \rangle$
$\langle X0\ head\ mood \rangle = declarative$

Figure 12. Sample rule from a unification-based grammar.

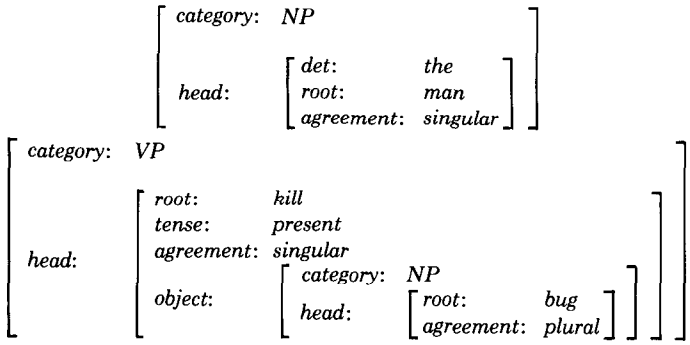

Figure 13. Feature structures representing analyses of "the man" and "kills bugs."

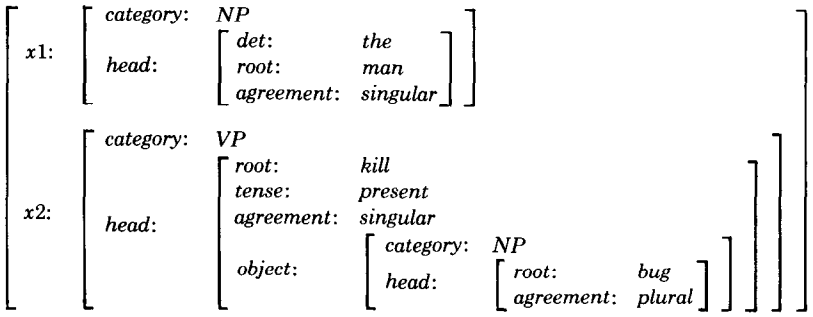

Figure 14. Two feature structures combined with "dummy" features $x1$ and $x2$.

the *head* of $X2$ (i.e., the sentence inherits the features of the verb phrase). The last equation states that the *head* of $X0$ should have a *mood* feature whose (atomic) value is *declarative*.

Now where does unification come in? Unification is the operation that *simultaneously* performs the two tasks of building up structure and blocking rule applications. Rule application works as follows:

(1) Gather constituent structures. Suppose we have already built up the two structures shown in Figure 13. These structures represent analyses of the phrases "the man" and "kills bugs." In parsing, we want to combine these structures into a larger structure of category S.

(2) Temporarily combine the constituent structures. *NP* and *VP* are combined into a single structure by way of "dummy" features $x1$ and $x2$ (Figure 14).

(3) *Represent the grammar rule itself as a feature structure.* The feature structure for the sample rule given above is shown in Figure 15. The boxed coreference labels enforce the equalities expressed in the rule's augmentation.

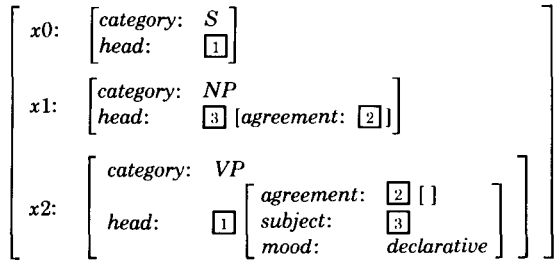

Figure 15. Feature structure representation of the grammar rule in Figure 12.

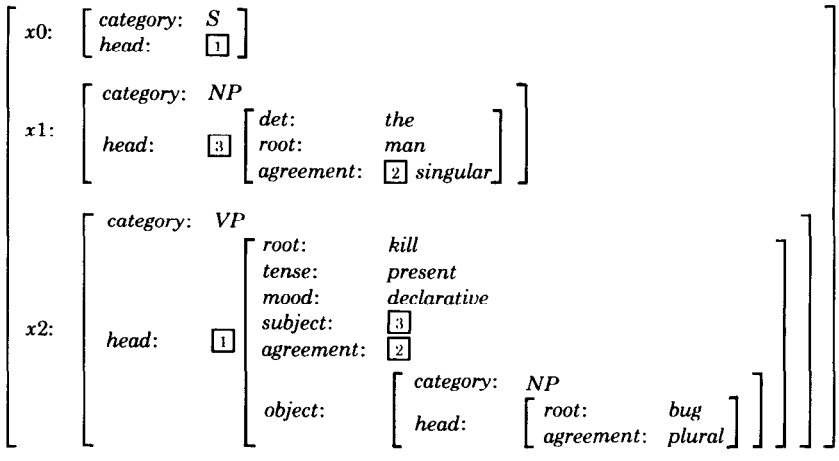

Figure 16. Result of unifying constituent and rule structures (Figures 14 and 15).

(4) Unify the constituent structure with the rule structure. We then get the structure shown in Figure 16. In this manner, unification builds larger syntactic constituents out of smaller ones.
(5) Retrieve the substructure labeled *X0*. In practice, this is the only information in which we are interested. (The final structure is shown in Figure 17.)

Now suppose the *agreement* feature of the original *VP* had been *plural.* Unification would have failed, since the *NP* has the same feature with a different (atomic) value—and the rule would fail to apply, blocking the parse of "The man kills bugs."

A few issues bear discussion. First, where did the structures *X1* and *X2* come from? In unification-based grammar, the *lexicon*, or dictionary, contains basic feature structures for individual words. Figure 18, for example, shows a possible lexical entry for the word "man." All higher level feature structures are built up from these lexical structures (and the grammar rules).

Second, how are grammar rules chosen? It seems that only a few rules can possibly apply successfully to a given set of structures. Here is where the *category* feature comes into play. Any parsing method, such as Earley's algorithm [Earley 1968, 1970] or generalized LR parsing [Tomita 1985a, 1985b, 1987], can direct the choice of grammar rules to apply, based on the *categories* of the constituents generated.

Third, what are the structures used for? Feature structures can represent syntactic, semantic, or even discourse-based information. Unification provides a kind of constraint-checking mechanism for merging information from various sources. The feature structure built up from the last rule application is typically the output of the parser.

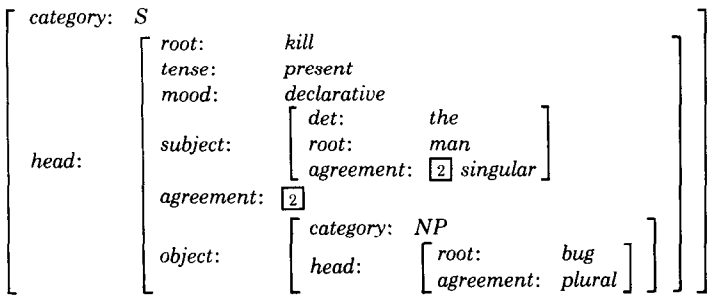

Figure 17. Feature structure analysis of "The man kills bugs."

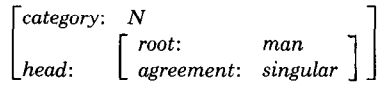

Figure 18. Sample lexical entry for the word "man."

Finally, why do the rules use equations rather than explicit tests and structure-building operations? Recall that unification both blocks rule applications if certain tests are not satisfied and builds up new structures. Another widely used grammar formalism, the Augmented Transition Network (ATN) of Woods [1970], separates these two activities. An ATN grammar is a network of *states*, and parsing consists of moving from state to state in that network. Along the way, the parser maintains a set of registers, which are simply variables that contain arbitrary values. When the parser moves from one state to another, it may *test* the current values of any registers and may, if the tests succeed, *change* the values of any registers. Thus, *testing* and *building* are separate activities.

An advantage of using unification, however, is that it is, by its nature, bidirectional. Since the equations are stated declaratively, a rule could be used to create *smaller* constituents from *larger* ones. One interesting aspect of unification-based grammars is the possibility *using the same grammar* to parse and generate natural language.

8.2 Research

An excellent, modern introduction to the use of unification in natural language processing is Shieber's [1986] book. This book presents motivations and examples far beyond the above exposition. The reader interested in this area is urged to find this source. Now I will discuss research on feature structures, unification algorithms, and unification-based grammar formalisms.

Karttunen [1984] discusses features and values at a basic level and provides linguistic motivation for both unification and generalization (see also Section 11). Structure-sharing approaches to speeding up unification are discussed by Karttunen and Kay [1985] and by Pereira [1985], who imports Boyer and Moore's [1972] approach for terms. Wroblewski [1987] gives a simple nondestructive unification algorithm that minimizes copying.

Much unpublished research has dealt with extensions to unification. For example, it is often useful to check for the presence or absence of a feature during unification rather than simply merging features. Other extensions include (1) *multiple-valued features*—a feature may take on more than one value, for example, "*big, red, round*"; (2) *negated features*:—a feature may be specified by the values it *cannot* have rather than one it *can* have; and (3) *disjunctive features*:—a feature may be specified by a set of values, at least *one* of which must be its true value.

Disjunctive feature structures are of great utility in computational linguistics, but their manipulation is difficult. Figure 19 shows an example of unification of disjunctive feature structures.

The curly brackets indicate disjunction; a feature's value may be any one of the structures inside the brackets. When performing unification, one must make sure

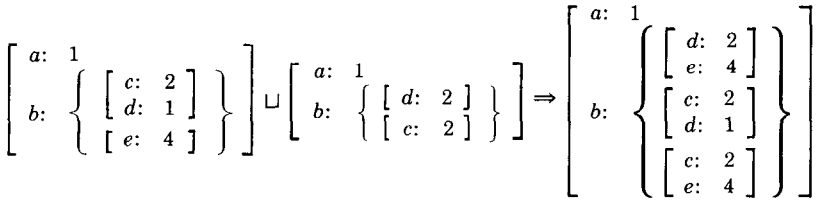

Figure 19. Unification of disjunctive feature structures.

that all possibilities are maintained. This usually involves cross multiplying the values of the two disjunctive features. In the above example, there are four possible ways to combine the values of the "b" feature; three are successful, and one fails (because of the conflicting assignment to the "d" feature).

The mathematical basis of feature structures has been the subject of intensive study. Pereira and Shieber [1984] developed a denotational semantics for unification grammar formalisms. Kasper and Rounds [1986] and Rounds and Kasper [1986] presented a logical model to describe feature structures. Using their formal semantics, they were able to prove that the problem of unifying disjunctive structures is $\mathcal{NP}$-complete. Disjunctive unification has great utility, however, and several researchers, including Kasper [1987] and Eisele and Dörre [1988], have come up with algorithms that perform well in the average case. Rounds and Manaster-Ramer [1987] discussed Kay's Functional Grammar in terms of his logical model. Moshier and Rounds [1987] extended the logic to deal with negation and disjunction by means of intuitionistic logic. Finally, in his dissertation, Aït-Kaci [1984] gave a semantic account of ψ-terms, logical constructs very close to feature structures—these constructs were discussed in Section 7.

The past few years have seen a number of new grammar formalisms for natural languages. A vast majority of these formalisms use unification as the central operation for building larger constituents out of smaller ones. Gazdar [1987] enumerated them in a brief survey, and Sells [1985] described the theories underlying two of the more popular unification grammar formalisms. Gazdar et al. [1987a] present a more complete bibliography. The formalisms can be separated roughly into families:

In the "North American Family," we have formalisms arising from work in computational linguistics in North America in the late 1970s. Functional Unification Grammar (FUG; previously FG and UG) is described by Kay [1979, 1984, 1985a]. Lexical Functional Grammar (LFG) is introduced in Bresnan's [1982] book.

The "Categorial Family" comes out of theoretical linguistic research on categorial grammar. Representatives are Unification Categorial Grammar (UCG), Categorial Unification Grammar (CUG), Combinatory Categorial Grammar (CCG), and Meta-Categorial Grammar (MCG). UCG was described by Zeevat [1987], CUG by Uzkoreit [1986], and CCG by Wittenburg [1986]. Karttunen [1986b] also presented work in this area.

The "GPSG Family" begins with Generalized Phrase Structure Grammar (GPSG), a modern version of which appears in Gazdar and Pullum [1985]. Work at ICOT in Japan resulted in a formalism called JPSG [Yokoi et al. 1986]. Pollard and colleagues have been working recently on Head-Driven Phrase Structure Grammar (HPSG); references include Pollard [1985] and Sag and Pollard [1987]. Ristad [1987] introduces Revised GPSG (RGPSG).

The "Logic Family" consists of grammar formalisms arising from work in logic programming. Interestingly, Prolog was developed with natural language applications in mind: Colmerauer's [1978, 1982a] Metamorphosis Grammars were the first grammars in this family. Pereira and Warren's [1980] Definite Clause Grammar (DCG) has become extremely popular. Other logic grammars include Extraposition Grammar

(XG) [Pereira, 1981], Gapping Grammars, [Dahl 1984; Dahl and Abramson 1984] Slot Grammar [McCord 1980], and Modular Logic Grammar (MLG) [McCord 1985].

Prolog has been used in a variety of ways for analyzing natural languages. Definite Clause Grammar (DCG) is a basic extension of Prolog, for example. Bottom-up parsers for Prolog are described in Matsumoto and Sugimura [1987], Matsumoto et al. [1983], and Stabler [1983]. Several articles on logic programming and natural language processing are collected in Dahl and Saint-Dizier [1985]. Implementations of LFG in Prolog are described in Eisele and Dörre [1986] and Reyle and Frey [1983].

Pereira [1987] presented an interesting investigation of unification grammars from a logic programming perspective. In a similar vein, Kay [1985b] explained the need to go beyond the capabilities of unification present in current logic programming systems.

Finally, in the "miscellaneous" category, we have PATR-II, an (implemented) grammar formalism developed by Karttunen [1986b], Shieber [1984, 1985], and others. PATR is actually an environment in which grammars can be developed for a wide variety of unification-based formalisms. In fact, many of the formalisms listed above are so similar that they can be translated into each other automatically; for example, see Reyle and Frey [1983]. Hirsh [1986] described a compiler for PATR grammars.

9. UNIFICATION AND EQUATIONAL THEORIES

Unification under equational theories is a very active area of research. The research has been so intense that there are several papers devoted entirely to classifying previous research results [e.g., Siekmann 1984, 1986; Walther 1986]. Many open problems remain, and the area promises to provide new problems for the next decade.

9.1 Unification as Equation Solving

At the very beginning of this paper, I presented an example of unification. The terms were $s = f(x, y)$ and $t = f(g(y, a), h(a))$, and the most general unifier σ was

$$x \leftarrow g(h(a), a)$$
$$y \leftarrow h(a)$$

From an algebraic viewpoint, we can think of this unification as solving the equation $s = t$ by determining appropriate values for the variables x and y.

Now, for a moment, assume that f denotes the function *add*, g denotes the function *multiply*, h denotes the function *successor*, and a denotes the constant *zero*. Also assume that all the axioms of number theory hold. In this case, the equation $s = t$ is interpreted as $add(x, y) = add(mult(y, 0), succ(0))$. It turns out that there are many solutions to the equation, including the substitution τ:

$$x \leftarrow h(a)$$
$$y \leftarrow a$$

Notice that $\tau(s)$ is $f(h(a), a)$ and that $\tau(t)$ is $f(g(a, a), h(a))$. These resulting two terms are not *textually identical*, but under the interpretation of f, g, h, and a given above, they are certainly *equivalent*. The former is $succ(0) + 0$, or $succ(0)$; the latter is $(0 \cdot 0) + succ(0)$, or $succ(0)$. Therefore, we say that τ unifies s and t *under the axioms of number theory*. It is clear that determining whether two terms are unifiable under the axioms of number theory is the same problem as solving equations in number theory.

Number theory has complex axioms and inference rules. Of special interest to unification are simpler equational axioms such as

$f(f(x, y), z) = f(x, f(y, z))$	*associativity*
$f(x, y) = f(y, x)$	*commutativity*
$f(x, x) = x$	*idempotence*

A *theory* is a finite collection of axioms such as the ones above. The problem of unifying two terms under theory T is written $\langle s = t \rangle_T$.

Consider the problem of unification under commutativity: $\langle s = t \rangle_C$. To unify $f(x, y)$ with $f(a, b)$ we have *two* substitutions available to us: $\{x \leftarrow a; y \leftarrow b\}$ and $\{x \leftarrow b; y \leftarrow a\}$. With commutativity, we no

longer get the unique *most general* unifier of Robinson's null-theory unification—compare this situation to Gould's findings on higher order unification.

There are many applications for unification algorithms specialized for certain equational theories. A theorem prover, for example, may be asked to prove theorems about the multiplicative properties of integers, in which case it is critical to know that the multiplication function is *associative*. One could add an "associativity axiom" to the theorem prover, but it might waste precious inferences simply bracketing and rebracketing multiplication formulas. A better approach would be to "build in" the notion of associativity at the core of the theorem prover: the unification algorithm. Plotkin [1972] pioneered this area, and it has since been the subject of much research.

9.2 Research

Siekmann's [1984] survey of unification under equational theories is comprehensive. The reader is strongly urged to seek out this source. I will mention here only a few important results.

We saw above that unification under commutativity can produce more than one general unifier; however, there are always a *finite* number of general unifiers [Livesey et al. 1979]. Under associativity, on the other hand, there may be an *infinite* number of general unifiers [Plotkin 1972]. Baader [1986] and Schmidt-Schauss [1986] independently showed that unification under associativity *and* idempotence has the "unpleasant" feature that no complete set of general unifiers even *exists*. (That is, there are two unifiable terms s and t, but for every unifier σ, there is a more general unifier τ.)

Results of this kind place any equational theory T into a "hierarchy" of theories, according to the properties of unification under T. Theories may be *unitary* (one MGU), *finitary* (finite number of general unifiers), *infinitary* (infinite number of general unifiers), or *nullary* (no complete set of general unifiers). Siekmann [1984] surveys the classifications of many naturally arising theories. The study of Universal Unification seeks to discover properties of unification that hold across a wide variety of equational theories, just as Universal Algebra abstracts general properties from individual algebras.

To give a flavor for results in Universal Unification, I reproduce a recent theorem due to Book and Siekmann [1986]:

Theorem 9.1

If T is a suitable first-order equation theory that is not unitary, then T is not bounded.

This means the following: Suppose that unification under theory T produces a finite number of general unifiers but no MGU (as with commutativity). Then, there is no particular integer n such that for *every* pair s and t, the number of general unifiers is less than n (T is not bounded).

Term-rewriting systems, systems for translating sets of equations into sets of rewrite rules, are basic components of unification algorithms for equational theories. Such systems were the object of substantial investigation in Knuth and Bendix [1970]. A survey of work in this area is given in Huet and Oppen [1980].

Some recent work [Goguen and Meseguer 1987; Meseguer and Goguen 1987; Smolka and Aït-Kaci 1987; Walther 1986] investigates the equational theories of order-sorted and many-sorted algebras. Smolka and Aït-Kaci [1987] show how unification of Aït-Kaci's ψ-terms (see Section 7) is an instance of unification in order-sorted Horn logic. In other words, unification with inheritance can be seen as unification with respect to a certain set of equational axioms.

Unification algorithms have been constructed for a variety of theories. Algorithms for particular theories, however, have usually been handcrafted and based on completely different techniques. There is interest in building more general (but perhaps inefficient) "universal" algorithms [e.g., Fages and Huet 1986; Fay 1979; Gallier and Snyder 1988; Kirchner 1986]. There is also interest in examining the computational complexity of unification

under various theories [e.g., Kapur and Narendran 1986].

10. PARALLEL ALGORITHMS FOR UNIFICATION

Lewis and Statman [1982] wrote a note showing that nonunifiability could be computed in nondeterministic log space (and thus, by Savitch's theorem, in deterministic $\log^2$ space). By way of the *parallel computation thesis* [Goldschlager 1978], which relates sequential space to parallel time, this result could have led to a $\log^2 n$ parallel time algorithm for unification.

In the process of trying to find such an algorithm, Dwork et al. [1984] found an error in the above-mentioned note. They went on to prove that unifiability is actually log space complete for $\mathscr{P}$, which means that it is difficult to compute unification using a small amount of space. This means that it is highly unlikely that unification can be computed in $O(\log^k n)$ parallel time using a polynomial number of processors (in complexity terminology, unification is in the class $\mathscr{NC}$ only if $\mathscr{P} = \mathscr{NC}$).

In other words, even massive parallelism will not significantly improve the speed of unification—it is an inherently sequential process. This lower bound result has a wide range of consequences; for example, using parallel hardware to speed up Prolog execution is unlikely to bring significant gains (but see Robinson [1985] for a hardware implementation of unification).

Yasuura [1984] independently derived a complexity result similar to that of Dwork et al. [1984] and gave a parallel unification algorithm that runs in time $O(\log^2 n + m \log m)$, where m is the *total* number of variables in the two terms.

Dwork et al. [1984] mentioned the existence of a polylog parallel algorithm for *term matching*, the variant of unification in which substitution is allowed into only one of the two terms. They gave an $O(\log^2 n)$ time bound but required about $O(n^5)$ processors. Using randomization techniques, Dwork et al. [1986] reduced the processor bound to $O(n^3)$.

Maluszynski and Komorowski [1985], motivated by Dwork et al.'s [1984] negative results, investigated a form of logic programming based on matching rather than unification.

Vitter and Simons [1984, 1986] argued that unification *can* be helped by multiple processors in a practical setting. They developed complexity classes to formalize this notion of "practical" speedup and gave a parallel unification algorithm that runs in time $O(E/P + V \log P)$, where P is the number of processors, E the number of edges in the term graph, and V the number of vertices.

Harland and Jaffar [1987] introduced another measure for parallel efficiency and compared several parallel algorithms including Yasuura's algorithm [1984], Vitter and Simon's algorithm [1984, 1986], and a parallel version of Jaffar's algorithm [1984].

11. UNIFICATION, GENERALIZATION, AND LATTICES

Generalization is the dual of unification, and it finds applications in many areas in which unification is used. Abstractly, the generalization problem is the following: Given two objects x and y, can we find a third object z of which both x and y are instances? Formally it is as follows:

Definition 11.1

An *antisubstitution* (denoted $\gamma, \eta, \zeta, \ldots$), is a mapping from terms into variables.

Definition 11.2

Two terms s and t are *generalizable* if there exists an antisubstitution γ such that $\gamma(s) = \gamma(t)$. In such a case, γ is called a *generalizer* of s and t, and $\gamma(s)$ is called a *generalization* of s and t.

Generalizability is a rather vacuous concept, since *any* two terms are generalizable under the antisubstitution that maps all terms onto the variable x. Of more interest is the following:

Definition

A generalizer γ of terms s and t is called the *most specific generalizer* (MSG) of s and

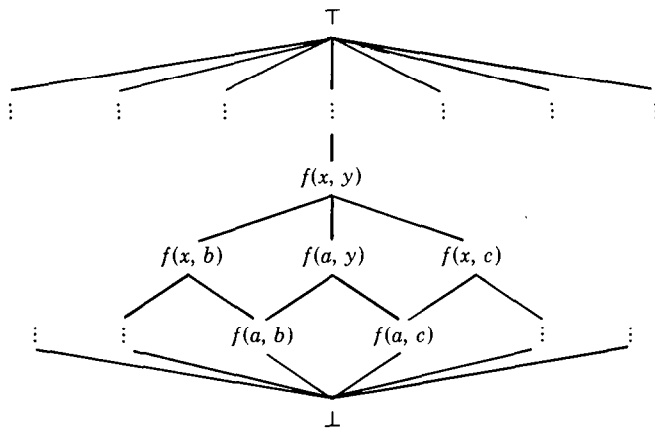

Figure 20. A portion of the lattice of first-order terms.

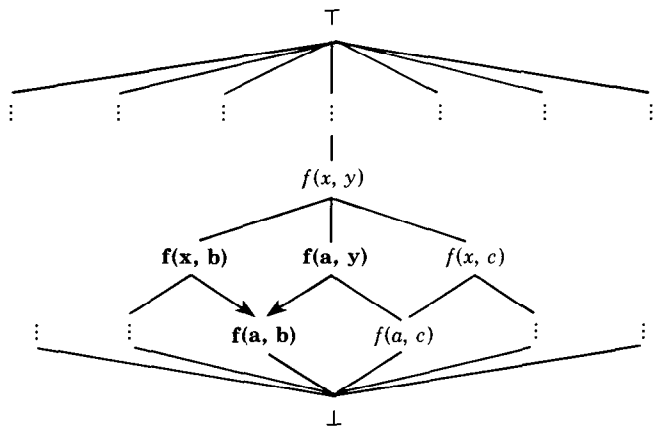

Figure 21. Unification of terms $f(x, b)$ and $f(a, y)$.

t if for any other generalizer η, there is an antisubstitution ζ such that $\zeta\gamma(s) = \eta(s)$.

Intuitively, the most specific generalization of two terms retains information that is common to both terms, introducing new variables when information conflicts. For example, the most specific generalization of $f(a, g(b, c))$ and $f(b, g(x, c))$ is $f(z, g(x, c))$. Since the second argument to f can be a or b, generalization "makes an abstraction" by introducing the variable z. Unification, on the other hand, would *fail* because of this conflict.

As mentioned in a previous section, Reynolds [1970] proved the existence of a unique MSG for first-order terms and came up with a generalization algorithm. (Plotkin [1970] independently discovered this algorithm.) Reynolds used the natural lattice structure of first-order terms, a partial ordering based on "subsumption" of terms. General terms subsume more specific terms; for example, $f(x, a)$ subsumes $f(b, a)$. Many pairs of terms, of course, do not stand in any subsumption relation, and this is where unification and generalization come in. Figure 20 shows a portion of the lattice of first-order terms augmented with two special terms called *top* ($\top$) and *bottom* ($\bot$).

Unification corresponds to finding the *greatest lower bound* (or *meet*) of two terms in the lattice. Figure 21 illustrates the

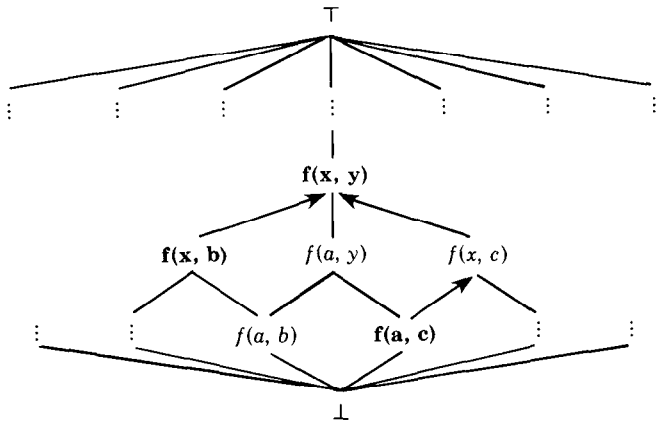

Figure 22. Generalization of terms $f(x, b)$ and $f(a, c)$.

unification of $f(x, b)$ and $f(a, y)$. The bottom of the lattice ($\perp$), to which all pairs of terms can unify, represents *inconsistency*. If the greatest lower bound of two terms is $\perp$, then they are *not unifiable*.

Generalization corresponds to finding the *least upper bound* (or *join*) of two terms in the lattice. Figure 22 illustrates the generalization of $f(x, b)$ and $f(a, c)$. The top of the lattice ($\top$), to which all pairs of terms can generalize, is called the *universal term*. Everything is an instance of this term.

Feature structures, previously discussed in Section 7, make up a more complex lattice structure.

Inheritance hierarchies, common structures in artificial intelligence, can be seen as lattices that admit unification and generalization. For example, the generalization of concepts "bird" and "fish-eater" in a certain knowledge base might be "animal," whereas the unification might be "pelican." Aït-Kaci extends this kind of unification and generalization to typed record structures (ψ-terms), as discussed in Section 7.

12. OTHER APPLICATIONS OF UNIFICATION

I have already discussed three application areas for unification: automatic theorem proving, logic programming, and natural language processing. Other areas are listed here.

12.1 Type Inference

Milner [1978] presented work on compile-time type checking for programming languages, with the goal of equating syntactic correctness with well typedness. His algorithm for well typing makes use of unification, an idea originally due to Hindley [1969]. More recent research in this area includes the work of Cardelli [1984], MacQueen et al. [1984], and Moshier and Rounds [1987].

12.2 Programming Languages

Pattern matching is a very useful feature of programming languages. Logic programming languages make extensive use of pattern matching, as discussed above, but some functional languages also provide matching facilities (e.g., PLANNER, QA4/QLISP). In these cases, functions may be invoked in a pattern-directed manner, making use of unification as a pattern-matcher/variable-binder. Stickel [1978] discusses pattern-matching languages and specialized unification algorithms in his dissertation.

12.3 Machine Learning

Learning concepts from training instances is a task that requires the ability to generalize. Generalization, the dual of unification, is an operation widely used in machine learning.

Plotkin [1970], who opened up work in unification under equational theories, also produced work on inductive generalization, that is, abstracting general properties from a set of specific instances. Mitchell's [1979] work on version spaces investigated the same problem using both positive and negative training instances. Positive examples force the learner to generate more general hypotheses, whereas negative examples force more specific hypotheses. The general-to-specific lattice (see Section 11) is especially useful, and Mitchell's algorithms make use of operations on this lattice.

13. CONCLUSION

This section contains a list of properties that unification possesses along with a summary of the trends in research on unification.

13.1 Some Properties of Unification

The following properties hold for first-order unification:

Unification is monotonic. Unification adds information, but never subtracts. It is impossible to remove information from one structure by unifying it with another. This is not the case with generalization.

Unification is commutative and associative. The order of unifications is irrelevant to the final result. Unification and generalization are not, however, distributive with respect to each other [Shieber 1986].

Unification is a constraint-merging process. If structures are viewed as encoding constraint information, then unification should be viewed as merging constraints. Unification also detects when combinations of certain constraint sets are inconsistent.

Unification is a pattern-matching process. Unification determines whether two structures match, using the mechanism of variable binding.

Unification is bidirectional since variable binding may occur in both of the structures to be unified. Matching is the unidirectional variant of unification and appears in the "function call" of imperative programming languages.

Unification deals in partially defined structures. Unification, unlike many other operations, accepts inputs that contain uninstantiated variables. Its output may also contain uninstantiated variables.

13.2 Trends in Unification Research

One trend in unification research seeks to discover faster algorithms for first-order unification.[5] Over the past two decades, many quite different algorithms have been presented, and although the worst-case complexity analyses are very interesting, it is still unclear which algorithms work best in practice. There seems to be no better way to compare algorithms than to implement and test them on practical problems. As it stands, which algorithm is fastest depends on the kinds of structures that are typically unified.

Another trend, begun by Robinson [1968], tackles the problems of unification in higher order logic. Such problems are in general unsolvable, and the behaviors of unification algorithms are less quantifiable. Systems that deal with higher order logic, for example theorem provers and other AI systems [Miller and Nadathur 1987], stand to gain a great deal from research in higher order unification.

A third trend runs toward incorporating more features into the unification operation. Automatic theorem provers once included separate axioms for commutativity, associativity, and so on, but research begun by Plotkin [1972] showed that building these notions directly into the unification routine yields greater efficiency. Logic programming languages are often called on to reason about hierarchically organized data, but they do so in a step-by-step fashion. Aït-Kaci and Nasr [1986] explain how to modify unification to use inheritance information, again improving efficiency. Many unification routines for natural language parsing perform unification over negated,

[5] This trend toward efficiency also explores topics such as structure sharing and space efficiency.

multiple-valued, and disjunctive structures. New applications for unification will no doubt drive further research into new, more powerful unification algorithms.

ACKNOWLEDGMENTS

I would like to thank Yolanda Gil, Ed Clarke, and Frank Pfenning for their useful comments on earlier drafts of this paper. My thanks also go to the reviewers for making many suggestions relating to both accuracy and clarity. Finally, I would like to thank Masaru Tomita for introducing me to this topic and for helping to arrange this survey project.

REFERENCES

AHO, V. A., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

ANDREWS, P. B. 1971. Resolution in type theory. *J. Symbolic Logic 36*, pp. 414–432.

ANDREWS, P. B., AND COHEN, E. L. 1977. Theorem proving in type theory. In *Proceedings of International Joint Conference on Artificial Intelligence*.

ANDREWS, P., MILLER, D., COHEN, E., AND PFENNING, F. 1984. Automating higher-order logic. Vol. 29. In *Automated Theorem Proving: After 25 Years*, W. Bledsoe and D. Loveland, Eds. American Mathematical Society, Providence, R.I., Contemporary Mathematics Series.

AÏT-KACI, H. 1984. A lattice theoretic approach to computation based on a calculus of partially ordered type structures. Ph.D. dissertation. Univ. of Pennsylvania, Philadelphia, Pa.

AÏT-KACI, H. 1986. An algebraic semantics approach to the effective resolution of type equations. *Theor. Comput. Sci. 45*.

AÏT-KACI, H., AND NASR, R. 1986. LOGIN: A logic programming language with built-in inheritance. *J. Logic Program. 3*. (Also MCC Tech. Rep. AI-068-85, 1985.)

BAADER, F. 1986. The theory of idempotent semigroups is of unification type zero. *J. Autom. Reasoning 2*, pp. 283–286.

BARWISE, J., AND PERRY, J. 1983. *Situations and Attitudes*. MIT Press, Cambridge, Mass.

BATTANI, G., AND MELONI, H. 1973. Interpreteur du langage de programmation PROLOG. Groupe Intelligence Artificielle, Université Aix-Marseille II.

BAXTER, L. 1973. An efficient unification algorithm. Tech. Rep. No. CS-73-23, Univ. of Waterloo, Waterloo, Ontario, Canada.

BAXTER, L. 1976. The complexity of unification. Ph.D. dissertation. Univ. of Waterloo, Waterloo, Ontario, Canada.

BAXTER, L. 1978. The undecidability of the third order dyadic unification problem. *Inf. Control 38*, pp. 170–178.

BOOK, R. V., AND SIEKMANN, J. 1986. On unification: Equational theories are not bounded. *J. Symbolic Comput. 2*, pp. 317–324.

BOYER, R. S., AND MOORE, J. S. 1972. The sharing of structure in theorem-proving programs. *Mach. Intell. 7*.

BRESNAN, J., AND KAPLAN, R. 1982. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed. MIT Press, Cambridge, Mass.

BRUYNOOGHE, M., AND PEREIRA, L. M. 1984. Deduction revision by intelligent backtracking. In *Prolog Implementation*, J. A. Campbell, Ed. Elsevier, North-Holland, New York.

CARDELLI, L. 1984. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science*, No. 173, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds., Springer-Verlag, New York.

CHEN, T. Y., LASSEZ, J., AND PORT, G. S. 1986. Maximal unifiable subsets and minimal non-unifiable subsets. *New Generation Comput. 4*, pp. 133–152.

CHURCH, A. 1940. A formulation of the simple theory of types. *J. Symbolic Logic 5*, pp. 56–68.

CLOCKSIN, W. F., AND MELLISH, C. S. 1981. *Programming in Prolog*. Springer-Verlag, New York.

COLMERAUER, A. 1978. Metamorphosis grammars. In *Natural Language Communication with Computers*, L. Bolc, Ed. Springer-Verlag, New York.

COLMERAUER, A. 1982a. An interesting subset of natural language. In *Logic Programming*, K. L. Clark and S. Tärnlund, Eds. Academic Press, London.

COLMERAUER, A. 1982b. Prolog and infinite trees. In *Logic Programming*, K. L. Clark and S. Tärnlund, Eds. Academic Press, Orlando, Fla.

COLMERAUER, A. 1983. Prolog in ten figures. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

COLMERAUER, A., KANOUI, H., AND VAN CANEGHEM, M. 1973. Un systeme de communication homme-machine en Francais. Research Rep. Groupe Intelligence Artificielle, Université Aix-Marseille II.

CORBIN, J., AND BIDOIT, M. 1983. A rehabilitation of Robinson's unification algorithm. *Inf. Process. 83*, pp. 73–79.

COURCELLE, B. 1983. Fundamental properties of infinite trees. *Theor. Comput. Sci. 25*, pp. 95–169.

COX, P. T. 1984. Finding backtrack points for intelligent backtracking. In *Prolog Implementation*, J. A. Campbell, Ed. Elsevier, North-Holland, New York.

DAHL, V. 1984. More on gapping grammars. In *Proceedings of the International Conference on Fifth*

Generation Computer Systems. Sponsored by Institute for New Generation Computer Technology (ICOT).

DAHL, V., AND ABRAMSON. 1984. On gapping grammars. In *Proceedings of the 2nd International Conference on Logic Programming*. Sponsored by Uppsala University, Uppsala, Sweden.

DAHL, V., AND SAINT-DIZIER, P., EDS. 1985. *Natural Language Understanding and Logic Programming*. Elsevier North-Holland, New York.

DARLINGTON, J. 1968. Automatic theorem proving with equality substitutions and mathematical induction. *Mach. Intell. 3*. Elsevier, New York.

DARLINGTON, J. 1971. A partial mechanization of second-order logic. *Mach. Intell. 6*.

DARLINGTON, J. 1977. Improving the efficiency of higher order unification. In *Proceedings of the International Joint Conference on Artifical Intelligence*.

DE CHAMPEAUX, D. 1986. About the Paterson–Wegman linear unification algorithm. *J. Comput. Syst. Sci. 32*, pp. 79–90.

DWORK, C., KANELLAKIS, P. C., AND MITCHELL, J. C. 1984. On the sequential nature of unification. *J. Logic Program. 1*, pp. 35–50.

DWORK, C., KANELLAKIS, P. C., AND STOCKMEYER, L. 1986. Parallel algorithms for term matching. In *Proceedings of the 8th International Conference on Automated Deduction*.

EARLEY, J. 1968. An efficient context-free parsing algorithm. Ph.D. dissertation, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

EARLEY, J. 1970. An efficient context-free parsing algorithm. *Commun. ACM 6*, pp. 94–102.

EISELE, A., AND DÖRRE, J. 1986. A lexical functional grammar system in Prolog. In *Proceedings of the International Conference on Computational Linguistics*. North-Holland, Amsterdam, New York.

EISELE, A., AND DÖRRE, J. 1988. Unification of disjunctive feature descriptions. In *Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics*.

FAGES, F., AND HUET, G. 1986. Complete sets of unifiers and matchers in equational theories. *Theor. Comput. Sci. 43*, pp. 189–200.

FARMER, W. M. 1987. A unification algorithm for second-order monadic terms. Tech. Rep. (forthcoming). The MITRE Corporation, document MTP-253.

FAY, M. 1979. First order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction*, Austin, Texas.

GALLIER, J. H., AND SNYDER, W. 1988. Complete sets of transformations for general E-unification. Tech. Rep. MS-CIS-88-72-LINCLAB-130. Dept. of Computer and Information Science, Univ. of Pennsylvania, Philadelphia, Pa.

GAZDAR, G. 1987. The new grammar formalisms—A tutorial survey (abstract). In *Proceedings of the International Joint Conference on Artificial Intelligence*.

GAZDAR, G., AND PULLUM, G. K. 1985. Computationally relevant properties of natural languages and their grammars. *New Generation Comput. 3*, pp. 273–306.

GAZDAR, G., FRANZ, A., OSBORNE, K., AND EVANS, R. 1987. *Natural Language Processing in the 1980's—A Bibliography*. CSLI Lecture Notes Series, Center for the Study of Language and Information, Stanford, California.

GOGUEN, J. A., AND MESEGUER, J. 1987. Order-sorted algebra solves the constructor-selector multiple representation and coercion problems. Tech. Rep. CSLI-87-92, Center for the Study of Language and Information.

GOLDFARB, W. D. 1981. The undecidability of the second order unification problem. *J. Theoret. Comput. Sci. 13*, pp. 225–230.

GOLDSCHLAGER, L. 1978. A unified approach to models of synchronous parallel machines. In *Proceedings of the Symposium on the Theory of Computing*. Sponsored by ACM Special Interest Group for Automata and Computability Theory (SIGACT).

GOULD, W. E. 1966a. A matching procedure for ω-order logic. Ph.D. dissertation. Princeton Univ., Princeton, N.J.

GOULD, W. E. 1966b. A matching procedure for ω-order logic. Scientific Rep. 4, AFCRL 66–781.

GUARD, J. R. 1964. Automated logic for semi-automated mathematics. Scientific Rep. 1, AFCRL 64–411.

HARLAND, J., AND JAFFAR, J. 1987. On Parallel Unification for Prolog. *New Generation Comput. 5*.

HASIDA, K. 1986. Conditioned unification for natural language processing. In *Proceedings of the International Conference on Computational Linguistics*. North-Holland, Amsterdam, New York.

HENKIN, L. 1950. Completeness in the theory of types. *J. Symbolic Logic 15*, pp. 81–91.

HERBRAND, J. 1971. Recherches sur la theorie de la demonstration. Ph.D. dissertation. In *Logical Writings*, W. Goldfarb, Ed. Harvard University Press, Cambridge, Massachusetts.

HINDLEY, R. 1969. The principal type-scheme of an object in combinatory logic. *Trans. Am. Math. Soc. 146*.

HIRŠH, S. B. 1986. P-PATR: A compiler for unification-based grammars. Center for the Study of Language and Information.

HUET, G. 1972. Contrained resolution: A complete method for higher order logic. Ph.D. dissertation. Case Western Reserve Univ., Cleveland, OH.

HUET, G. 1973a. A mechanization of type theory. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

HUET, G. 1973b. The undecidability of unification in third order logic. *Inf. Control 22*, pp. 257–267.

HUET, G. 1975. A unification algorithm for typed λ-calculus. *Theoret. Comput. Sci. 1*, pp. 27–57.

HUET, G. 1976. Resolution d'equations dans les langages d'ordre 1, 2, . . . , ω. Ph.D. dissertation. Univ. de Paris VII, France.

HUET, G., AND OPPEN, D. 1980. Equations and rewrite rules: A survey. In *Formal Language Theory*, R. V. Book, Ed. Academic Press, Orlando, Fla.

JAFFAR, J. 1984. Efficient unification over infinite terms. *New Generation Comput. 2*, pp. 207–219.

JENSEN, D. C., AND PIETRZYKOWSKI, T. 1976. Mechanizing ω-order type theory through unification. *Theoret. Comput. Sci. 3*, pp. 123–171.

KAPUR, D., AND NARENDRAN, P. 1986. NP-completeness of the set unification and matching problems. In *Proceedings of the 8th International Conference on Automated Deduction*. Springer-Verlag, New York.

KAPUR, D., KRISHNAMOORTHY, M. S., AND NARENDRAN, P. 1982. A new linear algorithm for unification. Tech. Rep. 82CRD-100, General Electric.

KARTTUNEN, L. 1984. Features and values. In *Proceedings of the International Conference on Computational Linguistics*.

KARTTUNEN, L. 1986a. D-PATR: A development environment for unification-based grammars. In *Proceedings of the International Conference on Computational Linguistics*. North-Holland, Amsterdam, New York.

KARTTUNEN, L. 1986b. Radical lexicalism. Tech. Rep. CSLI-86-68, Center for the Study of Language and Information.

KARTTUNEN, L., AND KAY, M. 1985. Structure sharing with binary trees. In *Proceedings of the 23rd Annual Meeting of the of the Association for Computational Linguistics*.

KASPER, R. 1987. A unification method for disjunctive feature descriptions. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics*.

KASPER, R., AND ROUNDS, W. C. 1986. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*.

KAY, M. 1979. Functional grammar. In *5th Annual Meeting of the Berkeley Linguistic Society*. Sponsored by Berkeley Linguistics Society, Berkeley, California.

KAY, M. 1984. Functional unification grammar: A formalism for machine translation. In *Proceedings of the International Conference on Computational Linguistics*. North-Holland, Amsterdam, New York.

KAY, M. 1985a. Parsing in functional grammar. In *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge Univ. Press, Cambridge.

KAY, M. 1985b. Unification in grammar. In *Natural Language Understanding and Logic Programming*, V. Dahl and P. Saint-Dizier, Eds. Elsevier, North-Holland, New York.

KIRCHNER, C. 1986. Computing unification algorithms. In *1st Symposium on Logic in Computer Science*. Co-sponsored by IEEE Computer Society, Technical Committee on Mathematical Foundations of Computing; ACM Special Interest Group for Automata and Computability Theory (SIGACT); Association for Symbolic Logic; European Association for Theoretical Computer Science.

KNUTH, D., AND BENDIX, P. B. 1970. Simple word problems in universal algebra. In *Computational Problems in Abstract Algebra*, J. Leech, Ed. Pergamon Press, Oxford.

LEVY, J. 1983. A unification algorithm for concurrent Prolog. In *Proceedings of the 2nd Logic Programming Conference*. Sponsored by Uppsala University, Uppsala, Sweden.

LEWIS, H. R., AND STATMAN, R. 1982. Unifiability is complete for co-NLogSpace. *Info. Process. Lett. 15*, pp. 220–222.

LIVESEY, M., SIEKMANN, J., SZABO, P., AND UNVERICHT, E. 1979. Unification problems for combinations of associativity, commutatitivity, distributivity and idempotence axioms. In *Proceedings of the 4th Workshop on Automated Deduction*, Austin, Texas.

LUCCHESI, C. L. 1972. The undecidability of the unification problem for third order languages. Tech. Rep. CSRR 2059, Dept. of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

MACQUEEN, D., PLOTKIN, G., AND SETHI, R. 1984. An ideal model for recursive polymorphic types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York.

MALUSZYNSKI, J., AND KOMOROWSKI, H. J. 1985. Unification-free execution of Horn-clause programs. In *Proceedings of the 2nd Logic Programming Symposium*. IEEE, New York. (Also, Harvard Univ., Computer Science Dept., Tech. Rep. TR-10-85, 1985.)

MANNILA, H., AND UKKONEN, E. 1986. On the complexity of unification sequences. In *Proceedings of the 3rd International Logic Programming Conference*. Springer-Verlag, New York.

MARTELLI, A., AND MONTANARI, U. 1976. Unification in linear time and space: A structured presentation. Internal Rep. No. B76-16, Ist. di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy.

MARTELLI, A., AND MONTANARI, U. 1977. Theorem proving with structure sharing and efficient unification. In *Proceedings of International Joint Conference on Artificial Intelligence*. (Also Internal Rep. No. S-77-7, Istituto di Scienze dell'Informazione, Univ. of Pisa, Italy).

MARTELLI, A., AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Prog. Lang. Syst. 4.*

MATSUMOTO, Y., AND SUGIMURA, R. 1987. A parsing system based on logic programming. In *Pro-*

ceedings of the International Joint Conference on Artificial Intelligence.

MATSUMOTO, Y., TANAKA, H., HIRAKAWA, H., MIYOSHI, H., AND YASUKAWA, H. 1983. BUP: A bottom-up parser embedded in Prolog. *New Generation Comput. 1*, pp. 145–158.

MATWIN, S., AND PIETRZYKOWSKI, T. 1982. Exponential improvement of exhuastive backtracking: Data structure and implementation. In *Proceedings of the 6th International Conference on Automated Deduction.* Springer-Verlag, New York.

MCCORD, M. C. 1980. Slot grammars. *Comput. Linguist. 6*, pp. 31–43.

MCCORD, M. C. 1985. Modular logic grammars. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics.*

MESEGUER, J., GOGUEN, J. A., AND SMOLKA, G. 1987. Order-sorted unification. Tech. Rep. CSLI-87-86, Center for the Study of Language and Information.

MILLER, D. A., AND NADATHUR, G. 1987. Some use of higher-order logic in computational linguistics. In *Proceedings of the 25rd Annual Meeting of the Association for Computational Linguistics.*

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, pp. 348–375.

MINSKY, M. 1975. A framework for representing knowledge. In *The Psychology of Computer Vision*, P. Winston, Ed. McGraw-Hill, New York.

MITCHELL, T. 1979. Version spaces: An approach to concept learning, Ph.D. dissertation, Stanford Univ., Stanford, Calif.

MOSHIER, D., AND ROUNDS, W. C. 1987. A logic for partially specified data structures. In *ACM Symposium on Principles of Programming Languages.* ACM, New York.

MUKAI, K. 1983. A unification algorithm for infinite trees. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

MUKAI, K. 1985a. Horn clause logic with parameterized types for situation semantics programming. Tech. Rep. TR-101, ICOT.

MUKAI, K. 1985b. Unification over complex indeterminates in Prolog. Tech. Rep. TR-113, ICOT.

MUKAI, K., AND YASUKAWA, H. 1985. Complex indeterminates in Prolog and its application to discourse models. *New Generation Comput. 3*, pp. 441–466.

NILSSON, N. J. 1980. *Principles of Artificial Intelligence.* Tioga, Palo Alto.

PATERSON, M. S., AND WEGMAN, M. N. 1976. Linear unification. In *Proceedings of the Symposium on the Theory of Computing.* ACM Special Interest Group for Automata and Computability Theory (SIGACT).

PATERSON, M. S., AND WEGMAN, M. N. 1978. Linear unification. *J. Comput. Syst. Sci. 16*, pp. 158–167.

PEREIRA, F. C. N. 1981. Extraposition grammars. *Comput. Linguist. 7.*

PEREIRA, F. C. N. 1985. A structure-sharing representation for unification-based grammar formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics.*

PEREIRA, F. C. N. 1987. Grammars and logics of partial information. In *Proceedings of the 4th International Conference on Logic Programming.* Springer-Verlag, New York.

PEREIRA, F. C. N., AND SHIEBER, S. 1984. The semantics of grammar formalisms seen as computer languages. In *Proceedings of the International Conference on Computational Linguistics.*

PEREIRA, F. C. N., AND WARREN, D. H. D. 1980. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artif. Intell. 13*, pp. 231–278.

PIETRZYKOWSKI, T. 1973. A complete mechanization of second-order type theory. *J. ACM, 20*, pp. 333–365. (Also Univ. of Waterloo, Research Rep., CSRR 2038, 1971.)

PIETRZYKOWSKI, T., AND JENSEN, D. 1972. A complete mechanization of ω-order logic. Research Rep. CSRR 2060, Univ. of Waterloo, Waterloo, Ontario, Canada.

PIETRZYKOWSKI, T., AND JENSEN, D. 1973. Mechanizing ω-order type theory through unification. Tech. Rep. CS-73-16, Univ. of Waterloo, Waterloo, Ontario, Canada.

PIETRZYKOWSKI, T., AND MATWIN, S. 1982. Exponential improvement of efficient backtracking: A strategy for plan based deduction. In *Proceedings of the 6th International Conference on Automated Deduction.* Springer-Verlag, New York.

PLAISTED, D. 1984. The occur-check problem in Prolog. *New Generation Comput. 2*, pp. 309–322.

PLOTKIN, G. 1970. A note on inductive generalization. *Mach. Intell. 5.*

PLOTKIN. G. 1972. Building-in equational theories. *Mach. Intell. 7.*

POLLARD, C. 1985. Phrase structure grammar without metarules. In *Proceedings of the 4th West Coast Conference on Formal Linguistics.*

REYLE, U., AND FREY, W. 1983. A PROLOG implementation of lexical functional grammar. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

REYNOLDS, J. C. 1970. Transformational systems and the algebraic structure of atomic formulas. *Mach. Intell. 5.*

RISTAD, E. S. 1987. Revised generalized phrase structure grammar. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics.*

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12*, pp. 23–41.

ROBINSON, J. A. 1968. New directions in mechanical theorem proving. In *Proceedings of the International Federation of Information Processing Congress.*

ROBINSON, J. A. 1969. Mechanizing higher-order logic. *Mach. Intell. 4.*

ROBINSON, J. A. 1970. A note on mechanizing higher order logic. *Mach. Intell. 5.*

ROBINSON, J. A. 1971. Computational logic: The unification computation. *Mach. Intell. 6.*

ROBINSON, P. 1985. The SUM: An AI coprocessor. *Byte 10.*

ROUNDS, W. C., AND KASPER, R. 1986. A complete logical calculus for record structures representing linguistic information. In *Proceedings of the 1st Symposium on Logic in Computer Science.* Cosponsored by the IEEE Computer Society, Technical Committee on Mathematical Foundations of Computing; ACM Special Interest Group for Automata and Computability Theory (SIGACT); Association for Symbolic Logic; European Association for Theoretical Computer Science.

ROUNDS, W. C., AND MANASTER-RAMER, A. 1987. A logical version of functional grammar. In *Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics.*

ROUSSEL, P. 1975. PROLOG, Manuel de reference et d'utilisation. Groupe Intelligence Artificielle, Université Aix-Marseille II.

SAG, I. A., AND POLLARD, C. 1987. Head-driven phrase structure grammar: An informal synopsis. Tech. Rep. CSLI-87-79, Center for the Study of Language and Information.

SCHMIDT-SCHAUSS, M. 1986. Unification under associativity and idempotence is of type nullary. *J. Autom. Reasoning 2,* pp. 277–281.

SELLS, P. 1985. *Lectures on Contemporary Syntactic Theories.* Univ. of Chicago, Chicago, Ill. Also, CSLI Lecture Notes Series.

SHAPIRO, E. 1983. A subset of concurrent Prolog and its interpreter. Tech. Rep. TR-003, ICOT.

SHIEBER, S. 1984. The design of a computer language for linguistic information. In *Proceedings of the International Conference on Computational Linguistics.* North-Holland, Amsterdam, New York.

SHIEBER, S. 1985. Using restriction to extend parsing algorithms for feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics.* Sponsored by International Joint Conference on Artificial Intelligence.

SHIEBER, S. 1986. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes Series, Center for the study of Language and Information, Stanford, California.

SIEKMANN, J. 1984. Universal unification. In *Proceedings of the 7th International Conference on Automated Deduction.* (Newer version to appear in the *J. Symbolic Comput.* special issue on unification, C. Kirchner, Ed., 1988.) Springer-Verlag, New York.

SIEKMANN, J. 1986. Unification theory. In *Proceedings of the 7th European Conference on Artificial Intelligence.* Sponsored by the European Coordinating Committee for Artificial Intelligence.

SMOLKA, G., AND AÏT-KACI, H. 1987. Inheritance hierarchies: Semantics and unification. Tech. Rep. AI-057-87, Microelectronics and Computer Technology Corporation (MCC), Austin, Texas. (To appear in the *J. Symbolic Comput.* special issue on unification, C. Kirchner, Ed., 1988.)

STABLER, E. P. 1983. Deterministic and bottom-up parsing in Prolog. In *Proceedings of the Conference of the American Association for Artificial Intelligence.*

STEELE, G. 1984. *Common LISP: The Language.* Digital Press, Bedford, Mass.

STICKEL, M. 1978. Mechanical theorem proving and artificial intelligence languages, Ph.D. dissertation, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

TOMITA, M. 1987. An efficient augmented-context-free parsing algorithm. *Comput. Linguist. 13,* pp. 31–46.

TOMITA, M. 1985a. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the International Joint Conference on Artificial Intelligence.*

TOMITA, M. 1985b. An efficient context-free parsing algorithm for natural languages and its applications, Ph.D. dissertation, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pa.

TOMITA, M., AND KNIGHT, K. 1988. Full unification and pseudo unification. Tech. Rep. CMU-CMT-87-MEMO. Center for Machine Translation, Carnegie-Mellon Univ., Pittsburgh, Pa.

TRUM, P., AND WINTERSTEIN, G. 1978. Description, implementation, and practical comparison of unification algorithms. Internal Rep. No. 6/78, Fachbereich Informatik, Universitat Kaiserlautern, W. Germany.

USZKOREIT, H. 1986. Categorial unification grammars. In *Proceedings of the International Conference on Computational Linguistics.* (Also CLSI Tech. Rep. CSLI-86-66.) North-Holland, Amsterdam, New York.

VAN EMDEN, M. H., AND KOWALSKI, R. A. 1976. The semantics of predicate logic as a programming language. *J. ACM 23,* pp. 733–742.

VENTURINI-ZILLI, M. 1975. Complexity of the unification algorithm for first-order expressions. Res. Rep. Consiglio Nazionale Delle Ricerche Istituto per le applicazioni del calcolo, Rome, Italy.

VITTER, J. S., AND SIMONS, R. A. 1984. Parallel algorithms for unification and other complete problems in P. In *ACM '84 Conference Proceedings* (San Francisco, Calif., Oct. 8–10). ACM, New York.

VITTER, J. S., AND SIMONS, R. A. 1986. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Trans. Comput. C-35.*

WALTHER, C. 1986. A classification of many-sorted unification problems. In *Proceedings of the 8th International Conference on Automated Deduction.* Springer-Verlag, New York.

WARREN, D. H. D., PEREIRA, F. C. N., AND PEREIRA, L. M. 1977. Prolog—The language and its implementation compared with LISP. In *Symposium on Artificial Intelligence and Programming Systems.* Co-sponsored by ACM Special Interest Group on Artificial Intelligence (SIGACI), ACM Special Interest Group on Programming Languages (SIGPL).

WINTERSTEIN, G. 1976. Unification in second order logic. Res. Rep. Fachbereich Informatik, Universitat Kaiserslautern, W. Germany.

WITTENBURG, K. 1986. Natural language parsing with combinatory categorial grammar in a graph-unification-based formalism. Ph.D. dissertation. Univ. of Texas, Austin, Texas.

WOODS, W. 1970. Transition network grammars for natural language analysis. *Commun. ACM 13,* pp. 591–606.

WROBLEWSKI, D. 1987. Nondestructive graph unification. In *Proceedings of the Conference on the American Association for Artificial Intelligence.*

YASUURA, H. 1984. On the parallel computational complexity of unification. In *Fifth Generation Computer Systems.* Sponsored by the Institute for New Generation Computer Technology (ICOT). (Also Yajima Labs, Tech. Rep. ER-83-01, 1983.)

YOKOI, T., MUKAI, K., MIYOSHI, H., TANAKA, Y., AND SUGIMURA, R. 1986. Research activities on natural language processing of the FGCS project. *ICOT J. 14,* pp. 1–8.

ZEEVAT, H., KLEIN, E., AND CALDER, J. 1987. Unification categorial grammar. In *Categorial Grammar, Unification Grammar, and Parsing,* N. Haddock, E. Klein, and G. Morrill, Eds. Centre for Cognitive Science, University of Edinburgh, Edinburgh, Scotland.

Received January 1988; final revision accepted July 1988.