LINEAR UNIFICATION

THESIS

Presented to the Graduate Council of the University of North Texas in Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

John W. Wilbanks, B.S., M.B.A
Denton, Texas
December, 1989
Wilbanks, John W., *Linear Unification*. Master of Science (Computer Science), December, 1989, 80pp., 2 figures, 3 tables, reference list, 37 titles.

Efficient unification is considered within the context of logic programming. Unification is explained in terms of equivalence classes made up of terms, where there is a constraint that no equivalence class may contain more than one function term. It is demonstrated that several well-known "efficient" but nonlinear unification algorithms continually maintain the said constraint as a consequence of their choice of data structure for representing equivalence classes. The linearity of the Paterson-Wegman unification algorithm is shown largely to be a consequence of its use of unbounded lists of pointers for representing equivalences between terms, which allows it to avoid the nonlinearity of "union-find".
I wish to thank C.-C. Yang for introducing me to the subject of unification and for providing patient support throughout my research.
## TABLE OF CONTENTS

- LIST OF FIGURES ........................................ vi
- LIST OF TABLES .................................................. vii
- LIST OF ALGORITHMS ........................................... viii

### Chapter

#### 1. UNIFICATION IN THE CONTEXT OF LOGIC PROGRAMMING 1

1.1. Introduction .................................................. 1
1.2. Logic Programming ............................................ 5
1.3. Prolog ....................................................... 7
1.4. Unification ................................................... 10
1.5. Unification at a Glance .................................... 14
1.6. A Nondeterministic Unification Algorithm ........ 16
1.7. Unification in Tabular Form .............................. 17

#### 2. THE COMPLEXITY OF UNIFICATION .................. 18

2.1. Introduction .................................................. 18
2.2. Preliminaries ................................................ 18
   2.2.1. Composition of Substitutions ...................... 18
   2.2.2. Properties of Substitutions ...................... 20
   2.2.3. The Most General Unifier (mgu) .................... 21
   2.2.4. The Form of mgus ..................................... 23
   2.2.5. The Disagreement Set ................................ 24
2.3. Robinson's Unification Algorithm ...................... 26
2.4. Description of the Algorithm ............................ 28
2.5. Failure to Unify ............................................ 31
2.6. Defining the Occur Check ................................ 32
2.7. The Computational Complexity of Robinson's
     Algorithm ..................................................... 33
2.8. The "Impossibility" of Linear Unification ............ 35
2.9. Simplifying the Study of Unification .................. 37
2.10. Finding Differences ....................................... 39
2.11. Chains of Bindings ....................................... 43
2.12. Clashing Bindings ......................................... 44
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. Eliminating Clashes</td>
<td>47</td>
</tr>
<tr>
<td>2.2. Worst-case Resolution of Clashes</td>
<td>48</td>
</tr>
</tbody>
</table>
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1. Landmark Unification Algorithms</td>
<td>5</td>
</tr>
<tr>
<td>1.2. Unification in Tabular Form</td>
<td>17</td>
</tr>
<tr>
<td>2.1. Cardinality of Disagreement Sets</td>
<td>26</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1. Nondeterministic Unification</td>
<td>16</td>
</tr>
<tr>
<td>2.1. Robinson's Unification Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>2.2. Robinson's Unification Algorithm, Made Explicit</td>
<td>28</td>
</tr>
<tr>
<td>2.3. udiff</td>
<td>41</td>
</tr>
<tr>
<td>2.4. CNF: Create New Fraction</td>
<td>46</td>
</tr>
<tr>
<td>3.1. graph</td>
<td>56</td>
</tr>
<tr>
<td>3.2. list-trivial</td>
<td>57</td>
</tr>
<tr>
<td>3.3. list-simple</td>
<td>58</td>
</tr>
<tr>
<td>3.4. unify</td>
<td>70</td>
</tr>
</tbody>
</table>
1.1. Introduction

Unification is the process of making two or more expressions syntactically identical by replacing their variables. The subject was first studied by Herbrand (1930), who in his thesis presented a nondeterministic algorithm to compute a unifier of two terms. However, his algorithm was designed to solve equations; he did not introduce the notion of unifiers and did not study their properties (Lassez, Maher, and Marriott 1987). In an influential paper on proof procedures, Prawitz (1960) introduced an idea which, with some modification, "leads directly to the unification algorithm" (Davis 1983), although unification as it is understood today was not the thrust of his work.

J. A. Robinson (1965) was the first to propose a unification algorithm within the context of what today would be termed logic programming; it is his work that supplies the modern notation for unification and provides the motivation for making the study of unification both useful and relevant (Knight 1989; Lassez, Maher, and Marriott
Robinson's algorithm was written in such a way that its correctness could be proved and its properties could be utilized in the development of his theory leading to the powerful inference rule of resolution. Unfortunately, as a computational device, Robinson's unification algorithm is grossly inefficient, having a worst-case complexity of $O(2^n)$.

Robinson later provided additional analysis of unification (Robinson 1971) and speculated that his new version might be much more efficient. However, Corbin and Bidoit (1983) demonstrated that the new method still has the same worst-case complexity.

Robinson also published a Lisp implementation of an improved algorithm in a subsequent book (Robinson 1979). This version appears in the appendix of his book and is not rigidly supported by theory, but it incorporates a number of programming optimizations that have become common in unification algorithms, such as recursively matching function arguments, representing a (most general) unifier as a graph, and avoiding the application of a new substitution to the original terms at each step of the algorithm.

L. D. Baxter developed a unification algorithm whose complexity is almost linear (Baxter 1973, 1976a, 1976b). In fact, one could claim that for practical purposes, it is linear. Its complexity is $O(nG(n))$, where $G(n)$ is the extremely slowly-growing functional inverse of Ackermann's
function. For values of $n$ up to $2^2 \ldots 2^2$ (65536 occurrences of 2), $G(n)$ is no greater than 3! Thus, for all conceivable uses of unification in practice, Baxter's algorithm can be considered to have a complexity no worse than $O(3n)$.

Huet (1976) presented an algorithm similar in concept to Baxter's. It is marked by its simplicity and clarity. (Huet's paper is in French, but an accessible sketch is available in Knight (1989)).

Martelli and Montanari developed a linear unification algorithm described in an internal technical report of the University of Pisa (Martelli and Montanari 1976), with later improvements (Martelli and Montanari 1977a, 1977b). A subsequent algorithm published in more readily-available sources (Martelli and Montanari 1982) is "efficient" ($O(n \log n)$) rather than linear, but is claimed to be superior in practice if not in theory.

In 1976, Paterson and Wegman published an algorithm by which all others are now measured (Paterson and Wegman 1976), as the algorithm is shown to be linear. It was followed by a substantially revised version (Paterson and Wegman 1978), with errors having been corrected by de Champeaux (1986). Despite its linearity, the algorithm has met a strange reception. It uses directed acyclic graphs (dags) as its main data structure, implying considerable complexity in the implementation of the algorithm via
traditional programming languages. Some authors, e.g. Escalada-Imaz and Ghallab (1988), maintain that the computational cost of building and using the requisite data structures is great enough to negate the algorithm's theoretical efficiency when used on typical logic programming problems, which tend to have a large number of simple unifications. The same claim is made by Martelli and Montanari (1982). De Champeaux (1986) has proposed pre-processing and post-processing techniques to convert problems into and out of the proper dag form, but this proposal tends merely to reinforce the notion that the algorithm employs intractable data structures.

A number of works in the 1980s have emphasized algorithms that are "practically efficient", i.e., algorithms that in typical "real-life" applications outperform the theoretically more efficient algorithms. The algorithm of Escalada-Imaz and Ghallab (1988) is one example, and the non-linear, but "efficient" algorithm of Martelli and Montanari (1982) is another.

In this work I am concerned with the efficiency of unification algorithms that are generally intended for use in logic programming. Since unification is a very heavily used component of logic programming systems, its efficiency is critical if logic programming is to be practical. A linear algorithm is certainly welcome. I explore an intellectual path that begins with Robinson's original
unification algorithm (which essentially defines the unification problem), proceeds through Baxter's nearly-linear algorithm, and ends with Paterson and Wegman's linear algorithm. I elucidate the conceptual underpinnings of the Paterson-Wegman algorithm, and the price to be paid to achieve its (theoretical) linearity.

Table 1.1 summarizes the landmark algorithms.

<table>
<thead>
<tr>
<th>Author(s)</th>
<th>Worst-case Complexity</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robinson</td>
<td>$O(2^n)$</td>
<td>&quot;Defines&quot; unification</td>
</tr>
<tr>
<td>Baxter</td>
<td>$O(nG(n))$</td>
<td>Equivalence classes</td>
</tr>
<tr>
<td>Huet</td>
<td>$O(nG(n))$</td>
<td>Similar to Baxter</td>
</tr>
<tr>
<td>Martelli&amp;Montanari</td>
<td>$O(n\log n)$</td>
<td>Equivalence relations</td>
</tr>
<tr>
<td>Paterson&amp;Wegman</td>
<td>$O(n)$</td>
<td>Complex data structs</td>
</tr>
</tbody>
</table>

1.2. Logic Programming

In logic programming, problems are solved by stating them in terms of symbolic logic and then finding proofs. This has obvious application to theorem-proving in mathematics and related disciplines. Of greater significance, however, is that logic programming can be used to solve problems that are less obviously related to theorem proving. For example, if one wants a robot arm to move a block from the bottom of a stack to the top without making the stack fall, a proof that the blocks can be rearranged as
desired constitutes a plan for doing so. As another example, given a list of humans and other beings, along with the rule that all humans are mortal, one can try to prove that nothing is mortal, fail by finding counterexamples, and thereby obtain a list of mortals.

In fact, it is not merely the case that some computations can be accomplished as proofs; in recent years it has become clear that all computations can be done so. The significance is that once a problem has been described in terms of symbolic logic, the description can be interpreted as a computation. In other words, logic programming promises an environment in which stating a problem is tantamount to solving it (assuming, of course, that a solution exists).

It is not practical for computers to use "traditional" symbolic logic to prove theorems; the rules of inference are so varied that it takes too much computation to find a proof. However, in a seminal paper J. A. Robinson (1965) introduced a new inference mechanism consisting of a single rule of inference, resolution. When logical statements are restricted to a form known as "Horn clauses", proofs can be obtained by repeated application of this simple inference rule. This new method of proof is too coarse to be used in the everyday work of logicians, mathematicians, and philosophers, but its primitive nature makes it suitable for mechanized proofs done by computer.
Resolution by itself is an inadequate logical tool; it must be augmented by unification to be useful. Unaugmented resolution can be used in any problem that can be solved with a truth table -- not a particularly rich set of problems. To become useful, it must incorporate unification, a process which, among other things, allows general statements to be matched with specific factual instances. For example, if we have a general logical statement about "all humans", and we know that John is a human, then by unification we are allowed to apply the general statement to John. When resolution is augmented by unification, it becomes possible to solve problems stated in terms of 'all' or 'some', i.e., the predicate calculus.

1.3. Prolog

In the early 1970s, Prolog (an acronym for Programming in Logic) was invented by Alain Colmerauer and his associates in France with the objective of integrating the resolution principle into a programming language (Colmerauer 1985; Clocksin and Mellish 1987).

There are now several forms of Prolog, the de facto standard being Edinburgh Prolog (Clocksin and Mellish 1987). In Prolog one forms logical statements by incorporating constants, variables, and predicates into logical expressions known as "clauses". In the Edinburgh syntax a constant is denoted by an identifier beginning with a lower-case letter. A variable symbol begins with an
upper-case letter. A predicate is a true-false function describing some characteristic of one or more constants or variables. For example, 'mother(john,mary)' is a predicate that we might take to mean "The mother of John is Mary." The predicate is true if Mary is, in fact, John's mother, and false otherwise. Similarly, 'mother(john,X)' in some sense refers to John's mother, who is not specifically named but who is identified by the variable X.

A typical Prolog program has two kinds of clauses: rules and facts. A rule has a single predicate in its head and one or more predicates in its body. A rule's head is separated from the body by the symbol ':-', usually read as 'if'. A fact consists of a single predicate. An example of a simple Prolog program is:

```
mortal(X) :- human(X).
human(socrates).
human(plato).
god(zeus).
```

This program states the rule that X is mortal if X is human, or expressed more traditionally, "all humans are mortal."
The program also includes the facts that Socrates and Plato are human, while Zeus is a god.

One can ask various questions of this program by presenting queries, which are represented by "?-" followed by one or more predicates, called goals. Prolog will answer by trying to match the each goal to the facts or to the heads of rules. Whenever the head of a rule is matched,
Prolog will make a goal out of each predicate in the rule's body and will try to "satisfy" each one.

For example, to determine if Socrates is human, one would present the query "?- human(socrates)." Prolog searches the rules and facts, finds an exact match with the fact "human(socrates)" , and replies "yes" to the query.

One can also ask who is human by presenting the query "?- human(X)." Prolog finds no exact matches, but it uses unification to match patterns by determining how to change variables to make two predicates match. That is, "human(X)" matches "human(socrates)" if the variable 'X' is replaced by 'socrates'. In a similar fashion, "human(X)" and "human(plato)" can be unified by replacing 'X' with 'plato'. Thus, Prolog can answer the query by reporting 'X=socrates' and 'X=plato'.

Finally, one might ask if Socrates is mortal by presenting the query "?- mortal(socrates)." Prolog unifies the goal with the head of the example's only rule by substituting 'socrates' for 'X'. Prolog then has to satisfy the body of the matched rule, which now consists of the predicate "human(socrates)". Finding an exact match to this, Prolog replies 'yes' to the query -- Socrates is mortal.

Unlike the foregoing simple example, typical Prolog programs contain large numbers of rules and facts. As Prolog attempts to answer a query, it must not only try to
unify many rules and facts, but the search for an answer may involve many instances of "blind alleys" and subsequent backtracking, meaning that the number of unifications is increased even further. In many Prolog programs, a simple query can result in hundreds or thousands of unifications. It is for this reason that unification must be efficient if Prolog (or any form of logic programming) is to be a practical programming tool. It is why many (or most) implementations of Prolog take the much-debated shortcut of leaving the expensive "occur-check" out of unification (Beer 1988; Plaisted 1984).

1.4. Unification

Stated informally, unification is the process of finding a set of variable replacements that will make two or more expressions syntactically identical. For example,

To unify ____________________________ Do this ___________________________
human(X), human(socrates) replace X by socrates.
human(X), human(Y) replace X by Y, OR replace X by Z, Y by Z, OR replace X by abe, Y by abe.

human(abe), human(amy) (not unifiable).
human(X), animal(X) (not unifiable).

Variable replacements are variously written in the forms "abe/X", "X/abe", "X<--abe", etc., depending upon the author. This paper uses the form, "abe/X", which has the informal meaning that the variable "X" is to be replaced by
Within the context of logic programming, unification is performed on expressions, where expressions are built by combining constants, variables, functions, predicates, terms, and (sub)expressions in carefully prescribed ways (see, for example, Lloyd (1987), Chang and Lee (1973), or Loveland (1978)). To simplify the discussion of unification we can restrict attention to just a subset of these building blocks. We can ignore constants since a constant can be viewed as a function with no arguments. Since a predicate is a function that returns the values "true" and "false", we can treat predicates as a special case of functions. Finally, we can consider expressions to be an elaboration on terms. We thus proceed with the following recursive definition of terms and functions.

**Definition 1.1.** A **term** is a variable or a function. A **function** is a function symbol and 0 or more arguments, where each argument is a term.

Given some terms to unify, we will have to replace some of the variables by other functions or variables. We need a way to designate a variable's replacement, then we need to group variable replacements in such a way that no variable has more than one replacement.

**Definition 1.2.** A **binding** (or replacement) is a term \( t \) and a variable \( V \), written \( t/V \).

Since a binding is written in a way that resembles a
fraction, it will be convenient to call $t$ the numerator and $V$ the denominator of binding $t/V$.

**Definition 1.3.** A substitution is a set of bindings $(t_1/V_1,...,t_n/V_n)$ such that the variables $V_1,...,V_n$ are distinct and each $t_i$ is a term distinct from its corresponding $V_i$.

In other words, a substitution never has two bindings for the same variable (since the $V_i$ are distinct), and never includes "useless" bindings such as $X/X$ (since each $t_i$ is distinct from its corresponding $V_i$).

A binding has both declarative and operational interpretations. In the declarative sense, a binding is merely an ordered pair $<t,V>$. However, from an operational perspective, a binding $t/V$ "directs" us to replace the variable $V$ in some expression by the term $t$. It is in this sense that we sometimes call a binding a replacement.

We will be using substitutions to replace variables that occur in terms.

**Definition 1.4.** Let $T$ be a term and $	heta = (t_1/V_1,...,t_n/V_n)$ a substitution. The instance of $T$ by $\theta$, written $T\theta$, is the term obtained by simultaneously replacing each occurrence of $V_i$ in $T$ by the term $t_i$ ($i=1,...,n$). When writing $T\theta$, we say that we apply $\theta$ to $T$.

For example, let $T = f(X,Y,g(Z))$, and $\theta = (W/X,U/Z)$. Then:
$$T \Theta = f(X,Y,g(Z)){W/X,U/Z}$$
$$= f(W,Y,g(U)).$$

The definition's notion of simultaneous replacement is important. For example, given $T = f(X,Y)$ and $\Theta = (X/Y,W/X)$, then
$$T \Theta = f(X,Y){X/Y,W/X}$$
$$= f(W,X),$$

obtained by replacing the $X$ and $Y$ in $f(X,Y)$ at the same time. It would be incorrect to perform the replacements sequentially, from left to right, for then one would first use $X/Y$ to change $f(X,Y)$ to $f(X,X)$, then would use $W/X$ to change $f(X,X)$ to $f(W,W)$, which is incorrect.

The point of unification (which I will repeat many times) is to make two or more terms identical by making variable replacements. We need a way to indicate that a substitution applies to a set of terms, not just one term.

**Notation.** If $T = \{t_1, \ldots, t_n\}$ is a set of terms and $\Theta$ is a substitution, then $T \Theta$ denotes the set $\{t_1 \Theta, \ldots, t_n \Theta\}$.

For example, let $T = \{f(X), f(Y), g(X)\}$ and $\Theta = (W/X, Z/Y)$. Then
$$T \Theta = \{f(X), f(Y), g(X)\}{W/X, Z/Y}$$
$$= \{f(X){W/X, Z/Y}, f(Y){W/X, Z/Y}, g(X){W/X, Z/Y}\}$$
$$= \{f(W), f(Z), g(W)\}.$$

We are ready to formalize the notion of unification. The idea is to simultaneously apply a substitution to the terms being unified. If the substitution is well-chosen, the
substituted terms will be identical, i.e., the set of terms will "collapse" into a singleton.

Definition 1.5. Let $T$ be a set of terms. A substitution $\theta$ is a unifier for $T$ if $T\theta$ is a singleton.

Example 1.1. Let $T = \{f(X,g(Y)), f(Z,W), f(X,W)\}$. Then $\theta = \{X/Z, g(Y)/W\}$ is a unifier for $T$ because

$T\theta = \{f(X,g(Y)), f(Z,W), f(X,W)\}\{X/Z, g(Y)/W\}$

$= \{f(X,g(Y))\}$.

1.5. Unification at a Glance

One can develop the ability to unify terms almost "at a glance", or at worst "with a scribble". The technique is based upon a nondeterministic unification algorithm originally expressed by Herbrand (1930) and elaborated by Martelli and Montanari (1982). The refinement given here relies upon the informal notion of "fractions" and can be attributed largely to Li, Yang, and Ng (1988) and Wilbanks (1988).

The idea is to write the original two terms as a "fraction" and to gradually refine this into a set of bindings that constitute a unifier. Without being too precise, the rules are:

1. Always replace a function over a function by a new set of fractions formed by pairing the corresponding function arguments. For example, replace

\[
\frac{f(X,Y,g(Z))}{f(Y,W, U)}
\]

by
\[
\frac{X}{Y} g(Z) \\
\frac{Y}{W} U
\]

(If the functions differ in name or arity, fail).

2. Swap a function into the numerator if it will leave a variable in the denominator. For example, change
\[
\frac{X}{f(Y)}
\]
to
\[
\frac{f(Y)}{X}.
\]

3. Delete any fraction where the numerator = the denominator.

4. If two fractions have the same variable in the denominator, replace the denominator of one by the numerator of the other. For example, change
\[
\frac{f(X)}{Z} \frac{f(Y)}{Z}
\]
to
\[
\frac{f(X)}{Z} \frac{f(Y)}{f(X)}.
\]

5. Whenever a fraction has just a variable in the denominator, and that variable occurs in the numerator of other fractions, replace it in those other fractions. For example, change
\[
\frac{f(X)}{Z} \frac{f(Z)}{W}
\]
to
\[
\frac{f(X)}{Z} \frac{f(f(X))}{W}.
\]

However, if the variable occurs in its own numerator, fail.
Example 1.2. Let the terms to be unified be 
\( h(f(X), W, U, Z) \) and \( h(Y, f(Y), U, W) \).

Create a fraction:

\[
\frac{h(f(X), W, U, Z)}{h(Y, f(Y), U, W)}
\]

Break out the fcn arguments:

\[
\frac{f(X)}{Y}, \frac{W}{f(Y), U, Z}
\]

Eliminate useless bindings:

\[
\frac{f(X)}{Y}, \frac{W}{f(Y), U, W}
\]

Swap fcn f(Y) to the top:

\[
\frac{f(X)}{Y}, \frac{f(Y)}{W, Z}
\]

Eliminate duplicate W's:

\[
\frac{f(X)}{Y}, \frac{f(Y)}{W, W}
\]

Swap fcn f(Y) to the top:

\[
\frac{f(X)}{Y}, \frac{f(Y)}{W, Z}
\]

Replace top Y's by f(X):

\[
\frac{f(X)}{Y}, \frac{f(f(X))}{f(f(X))}
\]

The resulting set of fractions (bindings) constitute a unifier for the original terms.

1.6. A Nondeterministic Unification Algorithm

The intuitive and imprecise method of "unification at a glance" is made formal by the following algorithm, which is, in essence, Martelli and Montanari's (1982) nondeterministic algorithm, modified to employ the notion of "fractions".

Algorithm 1.1. Nondeterministic Unification.

Let a fraction be a term s and a term t, written s/t. Let the two terms to unify be initially written as a fraction.

Apply any of the following transformations until none can be applied:

1. Replace a fraction of the form v/t by t/v.
2. Delete a fraction of the form v/v.
3. Replace a fraction of the form 
\( f(t_1, \ldots, t_n)/f(s_1, \ldots, s_n) \) by \( t_1/s_1, \ldots, t_n/s_n \).
4. If there is a fraction $t/v$ such that 1) $v$ does not occur in $t$ and 2) $v$ occurs as the numerator or denominator of some other fraction, replace all such occurrences of $v$ by $t$ (except in $t/v$, itself).

5. If there is a fraction of the form $f(...)/g(...)$, $f\neq g$, fail.

6. If there is a fraction of the form $f(t_1,...,t_n)/g(s_1,...,s_m)$, $n\neq m$, fail.

7. If there is a fraction $t/v$ such that $v$ occurs in $t$, fail.

1.7. Unification in Tabular Form

Cohen (1985) expressed nondeterministic unification in the form of a table. When altered to fit the conventions of the present paper, the tabular representation of unification is given by Table 1.2.

**Table 1.2. Unification in Tabular Form.**

<table>
<thead>
<tr>
<th>Term \ Term</th>
<th>(variable)</th>
<th>(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 \ 1</td>
<td>V1</td>
<td>F1</td>
</tr>
<tr>
<td>(variable)</td>
<td>succeed with</td>
<td>succeed with</td>
</tr>
<tr>
<td>V2</td>
<td>V1/V2</td>
<td>F1/V2</td>
</tr>
<tr>
<td>(function)</td>
<td>succeed with</td>
<td>succeed if</td>
</tr>
<tr>
<td>F2</td>
<td>F2/V1</td>
<td>1) $F_1$ and $F_2$ have same name and arity, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) the matching of corresponding fcn arguments succeeds</td>
</tr>
</tbody>
</table>
2.1. Introduction

Robinson's algorithm (Robinson 1965) is the starting point for the study of the complexity of unification. This chapter introduces Robinson's algorithm, analyzes it, and draws from it those aspects of unification that are relevant to any unification algorithm.

2.2. Preliminaries

Robinson's algorithm requires the use of three notions not yet discussed. The first is the composition of substitutions, the second is the idea of the most general unifier, and the last is disagreement sets. The first two notions are not unique to Robinson's algorithm; they are employed in all unification algorithms. The last notion is an instance of a more generalized idea which manifests itself in all the other algorithms in various ways.

2.2.1. Composition of Substitutions

A substitution is applied to a set of terms. For example, let

\[ T = \{h(f(X), f(Y)), h(Y, Z)\} \]

be a set of terms and

\[ \theta_1 = (W/X) \]

a substitution. Then,
Tel = {h(f(W), f(Y)), h(Y, Z)} is the instance of T by θ₁. But Tel is itself a set of terms, which means we can apply another substitution to it. That is, given

θ₂ = (V/Y), then we can compute

(TTel)θ₂ = {h(f(W), f(V)), h(V, Z)}. In general, given substitutions θ₁, . . . , θₙ and a set of terms T, we can compute

(...(((Tel)θ₂)θ₃)...)θₙ. We want to define a new operation that allows us to compute the same result by "grouping" the substitutions, as by T((...((θ₁θ₂)θ₃)...)θₙ). More succinctly, we would like to ascribe to T(θ₁...θₙ) a meaning that corresponds to the left-to-right application of the substitutions. So we need to define an operation where a substitution is applied to another substitution instead of to a set of terms. The composition of substitutions is defined so as to produce the desired characteristics (Robinson 1965; Lloyd 1987):

**Definition 2.1.** Let θ₁ = {s₁/u₁, . . . , sₘ/uₘ} and θ₂ = {t₁/v₁, . . . , tₙ/vₙ} be substitutions. The composition θ₁θ₂ of θ₁ and θ₂ is the substitution obtained by performing the following steps in sequence:

1) Apply θ₂ to the numerator of each binding in θ₁ to create θₐ.

2) Delete from θₐ any binding where the numerator is identical to the denominator.

3) Add to θₐ the bindings of θ₂ which have a denominator not already represented in θₐ.
2.2.2. Properties of Substitutions

Definition 2.2. The identity substitution is the empty set and is denoted by the symbol \( \emptyset \). \( t\emptyset = t \) for all terms \( t \).

Substitutions have the following fundamental properties (Lloyd 1987). Let \( \theta_1, \theta_2, \) and \( \theta_3 \) be substitutions, then

1) \( \theta_1 \emptyset = \emptyset \theta_1 = \theta_1 \).
2) \( (t\theta_1)\theta_2 = t(\theta_1\theta_2) \), for all terms \( t \).
3) \( (\theta_1\theta_2)\theta_3 = \theta_1(\theta_2\theta_3) \).

The second property above is a direct result of the definition of the composition of substitutions. The third property (associativity) allows us to omit parentheses when writing a composition \( \theta_1\theta_2...\theta_n \) of substitutions.

The properties of substitutions are useful in unification algorithms since one typically constructs a unifier incrementally, composing a unifier-in-progress and a single-element incremental substitution at each step.

In general, the bindings in a substitution may have either variables or functions in their numerators. A substitution is variable-pure if each of its bindings has only a variable in the numerator.

It is intuitively clear that some sets of terms are equivalent to others, with only variable names having been changed. For example \( \{f(X),Y\} \) and \( \{f(Z),W\} \) are the same if \( X \) is renamed to \( Z \) and \( Y \) to \( W \). A substitution whose purpose is merely to rename variables is a renaming substitution, and has the following formal definition:
Definition 2.3. A **renaming substitution** for a term t (or a set of terms T) is a variable-pure substitution such that

1) each denominator is a variable occurring in t (or T),
2) the numerators are distinct, and
3) no numerator is equal to any unreplaced variable in t (or T).

2.2.3. The Most General Unifier (mgu)

Say that we have two very simple terms to unify: X and Y. The aim of unification is to find a substitution that makes the two terms syntactically equivalent. Many substitutions unify X and Y. Consider just a few:

1) \{X/Y\} Replace one var by the other
2) \{Y/X\} Replace one var by the other
3) \{Z/X, Z/Y\} Replace both vars by new var
4) \{Z/X, Z/Y, Z/W\} Include a superfluous binding
5) \{Z/X, Z/Y, U/V\} Include a superfluous binding
6) \{f(U)/X, f(U)/Y\} Replace both vars by new fcn
7) \{f(U,V)/X, f(U,V)/Y\} Replace both vars by new fcn
8) \{f(U,g(W))/X, f(U,g(W))/Y\} " " " " " " "

If a set of terms is unifiable, it has an infinite number of unifiers (since given a unifier, we can forever add "spurious" bindings of variables that do not occur in the terms being unified). Fortunately, among the unifiers is at least one "simplest" one, with all the other unifiers
being variants of it. There may actually be multiple "simplest" unifiers, but they are all variants of one another, where all their variables have been renamed. Such a "simplest" unifier is called a most general unifier (mgu) and is defined formally as follows.

**Definition 2.4.** Let $T$ be a finite set of terms and let $\Theta$ be a unifier for $T$. $\Theta$ is a **most general unifier** (mgu) for $T$ if, for each unifier $\Theta_1$ of $S$, there exists a substitution $\Theta_2$ such that $\Theta_2 \Theta_1 = \Theta_1$.

**Example 2.1.** Continuing the example from the beginning of this subsection in which $T = \{X,Y\}$, the mgu for $T$ is $\{X/Y\}$. Each of the example unifiers is a variant of $\{X/Y\}$:

1. $\{X/Y\} \emptyset = \{X/Y\}$
2. $\{X/Y\} \{Y/X\} = \{Y/X\}$
3. $\{X/Y\} \{Z/X\} = \{Z/X, Z/Y\}$
4. $\{X/Y\} \{Z/X, Z/W\} = \{Z/X, Z/Y, Z/W\}$
5. $\{X/Y\} \{Z/X, U/V\} = \{Z/X, Z/Y, U/V\}$
6. $\{X/Y\} \{f(U)/X\} = \{f(U)/X, f(U)/Y\}$
7. $\{X/Y\} \{f(U,V)/X\} = \{f(U,V)/X, f(U,V)/Y\}$
8. $\{X/Y\} \{f(U,g(W))/X\} = \{f(U,g(W))/X, f(U,g(W))/Y\}$

There are several definitions of mgu in the literature; the one used here is the generally accepted one. Robinson was the first to define an mgu (Robinson 1965). His definition was purely operational: he defined an mgu to be the output of his unification algorithm. In his later book
(Robinson 1979), he redefined $\theta$ to be an mgu iff for every unifier $\$, $\$ = $\theta$. Definition 2.4, above, is more general than Robinson's. It was provided by Chang and Lee (1973) and was adopted by Lloyd (1987). For additional discussion, refer to Lassez, Maher, and Marriott (1987).

**2.2.4. The Form of mgus**

In general it is difficult to determine by its form whether a unifier is an mgu or not. For example, $T_1 = \{f(X,Y), f(Z,W)\}$ has the unifier $\theta_1 = (X/Z, Y/W)$, which is an mgu, and $\theta_2 = (Z/X, Z/Y, Z/W)$, which is not. We will pay particular attention to $\theta_2$. It is a unifier since $\{f(X,Y), f(Z,W)\} \theta_2 = \{f(Z,Z)\}$ is a singleton.

It is not an mgu; if it were, there would be some substitution $\$ such that $\theta_2 \$ = $\theta_1$. But this is impossible since all the bindings in $\theta_2$ have $Z$ in the numerator, and there is no substitution that when composed with $\theta_2$ will yield both $X$ and $Y$ in the numerator of a pair of bindings as required by $\theta_1$.

We might speculate that some aspect of $\theta_2$'s form reveals that it is not an mgu. But this is not so, since $\theta_2$ is the mgu of another set of terms, $\{g(Z,Z,Z), g(X,Y,W)\}$.

Intuitively, a unifier is not an mgu if it has "unnecessary" bindings. For example, the mgu of $\{X,Y\}$ is $\{X/Y\}$. If we augment $\{X/Y\}$ with an "unnecessary" binding to get $\{X/Y, U/V\}$, the new substitution is still a unifier but is not an mgu. So we might conjecture that a unifier for $T$
is not an mgu if it includes variables not present in $T$. But this is inadequate since it is difficult to account for the effects of variable renaming.

It has been shown by Lassez, Maher, and Marriott (1987) in their Proposition 15 that a substitution $\theta$ is an mgu of some set of terms iff for every variable $v$ that occurs in both a numerator and denominator in $\theta$, there is some binding in $\theta$ such that $v$ is the numerator. By this criterion, $(X/Y, Y/Z)$ is an mgu since 1) $Y$ occurs in the denominator of $X/Y$ and the numerator of $Y/Z$, and 2) $Y$ is the numerator of binding $Y/Z$. In contrast, $(X/Y, f(Y)/Z)$ is not an mgu since $Y$ occurs in a denominator and a numerator, but $Y$ is not the numerator of any binding (although it occurs in the numerator of $f(Y)/Z$).

2.2.5. The Disagreement Set

As with any unification algorithm, Robinson's must identify and isolate the differences among terms to be unified in order to resolve those differences. Numerous ways exist for isolating differences among terms. Robinson chose to treat terms as strings and to isolate differences by comparing them. Thus the notion of a disagreement set:

**Definition 2.5.** Let $T$ be a finite set of terms. Locate the leftmost symbol position at which not all terms in $T$ have the same symbol. Extract from each term in $T$ the (sub)term beginning at that symbol position. The set of such (sub)terms is the disagreement set.
Example 2.2.

The disagreement set of

\[ T = \{ h(X,f(Y)), h(X,f(Z)), h(X,U) \} \]

is

\[ \{ f(Y), f(Z), U \}. \]

One should not misunderstand the disagreement set, thinking that it is a collection of all the differences among the terms being unified; it is the set of left-most differences. If \( n \) terms are being unified, the disagreement set is obtained by writing the terms in a vertical list, then moving left to right to find the first spot where the \( n \) terms differ. There may be other differences "to the right" of the leftmost differences, but they are not included in the disagreement set.

It should be clear, then, that the disagreement set for a set of \( n \) terms can have at most \( n \) members. The disagreement set has \( n \) members exactly when the \( n \) left-most corresponding terms are all different. For example, for

\[ T = \{ t_1, t_2, t_3 \} = \{ f(U,V,Y), f(U,W,Z), f(U,X,Y) \}, \]

the disagreement set is \( \{ V, W, X \} \); it has the same cardinality as \( T \).

If the disagreement set contains only one term, then there has been an error, namely an arity mismatch. Consider, for example,

\[ T = \{ t_1, t_2 \} = \{ f(U), f(U,V) \}. \]
The leftmost term at which \( f(U) \) and \( f(U,V) \) differ is \( V \). But since there is no subterm in \( f(U) \) that corresponds to \( V \), the disagreement set is \( \{V\} \), a set with just one member.

Of course, if the disagreement set is empty, it means that the terms are identical.

The disagreement set of \( \emptyset \) or a singleton is empty.

The following table summarizes what can be inferred from the cardinality of the disagreement set:

Table 2.1. Cardinality of Disagreement Sets.

<table>
<thead>
<tr>
<th>cardinality</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>all the terms are identical (unified)</td>
</tr>
<tr>
<td>1</td>
<td>arity mismatch among functions</td>
</tr>
<tr>
<td>2..(n-1)</td>
<td>some corresponding subterms are the same or there is an arity mismatch</td>
</tr>
<tr>
<td>n</td>
<td>all corresponding subterms are different</td>
</tr>
</tbody>
</table>

2.3. Robinson's Unification Algorithm

When Robinson presented his unification algorithm, the intent was to prove properties of the resolution principle, rather than to provide an efficient algorithm.

The algorithm is simple in form, consisting of just three steps. Here is the algorithm as presented by Lloyd (1987). I have added indentation to make the algorithm's structure more apparent.
Algorithm 2.1. Robinson's Unification Algorithm.

Let $T$ be a set of terms.
1. Put $k=0$ and $S_0 = \emptyset$.
2. If $T^k$ is a singleton then
   stop -- $S^k$ is an mgu of $T$.
   Otherwise
   find the disagreement set $D^k$ of $T^k$.
3. If there exist $v$ and $t$ in $D^k$ such that $v$ is a variable
   that does not occur in $t$, then
   put $S_{k+1} = S_k(t/v)$,
   increment $k$, and
   go to 2.
   Otherwise
   stop -- $T$ is not unifiable.

In order better to understand Robinson's algorithm, I prefer the following equivalent presentation which 1) makes important operations more explicit, 2) uses a C-like programming language format, and 3) avoids the use of subscripts by using programming conventions, instead.
Algorithm 2.2. Robinson's Algorithm, Made Explicit.

Given: A set $T = \{t_1, \ldots, t_n\}$ of terms to be unified.
Returns: $\$ = an mgu of $T$, if successful.

1. $\$ = \emptyset$; /* Identity subst'n */
2. while (1) {
3. $Q = T\$; /* Apply $\$ to $T$ */
4. $D = \text{disagreement set of } Q$; /* Isolate diffs */
5. if (D is empty) /* No differences? */
   6. succeed; /* $\$ is an mgu of $T$ */
7. if (D contains only 1 term)
   8. fail; /* Arity mismatch */
9. if (D contains no variables) /* Only functions? */
10. fail; /* They don't match */
11. $v = \text{some variable in } D$; /* Pick a variable */
12. $t = \text{some term in } D (\neq v)$; /* Pick a term */
13. if ($v$ occurs in $t$) /* "occur check" */
14. fail; /* Self-referential */
15. $\$ = \$(t/v)$; /* Augment unifier */
16. }

2.4. Description of the Algorithm

The analysis that follows is based on the explicit version of Robinson's algorithm, Algorithm 2.2.

The algorithm starts with an empty "unifier-in-progress", $\$. At each iteration of the loop, a
single disagreement is isolated. A binding is constructed from that disagreement, and $\$ is augmented (by composition). The new $\$ is applied to $T$ and the next disagreement is isolated. The process continues until there are no more disagreements or until it is determined that unification is impossible.

As an aid to understanding, it is important to remember that this algorithm is quite general in that it can unify any (finite) number of terms at once, not just pairs of terms. On any iteration, the disagreement set may have more than two members, with each member being either a variable or a function. Consequently the algorithm exhibits an element of nondeterminism at lines 11 and 12, where two terms are to be selected from the n>=2 terms in the current disagreement set. As will be demonstrated later, it is actually sufficient to limit our attention to the more special case of algorithms that unify exactly two terms at a time, with the beneficial side-effect that the nondeterminism is eliminated and some of the operations are made more precise.

The "unifier-in-progress" is the substitution $\$. It is initially set to the empty set (i.e., the identity substitution) at line 1. Bindings are added to $\$ one at a time, at line 15. Each time a binding is added, it eliminates one more difference among the terms being unified. The binding is added to $\$ by using a degenerate
form of the composition of substitutions: the binding is made the only member of a new substitution, then $* is composed with that new substitution.

Each time the "unifier-in-progress" is altered, it is re-applied to the original set of terms being unified (line 3). This is a source of great discomfort to programmers, for it appears to be, and in fact is, grossly and unnecessarily inefficient. Say that $T$ consists of several very long terms. On each iteration, $*$ has the property of being able to make the leftmost portions of $T$'s terms identical. No enhancements to $*$ will change this fact, so it is unnecessary to apply $*$ to the leftward parts of $T$'s terms that have already been taken care of. It is also useless to apply the "unifier-in-progress" to most of the rightward portions of the terms, as well, since the results are not used immediately and are also likely to change as $*$ changes. This leads to a commonly used recursive approach in which disagreements among $T$'s original set of terms are isolated one by one, and the "unifier-in-progress" is applied to only one such disagreement (a simple pair of terms) at a time. This is viable since it is clear that no "unifier-in-progress" can make any initially identical portions of $T$'s terms unidentical, so it is sufficient to confine attention to those portions of $T$'s terms that are in disagreement at the outset.

However, regardless of how uncomfortable the
inefficiency of line 3 may make one feel, that operation is not the fundamental source of the algorithm's inefficiency. The algorithm has a worst-case complexity of no better than $O(2^n)$ as a result of the so-called "occur check" at line 13. Worse yet, even without the occur check the algorithm has the same terrible computational complexity due to the rewriting of $T\$ at line 3. This will be considered in more detail after we have discussed some other salient attributes of the algorithm.

2.5. Failure to Unify

The algorithm fails if the conditions at lines 8, 10, or 14 are met. By "fail", it is not meant that the algorithm is unable to find a unifier, but rather that a unifier for the given terms does not exist.

The reason for a failure at line 8 is an arity mismatch among functions. It is simply impossible to find a substitution that can make something like $f(U)$ and $f(U,V)$ identical.

Unification fails at line 10 if the disagreement set is not empty but contains no variables. The whole point of unification is to eliminate differences by replacing variables, but this is impossible if the differences include no variables to replace. It means that the disagreement set consists solely of functions. These functions must have different function symbols (or else there would be no disagreement; if functions have the same function symbol but
differ at some argument position, the disagreement set consists of the terms at that argument position). It is impossible to unify two functions that have different function symbols. For example, no substitution can make $f(X)$ and $g(X)$ identical.

2.6. Defining the Occur Check

The final cause of failure is the so-called "occur check" at line 13, which detects an attempt to unify a variable and a function where the string of symbols making up the function's argument list contains the given variable anywhere in it. For example, the variable $X$ occurs in $f(X)$. It also occurs in $f(Y, g(Y, h(Y, i(Y, X))))$. It is impossible to unify such a pair of terms because any time the substitution is applied it will increase the length of both terms by exactly the same amount; the function-term will always remain longer than the variable-term, so no substitution can make the two identical.

In the Robinson algorithm, the occur check also prevents an infinite loop. For example, assume we are trying to unify $T = \{f(X), X\}$ without doing the occur check. $\$ quickly becomes $(f(X)/X)$ and stays that way: on each iteration, $\$ is applied to $T$ to yield $Q = \{f(f(X)), f(X)\}$, which always has the same disagreement set, $(f(X), X)$. The binding $f(X)/X$ is created from the disagreement set and is used to update $\$, which does not change because $\$ already is $(f(X)/X)$. The process never ends.
Some of the other unification algorithms do not go into an infinite loop if the occur check is omitted. If the occur check is omitted from them they can erroneously report that, say, \(X\) and \(f(X)\) are unifiable.

### 2.7. The Computational Complexity of Robinson's Algorithm

A perverse and well-known example is often used to show how Robinson's algorithm exhibits \(O(2^n)\) computational complexity (Bibel 1982; Lloyd 1987):

**Example 2.3. Robinson's Devil.**

Let

\[
T = \{ p(f(X_0,X_0), f(X_1,X_1), \ldots, f(X_{n-1},X_{n-1})),
\]
\[
p( X_1, X_2, \ldots, X_n ) \}.
\]

On successive iterations of the algorithm, the occur check at line 13 checks whether

- \(X_1\) occurs in \(f(X_0,X_0)\),
- \(X_2\) occurs in \(f(f(X_0,X_0), f(X_0,X_0))\),
- \(X_3\) occurs in \(f(f(f(X_0,X_0), f(X_0,X_0)), f(f(X_0,X_0), f(X_0,X_0)))\),
- \(\ldots\)
- \(X_k\) occurs in (term with \(2^{k-1}\) \(f\) symbols; \(2^k\) \(X_0\) symbols),
- \(\ldots\).

Thus, for variable \(X_k\), the occur check must scan and compare a number of symbols that is exponential with respect to \(k\). The occur check incurs a significant computational cost.

In this particular example the occur check is unnecessary; variable \(X_k\) can never occur in \(f(\ldots X_{k-1} \ldots)\). But even if the occur check is eliminated, the algorithm
still retains exponential complexity when the same example is used: on each iteration, the binding added to $\$ is exponentially larger that the binding added in the prior iteration. Consequently, line 15 involves a composition of substitutions whose complexity increases exponentially. Similarly, the application of $\$ to $\text{T}$ at line 3 incurs an expense that grows exponentially.

Assume, for the sake of argument, that this terribly pathological example did not exist. The algorithm would still have at least $O(n^2)$ complexity. Consider:

**Example 2.4.** Robinson's Imp.

\[
\text{T} = (f(X_1, X_2, X_3, \ldots, X_n), \\
f(X_0, X_0, X_0, \ldots, X_0)).
\]

On successive iterations, $\$ is set to

\[
(X_1/X_0), \text{ then} \\
(X_2/X_0, X_2/X_1), \text{ then} \\
(X_3/X_0, X_3/X_1, X_3/X_2), \text{ then} \\
\ldots \\
(X_n/X_0, X_n/X_1, \ldots, X_n/X_{n-1}).
\]

Furthermore, on each iteration, $\$ is applied to $\text{T}$, and all $n$ occurrences of $X_0$ are replaced at line 3. The algorithm performs $n$ iterations, so at line 3 there are $n$ iterations * $n$ replacements per iteration = $n^2$ replacements.

The example that results in $O(2^n)$ complexity is well known in the literature and is here called Robinson's Devil. The example that results in $O(n^2)$ complexity is named
Robinson's Imp in this paper. It will be seen that Robinson's Imp causes significant difficulty for many unification algorithms.

2.8. The "Impossibility" of Linear Unification

Robinson's Devil is particularly perverse in that just printing the mgu requires an exponential number of symbols, and thus exponential time. That is, even if some algorithm could find an mgu for Robinson's Devil in linear time, the act of reporting the result would immediately make the algorithm exponential. In some sense, then, linear unification is impossible.

The efficient unification algorithms avoid this conundrum by using a more compact representation of mgus, which consists of 1) a list of bindings and 2) a rule for applying the bindings one to another (generally left-to-right) in such a way that a fully explicit mgu could be generated if desired. To illustrate, consider the unification

\[
\frac{h(f(X), f(Y), f(Z), f(W))}{h(Y, Z, W, U)},
\]

which has mgu

\[
\{ f(X)/Y, f(f(X))/Z, f(f(f(X)))/W, f(f(f(f(X))))/U \}.
\]

The mgu is represented more compactly as the (ordered) list

\[
[ f(X)/Y, f(Y)/Z, f(Z)/W, f(W)/U ]
\]

plus the rule that the fully explicit mgu can be obtained by moving left-to-right through the list, applying each binding
to the numerators of the bindings that follow it. For example, \( f(X)/Y \) is substituted into \( f(Y)/Z \) to yield \( f(f(X))/Z \), which is later substituted into \( f(Z)/W \) to yield \( f(f(f(X)))/W \), and so on.

In the case of Robinson's Devil, the explicit mgu
\[
\{ f(X_0,X_0)/X_1, f(f(X_0,X_0),f(X_0,X_0))/X_2, \ldots, --/X_n \}
\]
has an exponential number of symbols with respect to \( n \). When written in the new list form, the same mgu can be represented by a number of symbols that is linear with respect to \( n \):
\[
[ f(X_0,X_0)/X_1, f(X_1,X_1)/X_2, \ldots, f(X_{n-1},X_{n-1})/X_n ].
\]

The use of this representation is not a deception or a delusion; it is not meant to hide the true complexity of unification. It is viable in the logic programming context, because there, the purpose of unification is not to find and report an mgu. In answering a query, a logic programming system finds a chain of inferences, with each link in the chain having its own mgu. That is, a query yields a sequence of mgus. An answer is obtained by taking the composition of these mgus and extracting the bindings for variables mentioned in the original goal (the composition may include many unrelated bindings, i.e., bindings for variables not present in the query). From a practical standpoint, then, the criterion for accepting some compact representation of mgus must be that finding the composition of the compact mgus is easy and inexpensive.
Baxter (1976b) shows that the aforementioned compact representation of mgus is merely an mgu expressed as a composition of substitutions.

2.9. Simplifying the Study of Unification

We have already made two simplifying assumptions in studying unification. We first assumed that we can confine discussion to terms alone, since for our purposes terms logically subsume the concepts of expressions, functions, predicates, constants, and variables. We have also simplified the problem by noting that we can represent mgus in a compact list form.

Robinson's algorithm is general in so far as it can unify any finite number of terms. It is sufficient, however, to consider only algorithms that unify two terms at a time, for the unification of n terms can be accomplished by performing n-1 unifications of pairs of those n terms in a prescribed manner. Specifically, unifying

\[ \{t_1, t_2, t_3, \ldots, t_n\} \]

is equivalent to performing the sequence of unifications

\[ \{t_1, t_2\} \]
\[ \{t_1, t_3\} \]
\[ \ldots \]
\[ \{t_1, t_n\}. \]

The justification for this is that 1) unification is the process of making n terms identical to one another, so 2) if one can pick one term and make the other n-1 terms identical
to it, the n-1 other terms are necessarily made equal to one another. Yang (1989) provides a formal proof.

A benefit of this simplification is that it can reduce some nondeterministic unification algorithms to deterministic ones. In particular, it does so for Robinson's algorithm. In that algorithm, there is a point where it is necessary to choose a variable and some other term from the disagreement set, which can contain as many as n terms if n terms are being unified. But if only two terms are unified at a time, the disagreement set will always have exactly two terms. In consideration of our confining attention strictly to variables and functions, the two terms in the disagreement set will either be 1) both variables, or 2) a variable and a function, or 3) two functions. The cases are well-defined, and the nondeterministic instruction to select a variable and some other term can be replaced by the deterministic consideration of the three possible cases.

As previously mentioned, in logic programming a query results in a sequence of mgus that are, in the end, composed one with the other. Several mgus may have different bindings for the same variable. For example, one mgu may include the binding $f(Y)/X$, while another mgu may include $f(Z)/X$. Consequently, implementations of unification usually include indicators of the environment within which each variable occurs. Environment is employed in Cohen (1985) and in Robinson's example implementation (Robinson
1979), and is discussed by Martelli and Rossi (1986). However, environment is not a factor in any individual unification. It becomes a factor only when the results of separate unifications must be combined in some way. Environment is not directly relevant to the study of unification, except to say that a unification algorithm will be of practical use only if it employs data structures that are amenable to convenient manipulation, especially in terms of environment. For maintaining clarity we can therefore ignore environment while studying or developing unification algorithms.

We therefore have the following simplifications, which make the study of unification easier while implying no loss of generality:

1) Consider only functions and variables, understanding that expressions, predicates, constants, etc. can be treated as special cases.

2) Represent mgus in a list form (along with an implied operation for transforming the list into a true mgu).

3) Consider only the unification of two terms at a time.

4) Ignore environments.

2.10. Finding Differences

The aim of unification is to identify the differences between two or more terms and to build a substitution that can then eliminate those differences. Robinson's algorithm
repetitively applies an mgu-in-progress to terms until no differences remain. Each time the algorithm checks for differences it can waste resource, because if two subterms matched in a previous step, they will still match in the present step and in all that follow. There is no need to verify over and over that particular subterms either match in the original terms or match in substituted terms.

For example, in unifying

\[ f(X,Y,Z,T,U) \] and

\[ f(X,Y,Z,V,W), \]

Robinson's algorithm first determines that \( X=X, Y=Y, \) and \( Z=Z, \) but that the difference between \( T \) and \( V \) must be resolved. It resolves the difference, then once more determines that \( X=X, Y=Y, \) and \( Z=Z, \) and also that \( T \) has been made equal to \( V, \) finally discovering that \( U \) and \( W \) are different. The algorithm resolves this difference, then once more determines that \( X=X, Y=Y, \) etc.

It is manifest that this repetition is unnecessary. If subterms match in the original terms being unified, they will still match in all subsequent steps of the algorithm. It is sufficient to isolate the terms' initial differences and to ignore the subterms that match.

The following algorithm isolates and prints the differences between two terms. With an eye toward use for unification it also does a bit more: it prints the differences in a form resembling bindings. Each difference
Algorithm 2.3. udiff.
Isolate unifiable differences between terms.
udiff(t1,t2) {
    case V(t1),V(t2): if (t1!=t2)
        print(t2/t1);
    case V(t1),F(t2): print(t2/t1);
    case F(t1),V(t2): print(t1/t2);
    case F(t1),F(t2): if function names differ
        fail("function name mismatch");
        if arities differ
            fail("function arity mismatch");
        for i = 1 to arity
            udiff(arg i of t1, arg i of t2);
}

The algorithm uses V(t), which is true if term t is a
variable and false otherwise. Similarly, F(t) is true if t
is a function and false otherwise.

If corresponding subterms are functions that differ
either in name or arity, it is impossible to find a variable
replacement that will make the two functions identical.
Since the algorithm is intended to identify only those
differences that can be resolved by unification, it reports
failure if such a mismatch is encountered.

In some cases the udiff algorithm by itself is able to find and report an mgu. For example, given

\[ h(U, W, Y) \] and
\[ h(V, X, Z), \]

it prints out \( U/V, W/X, \) and \( Y/Z \), which, when considered as a set, constitutes the mgu of the two expressions.

Similarly, for

\[ h(f(g(X, Y), Z)) \] and
\[ h(f(g(X, W), g(U, V))), \]

the algorithm prints \( Y/W \) and \( g(U, V)/Z \), representing the mgu of the two expressions.

But consider

\[ h(V, X, Z) \] and
\[ h(W, Y, W). \]

The algorithm prints \( \{V/W, X/Y, Z/W\} \). There are two bindings for \( W \), so this is not even a substitution, much less an mgu.

Next, consider

\[ h(X, Y) \] and
\[ h(Y, Z). \]

The algorithm prints \( \{X/Y, Y/Z\} \), which is a substitution but is not a unifier.

Finally, consider

\( f(X) \) and
\( X. \)
The algorithm prints \( \{f(X)/X\} \), which is not a unifier because \( X \) occurs in \( f(X) \).

So udiff by itself is not a unification algorithm, because there are cases where it:

1) Prints multiple bindings for a single variable.
2) Produces underdeveloped substitutions, not unifiers.
3) Prints bindings that fail the occur check.

2.11. Chains of Bindings

A set of bindings may have distinct denominators, making the set a substitution, but it may still not be a unifier. This happens when there is a "chain" of references among bindings. Consider, for example, \( T = \{f(X,Y), f(Y,Z)\} \), for which udiff produces the set of bindings \( B = \{X/Y, Y/Z\} \). \( B \) is not a unifier for \( T \) since \( TB = \{f(X,X), f(X,Y)\} \), which is not a singleton. There is a "chain" in \( B \) because the denominator of \( X/Y \) is equal to the numerator of \( Y/Z \). If this chain is eliminated by substituting \( X \) for \( Y \) in \( Y/Z \), the resulting \( B' = \{X/Y, X/Z\} \) is a unifier.

All of the unification algorithms that are claimed to be efficient are satisfied to produce a substitution having chains of bindings rather than a true unifier. They require that the substitution be written in such a way that the chain proceeds in one direction only, either forwards or backwards, thus precluding cycles.
2.12. Clashing Bindings

We will now consider how to transform a set of bindings into a substitution. Since a substitution is a set of bindings in which no two bindings have the same denominator, the transformation consists of eliminating bindings which clash.

Consider the unification

\[ f(X_1, X_3, X_5, X_7) \]
\[ f(X_2, X_4, X_6, X_6), \]

which has the list of differences

X_1 X_3 X_5 X_7
X_2 X_4 X_6 X_6

with two bindings for variable X_6.

We can proceed through the list left to right. The list of bindings can be considered a substitution until we encounter a clash. If we can eliminate that clash, then we can proceed on through the list, perhaps eliminating additional clashes. By the time we reach the end of the list, we will have eliminated all clashes. Therefore we will have transformed the set of bindings into a substitution.

The basic idea behind eliminating clashes is to consider how Robinson's algorithm works on a pair of terms. At any step, Robinson's algorithm has a substitution that has been constructed by comparing the leftmost \( k \) subterms of the two terms. That substitution is applied to all the subterms of both the original terms.
The analogous operation in our list structure is to find the longest initial substitution. The next binding, then, must be a clash (unless the complete list is already a substitution). We then apply this leftmost substitution to the numerator and denominator of the clashing binding.

The clash is thereby transformed into a new fraction (not necessarily a binding). The operation may have replaced the clash with a function over a function. Or it may have replaced it with a variable over a function. Neither of these are bindings. It may have replaced the clash with a binding, yet the binding may be a new clash. In any case, the analogy with Robinson's algorithm can be maintained by replacing the new fraction \( s/t \) with the list produced by \( \text{udiff}(s,t) \). This guarantees that the list remains a list purely of bindings. It also identifies what would be the next disagreement set in the Robinson algorithm.

The action to take for eliminating clashes in our list structure can be depicted as follows. Assume a list of bindings. Find the first clash \( t/V \):

\[
\ldots \frac{s}{V} \ldots \frac{t}{V} \ldots
\]

All the bindings to the left of \( t/V \) constitute a substitution. Call it \( \theta \). Replace \( t \) by \( t\theta \). \( V\theta \) happens to be \( s \), the numerator of the binding with which \( t/V \) clashes. So replace \( V \) by \( s \). The list now looks like:
But tθ/s may not be a binding. Replace it by udiff(tθ,s):

... s V s ...

Repeat the process until there are no more clashes.

The process is more succinctly described by:

**Algorithm 2.4. CNF: Create New Fraction.**

Eliminate clashing bindings.

while (there exists a leftmost clash t_i/V_i) {

  let θ = {t_1/V_1,...,t_{i-1}/V_{i-1}};

  replace t_i/V_i by udiff(t_iθ,V_iθ);

}

The idea represented by this algorithm can be seen in
Robinson's informal algorithm (Robinson 1979), Baxter (1973,
1976a, 1976b), and Huet (1976), and is also similar to the
"create new fraction" (CNF) operation of Li, Yang, and Ng
(1988) and the "create new binding" (CNB) of Yang (1989).

Unfortunately, this approach has a poor worst-case
complexity. When a clash is resolved, the new binding may
clash with still another variable. When this new clash is
resolved, the result can be a clash once more. And so on.

In a list of bindings having the following structure, a
clash at binding k will take k-1 steps to eliminate.

```
X1 X2 ... Xk-1 X0
X2 X3 ... Xk Xk
^ ^ ^ ^ ^
```

Binding #: 1 2 k-1 k ...
Figure 2.1 illustrates by way of a specific example.

### Figure 2.1. Eliminating Clashes.

<table>
<thead>
<tr>
<th>Portion</th>
<th>Substitution</th>
<th>Clash</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>X1 X2 X3 X4</td>
<td>X0</td>
<td>X5 clashes; replace by X4.</td>
</tr>
<tr>
<td></td>
<td>X2 X3 X4 X5</td>
<td>X5</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>X1 X2 X3 X4</td>
<td>X0</td>
<td>X4 clashes; replace by X3.</td>
</tr>
<tr>
<td></td>
<td>X2 X3 X4 X5</td>
<td>X4</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>X1 X2 X3 X4</td>
<td>X0</td>
<td>X3 clashes; replace by X2.</td>
</tr>
<tr>
<td></td>
<td>X2 X3 X4 X5</td>
<td>X3</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>X1 X2 X3 X4</td>
<td>X0</td>
<td>X2 clashes; replace by X1.</td>
</tr>
<tr>
<td></td>
<td>X2 X3 X4 X5</td>
<td>X2</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>X1 X2 X3 X4 X0</td>
<td>X0</td>
<td>Finished.</td>
</tr>
<tr>
<td></td>
<td>X2 X3 X4 X5 X1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

One can imagine a list of n bindings in which each binding $k$ ($k=1..n$) requires $k-1$ steps of clash elimination. The total number of clash eliminations would then be $\sum_{k=1}^{n}(k-1) = (n-1)n/2$, or $O(n^2)$. In fact, this worst-case scenario is created by the simple pattern:

\[
\begin{align*}
X_1 & \quad X_2 & \quad \ldots & \quad X_n \\
X_0 & \quad X_0 & \quad \ldots & \quad X_0
\end{align*}
\]

This pattern is a manifestation of Robinson's Imp. An example of the $O(n^2)$ behavior is provided in Figure 2.2.
Figure 2.2. Worst-case Resolution of Clashes.

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td>X2</td>
<td>X3</td>
<td>X4</td>
</tr>
<tr>
<td>X0</td>
<td>X0</td>
<td>X0</td>
<td>X0</td>
</tr>
</tbody>
</table>

| X1 | X2 | X3 | X4 | X5 |
| X0 | X1 | X0 | X0 | X0 |

Binding 3 causes 2 CNFs

<table>
<thead>
<tr>
<th>Binding 4 causes 3 CNFs</th>
</tr>
</thead>
</table>

Binding 5 causes 4 CNFs

Algorithms based on the foregoing clash elimination technique have difficulty handling Robinson's Imp. Eliminating this poor, $O(n^2)$, complexity is difficult. But the problem does yield to analysis when viewed in the light of equivalence classes, the subject of the next chapter.
3.1. Introduction

An aim of unification is to make terms syntactically identical. In the extreme, the $n$ terms of a set can be made identical simply by selecting any one of the terms to replace all the others. For example, the functions $f(X)$, $g(Y,Z)$, and $h(U)$ can be made identical by replacing each of the three by $g(Y,Z)$. But the definition of unification requires that a constraint be observed: only variables can be replaced. In other words, in unification it is not sufficient merely to make a set of terms syntactically identical; it must be done by replacing variables only.

Many unification algorithms are careful to enforce this constraint at all stages; these are the "slow" ones. In contrast, a linear algorithm such as Paterson and Wegman's (1978) first computes replacements -- whatever their form may be -- then refines them to satisfy the unification constraint.

3.2. Equivalence Relations

The process of making terms identical by selecting one to replace the others will be called "equating" them (after Robinson (1979)).
In equating terms, it does not matter what term is replaced by another. Thus the replacement of terms constitutes an equivalence relation since it is (trivially) reflexive, symmetric, and transitive:

1) A term can replace itself, so the relation is reflexive.

2) It does not matter which of two terms replaces the other, so the relation is symmetric.

3) If s replaces t, and t replaces u, then s replaces u, so the relation is transitive.

3.3. Equating Functions

If two functions have the same name and arity, they can be made syntactically identical either by equating them directly, or by equating their respective arguments.

Equating two terms is the same as saying they are in the same equivalence class. Sizeable equivalence classes naturally emerge as a consequence of equating functions' arguments. For example, given

\[ f(X_1, X_2, X_3, X_4, Y_1, Y_2, Y_3, Y_4) \text{ and } f(X_2, X_3, X_4, X_5, Y_2, Y_3, Y_4, Y_5), \]

it is clear that

\[ X_1 \text{ and } X_2 \text{ are in the same equivalence class, } X_2 \text{ and } X_3 \text{ are in the same equivalence class, } X_3 \text{ and } X_4 \text{ are in the same equivalence class, etc.} \]
3.4. Interpreting Equivalence Classes as Substitutions

The meaning of an equivalence class is that any one member can represent the class as well as any other with respect to some given property. The selected member is called the \textit{representative} of the equivalence class and is identified by enclosing it in square brackets \([]\). All the members of the class can be replaced by the representative and the property in question will be preserved.

This suggests a relationship between equivalence classes and substitutions. The meaning of a representative can (sometimes) be expressed in terms of a substitution in which the representative replaces all the other terms in the class. For example, assume an equivalence class of variables \(X_1,\ldots, X_4\). Then

\[
\begin{array}{ll}
\text{Equivalence class} & \text{Corresponds to substitution} \\
\{[X_1], X_2, X_3, X_4\} & \{X_1/X_2, X_1/X_3, X_1/X_4\} \\
\{X_1, [X_2], X_3, X_4\} & \{X_2/X_1, X_2/X_3, X_2/X_4\} \\
\{X_1, X_2, [X_3], X_4\} & \{X_3/X_1, X_3/X_2, X_3/X_4\} \\
\{X_1, X_2, X_3, [X_4]\} & \{X_4/X_1, X_4/X_2, X_4/X_3\}
\end{array}
\]

In other words, given an equivalence class of terms, it is possible to create a set of "directions" for replacing all the terms in the class by a representative. In some cases this set of directions can be written as a substitution, interpreted as a set of replacements for variables only. In
other cases the directions can be written in a form that resembles a substitution but is not, due to functions existing in the denominators.

This idea leads to a generalization of bindings and substitutions, formalized in the definitions below.

**Definition 3.1.** A *t-binding* is a pair of terms $s$ and $t$, written $s/t$.

In the same way that a binding associates a term with a variable, a t-binding associates a term with some other term. A binding is a special case of a t-binding where the denominator is a variable.

**Definition 3.2.** A *t-substitution* is a set of t-bindings $\{s_1/t_1, \ldots, s_n/t_n\}$ such that all $t_i$ are distinct and no $t_i = s_i$.

It is easy to see that a substitution is a special case of a t-substitution where all the denominators are variables.

For any representative of an equivalence class of terms, there is an associated t-substitution:

**Definition 3.3.** Let $Q = \{t_1, \ldots, t_n\}$ be an equivalence class of terms. Let $t_k$ be any member of $Q$ selected as the representative, written $[t_k]$. The t-substitution $\{t_k/t_1, \ldots, t_k/t_{k-1}, t_k/t_{k+1}, \ldots, t_k/t_n\}$ is called a **representative t-substitution** with respect to $t_k$, and is denoted $[t_k]^+$.

In some cases, a representative t-substitution is also
a substitution, i.e., all its denominators are variables. This depends upon the form of the equivalence class and the choice of representative.

3.5. Types of Equivalence Classes

Definition 3.4. Let Q be an equivalence class of terms. Q is

- **trivial** if it contains no functions,
- **simple** if it contains exactly one function, and
- **compound** if it contains more than one function.

**Example 3.1.**

\[
\{X_1, X_2, X_3, X_4\} \quad \text{is trivial.}
\]

\[
\{f(X_1), X_2, X_3, X_4\} \quad \text{is simple.}
\]

\[
\{f(X_1), f(X_2), X_3, X_4\} \quad \text{is compound.}
\]

Let functions be denoted by \(f, f_1, f_2, \text{ etc.}\), and variables by \(V_1, V_2, \text{ etc.}\). Remembering that a substitution is a \(t\)-substitution with only variables in the denominators of \(t\)-bindings, it is clear that:

1) If \(Q = \{V_1, \ldots, V_n\}\) is a trivial equivalence class, then any \(V_i\) can be selected as the representative and \([V_i]^+\) will be a substitution.

2) If \(Q = \{f, V_1, \ldots, V_n\}\) is a simple equivalence class, then a substitution can be formed only if the function term is selected as the representative. That is, \([f]^+\) is a substitution, but for every \([V_i]^+\), there is a \(t\)-binding with \(f\) in the denominator, so no \([V_i]^+\) can be a substitution.
3) If $Q$ is compound, then no substitution can be formed since for any choice of term $t$ as representative, $[t]^+$ will have at least one $t$-binding with a function in the denominator.

3.6. A Plan for Generalizing Unification

The plan of subsequent sections (after a few necessary detours) is to generalize the notion of unification. Informally, a set $T$ of terms is unified by any operation that makes $T$ a singleton. So far, the only way to make $T$ a singleton is to apply a substitution. But by extending definitions, it will be possible to unify $T$ by applying a $t$-substitution. Then since a $t$-substitution is derived from an equivalence class of terms, it will be possible to talk of unifying $T$ by applying a set of terms. Finally, it will be seen how to unify $T$ by applying a set of terms that is derived from $T$ itself.

3.7. Unifying with $t$-substitutions

Definition 3.5. Let $S = \{s'_1, \ldots, s'_m\}$ be a set of terms and $\theta = \{s_1/t_1, \ldots, s_n/t_n\}$ a $t$-substitution. The $t$-instance of $S$ by $\theta$, written $S^*\theta$, is the set of terms obtained by replacing each $s'_k$ in $S$ by $s_i$ where $t_i = s'_k$.

Now we generalize the notion of a unifier to include $t$-substitutions.

Definition 3.6. Let $S$ be a set of terms and $\theta$ a $t$-substitution. $\theta$ is a unifier for $S$ if $S^*\theta$ is a singleton.

It is CRITICAL to understand the distinction between
substitutions and instances on the one hand, and
t-substitutions and t-instances on the other. A
substitution can be used to replace the variables within
functions by other terms. By contrast, a t-substitution
replaces entire terms, and cannot replace functions or
variables that occur within functions. For example, let
\[ T = \{ f(X), f(Y), Z \} \]
be a set of terms,
\[ S = \{ W/X, W/Z \} \]
a substitution, and
\[ \$ = \{ W/X, W/Z, U/f(Y) \} \]
a t-substitution.
Then \( TS = \{ f(W), f(Y), W \} \), whereas
\[ T*\$ = \{ f(X), U, Z \}. \]
In \( T*\$ \), the \( f(X) \) in \( T \) is not touched; even though there is a
replacement for \( X \), it occurs within a function. The only
way that \( \$ \) could change \( f(X) \) is if there were a replacement
for the entire term \( f(X) \) (as is the case with \( f(Y) \) in \( T \)).

This difference in meaning must be reflected by
notation. That is why an instance is written \( TS \), while a
t-instance is written \( T*\$ \).

3.8. Equivalence Classes in Graph Form

In unifying two functions, it is usual to pairwise
inspect the functions' arguments, recording the fact that
the first argument of one function is in the same
equivalence class as the first argument of the other, and so
on, for all the arguments in both functions (which are
assumed to have the same arity).

It is natural to represent an equivalence class of
terms as a graph. The nodes are terms. An (undirected) edge is drawn between two terms whenever it is determined that the two terms are in the same equivalence class. With this representation, when given a term, it is possible to list all the members of the term's equivalence class by following all possible paths from the term, i.e., by computing the transitive closure.

Assume the use of the udiff algorithm, which when given a pair of terms to unify, returns a list of the differences, represented as bindings. To put the list of bindings into a graph, define a new algorithm as follows:

Algorithm 3.1. graph.
graph(list-of-bindings) {
    for (each binding t/V in list-of-bindings) {
        if (node(t) does not exist)
            create(node(t));
        if (node(V) does not exist)
            create(node(V));
        create edge(node(t),node(V));
    }
}

Example 3.2. Let $T = \{f(X_1,X_2,X_3,X_4), f(X_2,X_3,X_4,X_5)\}$. Then $\text{graph}(\text{udiff}(T))$ is:

$$X_1 \rightarrow X_2 \rightarrow X_3 \rightarrow X_4 \rightarrow X_5$$

Example 3.3. Let $T = \{f(X_1,X_2,X_3,X_4), f(X_0,X_0,X_0,X_0)\}$. Then $\text{graph}(\text{udiff}(T))$ is:

$$X_0 \rightarrow X_4$$

There is an obvious correspondence between the type of
an equivalence class and the structure of its graph. Hereafter, we will refer to types of graphs, saying that the graphs of simple, trivial, and compound equivalence classes are, respectively, trivial, simple, and compound.

3.9. Substitutions from Trivial Graphs

In both of the examples 3.2 and 3.3, T is trivial. Given the graph of a trivial equivalence class, it is easy to print a substitution by starting from any node in the graph. The following algorithm suffices.

**Algorithm 3.2. list-trivial.**

```plaintext
list-trivial(graph) {
    select any node from graph;
    list(node);
}

list(node) {
    mark(node);
    for (each unmarked sibling s) {
        print(node"/"s);
        list(node);
    }
}
```

For example 3.2, the list-trivial algorithm prints "X1/X2 X2/X3 X3/X4 X4/X5".

In the last chapter we saw that a problem such as example 3.3 can require $O(n^2)$ complexity to render a substitution when a "create new binding" approach is followed. However, by using the graph and list-trivial algorithms, the substitution is rendered with linear complexity. Given X0, list-trivial prints "X0/X1 X0/X2 X0/X3 X0/X4". Given X1, it prints "X1/X0 X0/X2 X0/X3
3.10. Substitutions from Simple Graphs

Simple equivalence classes cannot be handled as easily as trivial equivalence classes.

Example 3.4. Let \( T = \{ h(f(X), X_2, X_3), h(X_2, X_3, X_4) \} \). Then \( \text{graph}(\text{udiff}(T)) \) is:

\[
f(X_1) \rightarrow X_2 \rightarrow X_3 \rightarrow X_4
\]

In this case the list-trivial algorithm cannot be applied to an arbitrary node. If it were applied starting at node \( X_2 \), for example, its printout would include \( X_2/f(X_1) \), which is not a binding. The example has only one node that can be used as the starting point for list-trivial: the function node, \( f(X_1) \). The corresponding enhanced algorithm, modified to handle both simple and trivial classes, is:

Algorithm 3.3. list-simple.

```cpp
list-simple(graph) {
    if there is a function node in the graph
        select it;
    else
        select any node from graph;
    list(node);
}
list(node) {
    mark(node);
    for (each unmarked sibling s) {
        print(node"/"s);
        list(node);
    }
}
```

3.11. Simplifying Compound Equivalence Classes

A compound equivalence class can be made simple by unifying the functions in it.

Example 3.5. Let \( T = \{ h(f(X), f(Y)), h(W, W) \} \). Then
graph(udiff(T)) is

\[ f(X) \rightarrow W \rightarrow f(Y) \]

and the equivalence class represented by the graph is \( E = \{ f(X), f(Y), W \} \). \( E \) can be made simple if \( f(X) \) and \( f(Y) \) can be unified, which they can by application of the substitution \( \Theta = \{ X/Y \} \). Thus, \( E\Theta = \{ f(X), f(Y), W \}(X/Y) = \{ f(X), W \} \).

Many unification algorithms ensure that only simple equivalence classes exist at the end of each step. That way, at the end of the algorithm it is always possible to print a set of bindings (since a set of bindings can always be produced from a simple or a trivial equivalence class). These algorithms follow a policy of reducing a compound equivalence class to a simple one as soon as the condition is discovered. The policy is dictated by the algorithms' choice of abstract data structure for equivalence classes.

3.12. Equivalence Class Topologies

As already seen, an equivalence class can be represented by a graph of nodes connected by (undirected) edges. This is completely general; any node can be selected as the representative of the class, and any node can be reached from any other.

But unification requires trivial or simple equivalence classes -- those having at most one function. That one function, if it exists, has to be the representative of the class, or else it is not possible to produce a substitution. So it is sufficient to represent an equivalence class by a
structure in which there is a designated position for a function. All other nodes, consisting only of variables, are then linked to the function node by way of arcs, either directly or transitively. Possible topologies include:

\[ f(X) \]
\[ f(X) \leftarrow Y \leftarrow Z \leftarrow W \]

Star List Tree

The advantage of this approach is that any such graph can be implemented very economically: each variable need have only one pointer associated with it.

Given a term, then, it is very easy to determine the equivalence class the term is in. If the term is a function, it must be the representative of its own equivalence class. If the term is a variable, it is necessary only to follow pointers until a function is reached, in which case the class's "name" has been discovered.

In the case of trivial equivalence classes, the designated "function" position is occupied by any variable in the class (usually the first one encountered). That variable then acts as the class's representative, and all other variables in the class point to it either directly or transitively.
The choice of topology affects the cost of looking up a variable's equivalence class representative. In a star topology, all variables point directly to the representative, so the lookup can be accomplished in constant time. In the list topology, the cost of lookup varies by the position of the variable in the list. Likewise for the tree topology.

Looking up a node's representative is called a find in the literature.

However, the data structure must accommodate more than just a find: it must also support the merging of equivalence classes. The reason is that a unification algorithm does not "know" the equivalence classes "all at once"; it must "discover" the equivalence classes by comparing terms pairwise. At any one time, the algorithm has built structures that represent the problem's equivalence classes as best the algorithm has been able to discover. But a subsequent pair-equivalence can reference nodes from two different equivalence class structures, meaning that the two "equivalence classes" are actually one and must therefore be merged into a single structure. As an example, consider the unification of

\[ f(X_1,X_2,X_3,X_5,X_6,X_7,X_4) \]
\[ f(X_2,X_3,X_4,X_6,X_7,X_8,X_5). \]

Each function contains 7 subterms. After the first 6 pairs of subterms have been inspected, the graph of the
equivalence classes contains two disjoint components, looking like:

X1—X2—X3—X4       X5—X6—X7—X8

But when the 7th pair of subterms (X4,X5) is encountered, it is seen that an edge must be constructed between X4 and X5. In other words, the two equivalence classes must be merged into one.

The choice of topology makes a significant difference in the cost of merging equivalence classes.

Consider, for example, the cost of merging two "stars" A and B to create a new "star" C, where A's representative is to be retained in C. To merge the two stars, every node of B must be modified to point to the representative of A. In general, the cost of merging two stars is at least proportional to the number of nodes in the smaller star.

In contrast, consider a tree topology. In merging trees A and B to create tree C, it is necessary only to set a pointer from B's representative to any node in A. So with a tree topology, a merge has unit cost.

The process of merging two equivalence classes is called a union.

It is an unpleasant truth that a data structure facilitating an inexpensive find will have a more expensive union, and vice-versa. The best known data structure for accommodating the union-find problem is a balanced tree, as developed by Tarjan (1975) and described by Horowitz and
Sahni (1982). The combined cost of union and find can be reduced to $O(n \times G(n))$, where $G(n)$ is the extremely slowly-growing functional inverse of Ackermann's function, so that $n \times G(n)$ is almost linear.

The algorithms of Baxter, Huet, and others are fettered by the constraints of union-find. Presented with two subterms, these algorithms must first find what equivalence class each subterm is in, then secondly must merge the equivalence classes.

3.13. Avoiding Union-Find -- Achieving Linearity

In the implementation of unification, we typically examine one pair of (sub)terms at a time, understanding that each pair represents two members of some equivalence class. To divine the complete equivalence class, we store the pair-by-pair information in some data structure, be it abstract or concrete. That is, given two terms, we must first obtain the node associated with each term. Then we must store at least enough information to allow the equivalence between the two terms to be recovered at some later time. The simplest way to do this is either to set a pointer from one term (node) to the other or to have both terms point to each other. In either case, the cost of updating the data structure to reflect the new equivalence information is constant -- as long as there is no limit on the number of pointers a node may have. But if nodes are restricted to some fixed maximum number of pointers, it is
always possible to find some unification example in which the number of equivalences some node has is greater than the number of pointers available for that node. Representing a new equivalence in such a case requires following at least one pointer to some other node and establishing a link from there. This means that it is impossible always to update such a limited-pointer structure in constant time, and implies that there is always an example for which maintaining the data structure has greater than linear complexity. There is reason to believe, therefore, that the unification algorithms that are based on the use of spare data structures -- typically just one or two pointers per node -- cannot be linear.

Paterson and Wegman place no restriction on the number of pointers (or edges) a node may have. Consequently, given two equivalent terms, the algorithm can always update the equivalence class structure in constant time.

The updating of the equivalence class structure corresponds to a union. Paterson and Wegman have clearly selected a structure that minimizes the cost of doing unions.

What, then, about finds? We concluded in a previous section that the simpler the union, the more complex the find, and vice-versa. What is the cost of the find done by the Paterson and Wegman algorithm?

The cost is nothing, because the Paterson-Wegman
algorithm does not do a find. Instead, whenever the algorithm needs to process an equivalence class, it arbitrarily selects a function node. The selected node is connected to all the other nodes in the equivalence class. The selected node is made the representative of the class. If there are other function nodes in the equivalence class, they are eliminated by equating corresponding function arguments (possibly creating or augmenting other equivalence classes). The result is an equivalence class containing at most one function, making it a substitution. The Paterson and Wegman algorithm is superior because instead of giving a name to each equivalence class (i.e., selecting a representative) at the outset and then having to change the name upon receipt of new information, the algorithm waits until an equivalence class is completely built, then exercises the freedom of being able to select any one function as the name.

3.14. More on Simplifying Compound Equivalence Classes

It is necessary to consider in more detail how a compound equivalence class can be reduced to a simple equivalence class, for we must have a simple class in order to construct a unifier, but we must allow compound classes in order to construct equivalence classes efficiently.

Assume a set of bindings comprising a single equivalence class with more than one function. For example, unify $h(f(X), W, f(Y))$ and $h(W, Z, Z)$. The set of bindings from
udiff is \( f(X)/W, W/Z, f(Y)/Z \) and the graph of the equivalence class is

\[ f(X) \rightarrow W \rightarrow Z \rightarrow f(Y) \]

A compound equivalence class can be made simple by finding a unifier for all the functions in the compound class, then applying that unifier. The result is two (or more) equivalence classes that can be converted into substitutions and applied separately.

Continuing the example, \( f(X) \) and \( f(Y) \) can be unified by the substitution \( \{X/Y\} \), which has a corresponding equivalence class with the graph

\[ X \rightarrow Y \]

3.15. A Generalized Approach to Unification

**Definition 3.7.** Let \( Q \) be an equivalence class of terms. \( Q \) is called a **g-substitution** and is defined to be the t-substitution \( [q]^+ \) where \( q \) is any member of \( Q \).

A g-substitution derived from some equivalence class \( Q \) is computed non-deterministically -- any member of \( Q \) can be selected as the representative to be propagated. If \( Q \) is trivial, then every g-substitution is also a substitution. If \( Q \) is simple, then only the g-substitution based on \( Q \)'s single function term is also a substitution.

**Definition 3.8.** Let \( T \) be a set of terms and \( Q \) an equivalence class of terms. A **g-instance** \( T*Q \) of \( T \) by \( Q \) is the t-instance \( T*[q]^+ \), where \( q \) is any term in \( Q \).

In considering the udiff algorithm, it appeared that a
possible approach to unification would be to take the output of udiff -- an "undisciplined" set of bindings that already resembles an mgu -- and to gradually transform that set of bindings into first, a substitution, and second, a unifier. Now we make the notion more general and formalize it.

It is now clear that a set of bindings can be generated from a simple or trivial equivalence class, and that a set of terms can be interpreted to be an equivalence class. So we can elevate the discussion to one concerning just equivalence classes and sets of terms, considering sets of bindings to be a special case.

It is also the case that a compound equivalence class can be made simple by unifying the functions in it. So the thought occurs that to unify a set of terms T, we can interpret T to be an equivalence class. Then if the equivalence class is compound, we can find one or more substitutions that make it simple. This simplified equivalence class, along with the substitutions that make it simple, can then be combined in some (as yet unspecified way) to define the unifier for T.

**Example 3.6.** Let \( T = \{f(X), f(Y), Z\} \) be a set of terms to be unified. Let \( Q = T \) be an equivalence class. \( Q \) is compound; it has two functions. To make it simple, unify \( f(X) \) and \( f(Y) \) by using the equivalence class \( R_X = \{X, Y\} \). Then \( TR_X = \{(f(X), f(Y), Z)\{X, Y\} = \{f(X), Z\} \) (where \( [R_X] \) is taken to be \( [X]^+ \)). The unifier for \( T \) is the combination of
components $R_X(\text{TR}_X) = \{X,Y\}(f(X),Z)$. Interpreted as substitutions, this becomes the composition of substitutions $(X/Y)(f(X)/Z) = (X/Y, f(X)/Z)$, or the unifier for $T$.

In many (most) cases, it will be necessary to deal with a set of equivalence classes instead of just one. We extend our definitions to cover this.

**Definition 3.9.** Let $E = \{Q_1, \ldots, Q_n\}$ be a set of equivalence classes, $S$ a substitution, $\$\text{ a t-substitution,}\$ and $\$Q\text{ a q-substitution. The instance } E\text{ by } S \text{ is } \{Q_1S, \ldots, Q_nS\}$. The t-instance $E\$\text{ of } E \text{ by } \$ \text{ is } \{Q_1\$, \ldots, Q_n\$. The instance $E\$\text{ of } E \text{ by } \$ \text{ is } \{Q_1\$, \ldots, Q_n\$. S is a unifier for $E$ if $S$ is a unifier for all of $Q_1, \ldots, Q_n$. Similarly for $\$\text{ and } \$Q$.

We envision an algorithm that treats a set of terms $T$ as an equivalence class. It finds a set of substitutions that make $T$ simple. The set of substitutions can be composed to yield a new, single substitution. The following theorem shows that if we can find such a single, composed substitution, then we can define a unifier for $T$.

**Theorem 3.1.** Let $T$ be a set of terms and $E=T$ an equivalence class of terms. If $S$ is a substitution such that $ES$ is simple, then $S(ES)$ is a unifier for $T$.

**Proof:**

1) $|T * E| = 1$ - A set of terms is unified when it is its own q-substitution.
2) \(|TS \ast ES| = 1\) - Since E=T, then TR = ER. TR is a set of terms, and ER is its own q-substitution. So by the same reasoning as in step 1, TR unifies itself.

3) \(|(TS)(ES)| = 1\) - If \(|T\ast S| = 1\), then \(|T\ast| = 1\), by the following:

Lemma. Let T be a set of terms and S = \(\{Q_1, \ldots, Q_n\}\) a set of q-substitutions. If S is simple and T = Union(Qi in S) (i.e., the terms in T and the terms in S are the same), then \(|T\ast S| = 1\) implies \(|TS| = 1\).

Proof: First we establish that T is simple. By way of contradiction, assume that T is not simple. Then T has at least two functions f1 and f2. To equate f1 and f2 via a t-substitution, S must have an equivalence class that includes both f1 and f2. But this means that S is not simple -- a contradiction. Thus, T is simple.

Since T and S have exactly the same terms, S has exactly one replacement for each term in T. T is simple, so it has one function and some finite number of variables. Thus S, when viewed as a substitution, can equate the terms in T by replacing each of the variables. When equated, the terms in T are reduced to a singleton, i.e., \(|TS| = 1\).

By associativity, TR(ER) = T(R(ER)). Since this is a singleton, R(ER) is a unifier for T. [End of proof].
These concepts are combined to create the following unification algorithm.

Algorithm 3.4. unify.
Let $E$ be a set of equivalence classes.
\[\text{unify}(t_1,t_2) \{\]
\[E = \{ \{t_1,t_2\} \};\]
\[\text{while} (E \text{ has some compound equivalence class } Q) \{\]
\[F = \text{any function in } Q;\]
\[\text{for (each term } T \text{ equivalent to } F) \{\]
\[\text{if (} T \text{ is a function } \neq F) \{\]
\[\text{delete } T \text{ from } Q;\]
\[\text{for (} i=1..\text{arity}(F)\} \]
\[\text{equate}(\text{arg } i \text{ of } F, \text{ arg } i \text{ of } T);\]
\}
\[\text{if (} Q \text{ is a singleton) }\]
\[\text{delete } Q \text{ from } E;\]
\}

\[\text{equate}(t_1,t_2) \{\]
\[\text{Assume } E = \{Q_1,\ldots,Q_n\};\]
\[\text{if (} t_1 \text{ is in some } Q_i) \]
\[\text{if (} t_2 \text{ is in some } Q_j) \]
\[\text{if (} Q_i \neq Q_j) \{\]
\[\text{delete } Q_j \text{ from } E;\]
\[\text{merge } Q_j \text{ into } Q_i;\]
\}
\[\text{else }\]
\[\text{add } t_2 \text{ to } Q_i;\]
\[\text{else if (} t_2 \text{ is in some } Q_i) \]
\[\text{add } t_1 \text{ to } Q_i;\]
\[\text{else }\]
\[\text{add } \{t_1,t_2\} \text{ to } E;\]
\}

The use of such an algorithm results in a set of trivial and/or simple equivalence classes, which can be interpreted as a composition of substitutions. Since the composition of substitutions is not commutative, the order of the substitutions is significant.

In fact, a partial order on the equivalence classes must be enforced. The ordering relation is:
Ordering Relation on Equivalence Classes. Let $Q_1$, $Q_2$, and $Q_3$ be equivalence classes. If any variable term in $Q_1$ occurs in any function in $Q_2$, then $Q_1 < Q_2$. If $Q_1 < Q_2$ and $Q_2 < Q_3$, then $Q_1 < Q_3$. (Note: Normally, a partial ordering is taken to be defined by a relation that is transitive, antisymmetric, and reflexive. However, here I am using an alternative, but equivalent, definition whereby the ordering relation is transitive, asymmetric, and irreflexive. See Knuth (1973) pp. 258-259).

If $Q_1 < Q_1$, or $Q_1 < \ldots < Q_1$, then we say that a cycle exists among the equivalence classes. This implies a nonunifiability.

3.16. Paterson & Wegman's Linearity

Unification can be achieved by constructing a set of simple equivalence classes having no cycles.

In the Paterson & Wegman algorithm, every term and subterm is allocated a node. Whenever an equivalence between two terms is detected, a pointer from each node to the other is set. As a term may be equivalent to any number of other terms, its corresponding node may have any number of pointers. This insures that an equivalence can be recorded in constant time simply by setting pointers.

At the beginning of a unification of $t_1$ and $t_2$, only one equivalence is known: that between $t_1$ and $t_2$. So only one equivalence class is represented, with pointers between $t_1$ and $t_2$. If at least one of $t_1$ and $t_2$ is a variable, then
the unification is finished; all the equivalence classes (just one in this case) are non-compound. But if both \( t_1 \) and \( t_2 \) are functions, then the equivalences between the respective function arguments are recorded. This has the logical effect of either creating a new equivalence class, augmenting an existing equivalence class, merging two equivalence classes, or recording a redundant relation. In any case, the operation is done in constant time.

At no point is it necessary to "look up" an equivalence class; there is no find. A union of two equivalence classes is accomplished as a side-effect of equating two terms that happen already to belong to different equivalence classes.

Having established equivalences, the algorithm could choose to identify compound equivalence classes and simplify them. There are, however, two problems: 1) The equivalence classes are not identifiable as objects, so there is no easy way to cycle through them looking for compound equivalence classes; and 2) Redundant computations can be performed if compound classes are selected in the "wrong" order.

The latter problem arises because the algorithm is constantly adding to and deleting the links between nodes that define equivalence classes. If a link between two functions in a compound class is erased too early, it may be restored by a later operation and have to be erased again.

The algorithm sidesteps the problem of identifying compound equivalence classes. It merely selects a function
node -- arbitrarily. This function node serves as a "handle" for an equivalence class, as it is possible to find all other members of the class by following pointers from the node. If the equivalence class happens to be simple, nothing is lost; bindings are printed for all the variable terms in the class. On the other hand, if the class is compound, this fact will be discovered as the links from the "handle" are traversed. When another function node in the equivalence class is encountered, new equivalences between the arguments of this node and the "handle" are recorded and the second function node is deleted.

This trick neatly allows an equivalence class to be selected without having any explicit structure that identifies equivalence classes, but it does nothing to help solve the problem that equivalence classes must be processed in the "right" order. Fortunately, when traversing an equivalence class it is possible to detect very easily that some other equivalence class must be processed first, and to then do so before any damage is done. If some variable in the current class occurs in a function of some other class, the work on the current class must be suspended while the other class is processed.

The check to see if a variable occurs in some function is not expensive; it can be done in constant time. The price paid is in having "parent" pointers for each node. A parent pointer points to the function of which a node is an
argument. It can be established as the nodes are allocated before the algorithm begins, but it implies a "fatter" data structure. (The use of "parent" pointers in the Paterson & Wegman algorithm has been much maligned, especially by authors of unification algorithms having spare data structures).

3.17. Concrete Data Structures

The Paterson & Wegman algorithm assumes that some "node" exists for each term and subterm in the terms being unified. For example, if \( f(g(X)) \) and \( Y \) are being unified, there will be nodes for \( f(g(X)) \), \( g(X) \), \( X \), and \( Y \). If the algorithm is used within a programming system such as Prolog, the interpreter will likely have created a nearly adequate set of nodes while parsing the input terms. This set of nodes can be naturally arranged to form a directed acyclic graph (dag), a data structure that is assumed by the Paterson & Wegman algorithm.

In parsing terms, it is natural for the language interpreter to create a node for each variable or function, recording the name of the term, its type (variable or function), and a list of its arguments if it is a function. Abstractly, the node resembles:

```
| NAME | TYPE (FCN OR VAR) | LIST OF ARGUMENTS (IF FCN) |
```

The Paterson & Wegman algorithm requires that such a node be augmented by 1) a list of parent pointers, 2) a list
of equated terms, and 3) an indicator to mark the node during processing. Abstractly, the augmentation resembles:

```
| PARENT POINTERS | LIST OF EQUIVALENCES | MARKER |
```

In each of the lists (arguments, parents, and equivalences), values can be represented by the addresses of the nodes.

The parent pointers are static and are capable of being established by the interpreter as the terms are being parsed.

The list of equivalences cannot be limited in length, since, as has been discussed, the algorithm's efficiency in maintaining equivalence classes relies on the ability to add a new equivalence to a node at any time.

3.18. Practical Efficiency

Although the Paterson & Wegman algorithm is truly linear, it has been the object of much criticism due to its data structures. The fact that it uses dags has been viewed as a shortcoming, even though such a structure may be made available "for free" by the language processor using the unification algorithm. Some authors have claimed that getting a problem into dag form requires an unpalatable preprocessing step.

Logic programming problems tend to require a large number of unifications over small terms (de Champeaux 1986; Escalada-Imaz and Ghallab 1988). Unification under such
conditions is dominated by setup time, as the terms are short enough that the effects of, say, quadratic worst-case behavior are not felt. If such is actually the case, then unification algorithms with the simplest structures that afford moderate $O(n^2)$ worst-case behavior may provide superior average-case performance. The algorithm of Escalada-Imaz and Ghallab (1988) is designed with such a claim in mind.
REFERENCE LIST


Huet, G. 1976. Resolution d'equations dans les languages d'ordre 1,2,...,w. These d'etat. Specialite Mathematiques, Universite Paris VII.


Li, Guang, Chao-Chih Yang, and Peter A. Ng. 1988. An improved unification algorithm for finding a most general unifier of simple expressions. Unpublished manuscript. C.-C. Yang, Department of Computer Science, University of North Texas, Denton, Texas.


