THE COMPLEXITY OF UNIFICATION

Lewis Denver Baxter
Department of Computer Science
University of Waterloo

CS-77-25

September 1977

THE COMPLEXITY OF UNIFICATION

Abstract

by

LEWIS DENVER BAXTER

It is shown that the problems arising from the unification of expressions yield a rich class of complexities. The unification problem is to determine whether or not there exists a substitution of variables by expressions which, applied to two given expressions, makes them equal. The measure of complexity is the time taken to solve the unification problem.

An algorithm which solves the first-order unification problem is presented and shown to have a complexity which is linear times a very slowly-growing function related to the inverse of Ackermann's function. This is an improvement upon previous algorithms whose complexity was exponential or quadratic. Our algorithm is composed of a transformational stage followed by a sorting stage. The transformational stage manipulates a partition of expressions. Expressions are represented by trees and the partition is represented by a forest of trees. By performing certain operations on this partition using the technique of path-compression on balanced trees, we obtain an efficient transformational stage whose complexity is linear times a very slowly-growing function. The sorting stage attempts to topologically sort a directed graph which is induced by the output partition. This stage has linear complexity.

We prove that two unification problems: first-order subsumption and second-order instantiation have complexities equivalent to that of the well-known problem of recognizing tautologies. In each case, we construct, from an instance of the closely related problem of determining if a formula in clause form is satisfiable, an instance of the unification problem.

At the upper limit of the class of complexities, we show that a version of the third-order unification problem, in which the number of arguments in expressions is restricted to be at most two, is undecidable. We use a construction involving the undecidable Hilbert's Tenth Problem.

Acknowledgements

I especially wish to sincerely thank my supervisor, Tom Pietrzykowski, for his encouragement, help and criticism throughout the preparation of this thesis.

Also, I wish to thank Ian Munro, Maarten van Emden and Dave Younger for their thorough reading of this work, and, in particular, Stephen Cook, my external examiner, for his meticulous scrutiny of and helpful criticisms about this thesis.

Finally, a loving thank-you to Eva whose concern provided me with a happy environment in which to complete this thesis.

T A B L E   O F   C O N T E N T S

INTRODUCTION

We examine the problem of unifying expressions from a computational complexity viewpoint. The unification problem is to determine, given two expressions e_1 and e_2 containing variables, whether there exists a substitution of these variables by expressions which, applied to e_1 and e_2, makes them equal. We are interested in the number of steps (time) taken to solve various classes of unification problems.

The unification problem originally arose from theorem-proving which is concerned with the automation of logical inferences. In simple logic we have the valid syllogism: infer from the premisses "A implies B" and "B implies C" the conclusion "A implies C." This idea can be generalized to first-order logic, in which we allow variables to occur in expressions. For example, we can infer from the statements, "for all x and y, A(y,F(x),x) implies B(G(x),y)" and "for all u and v, B(u,F(v)) implies C(F(F(v)),G(u))" the result, "for all x, v, A(F(v),F(x),x) implies C(F(F(v)),G(G(x)))." The reason is because the logical expressions, B(G(x),y) and B(u,F(v)) can be made equal by substituting G(x) for u and F(v) for y. Applying this substitution to A(y,F(x),x) and to C(F(F(v)),G(u)) we obtain the correct conclusion. We thus see that unification can be used to elegantly combine substitution and syllogism to form a rule of inference known as "resolution" [27]. Unification can also be used in combining substitution with the rule of replacing equals by equals, to obtain an inference rule known as "paramodulation" [26].

Many useful computer programs use unification in order to make logical inferences. Examples of these are in program analysis/synthesis, deductive question-answering systems and robot planning programs [5]. Also some new programming languages of artificial intelligence, such as PROLOG [32], use unification as a mechanism to invoke subroutine calls by pattern-matching.

Historically, in 1921, unification was mentioned under the name of "L.C.M." [25] (in analogy to the arithmetic process of least common multiple) by Post, who was interested in the matching problems associated with generalizations of classical axiomatizations of propositional calculus. In 1955, Newell, Shaw and Simon [21] designed a computer program, "The Logic Theorist" which attempted to heuristically prove theorems of propositional calculus. This program used a matching process very similar to unification. Unification gained its official title in 1965 when Robinson [27] presented a unification algorithm in order to mechanize his rule of inference, resolution. This algorithm was discovered independently by Guard, under the name of "matching" [12]. Their algorithms either reported that unification was impossible or output the most general unifying substitution. Their correctness was proved.

The above algorithms solved the unification problem for first-order expressions, in which all the variables were of first-order, that is, ranged over individuals of some domain. In order to incorporate equality, induction, the axiom of choice and other higher-order concepts in theorem-proving, we must allow our variables to also range over functions. That is, we must allow second-order expressions. The corresponding second-order unification problem is complicated by the fact that there may be several or even an infinite number of independent unifiers. Gould [11] first recognized these difficulties in attempting to produce the "most common instance" of two expressions, a problem which is not as general as the unification problem. Pietrzykowski gave a procedure to enumerate all the unifiers of two second-order expressions [23]. This was later generalized to ω-order expressions by Pietrzykowski and Jensen [24]. Huet gave a semi-decision procedure to determine the existence of a unifier for ω-order expressions [14].

We now discuss the complexity aspects of these unification problems. The need to analyze the efficiency of the unification and related problems was first seen by Allen and Luckham [2]:

> "We do not have any theoretical study to tell us the most efficent way to implement the basic operations at the lowest level of the [theorem-proving] program (e.g., the tests for unification, alphabetic variants and subsumption)."

The purpose of this thesis is to provide such a study.

Robinson's original first-order unification algorithm was abstract, in that no representation was specified for expressions. However, implicit in the accompanying examples was that expressions are represented by a string-like data structure and that substitutions are explicitly applied by physically relocating and inserting new strings. Consequently, his algorithm requires exponential time, since expressions of exponential length, in relation to the length of the input expressions to be unified, may appear during unification.

In 1970, Robinson motivated the search for efficient first-order unification algorithms by presenting a new algorithm, in which expressions were represented by a tree-like data structure [28]. Quoting from his paper:

"The author thinks the [unification] program is very close to maximal efficiency, and offers it as a challenge."

Unfortunately, this algorithm also required exponential time. A paradoxical situation arose: an expression, which requires linear space to store in the form of a tree, in which common subexpressions are represented by the same subtree, may require exponential time to determine if a certain variable occurs in the expression. Nevertheless, the idea of using trees to represent expressions and of manipulating pointers to these trees to perform substitutions laid the foundations for more efficient algorithms. In 1975 Venturini-Zilli remedied the paradox by using a simple marking scheme, and to speed up the checking of whether a variable occurs in an expression, she maintained a list of variables associated with each subexpression [33]. She proved that the algorithm requires quadratic time.

In 1973, Baxter [3] presented an algorithm which avoided the previous source of inefficiency: the continual checking to determine if a given variable occurs in an expression. This algorithm consisted of two stages: a transformational stage followed by a sorting stage. Some aspects of the former stage are similar to those in a higher-order procedure presented by Huet [14]. The basic idea was elaborated upon in a later paper [4] in which a better choice of data structures gave an almost linear time bound. From personal communications, it appears that a similar idea was independently investigated by Huet [15] and Robinson [29], although their results have not been published.

We present this algorithm here. During the transformational stage of this algorithm, we manipulate a partition of expressions; classes in this partition are merged and the class containing a given expression is found. By using the well-known method of representing a partition by a forest of trees and by performing "path compression" on these balanced trees, we are able to show that the time required by the transformational stage is practically linear. That is, the time required is of order nG(n) where n is the length of the input and G is a very slowly-growing function related to the inverse of Ackermann's function. The partition which is output by the transformational stage is unifiable if and only if the input set of expressions is unifiable. Furthermore, it is easy to determine if the output partition can be unified - this is done by the sorting stage. During the sorting stage a directed graph is constructed from the output partition. If this graph contains a circuit, then unification fails, otherwise it can be topologically sorted to give the most general unifier. The time required by the sorting stage is linear and is based upon a well-known topological sorting algorithm [18].

A problem related to unification is the subsumption problem. Given two sets of expressions $\{e_1,\ldots,e_m\}$ and $\{E_1,\ldots,E_n\}$, in which no variables occur in any E_i, the subsumption problem is to determine whether there exists a substitution σ such that $\{\sigma(e_1),\ldots,\sigma(e_m)\}$ is included in $\{E_1,\ldots,E_n\}$. This problem arises from theorem-proving, as it gives a condition for eliminating certain lemmas found during a search for a particular theorem. In his pioneer paper, Robinson introduces the subsumption problem and justifies the use of the elimination condition [27].

We prove that this problem is NP-complete in the sense of Cook and Karp [8,17] . That is, the subsumption problem is as computationally complex as the problem of recognizing tautologies - which is of exponential difficulty if the famous P=NP conjecture holds. Our main result is the construction, from any formula of propositional calculus (expressed in clause form), of an instance of the subsumption problem. Thus, an efficient algorithm to solve the subsumption problem implies an efficient algorithm to solve the tautology problem.

Another unification problem which we prove to be NP-complete is the second-order instantiation problem. Given two second-order expressions e and E, in which E contains no variables, this problem is to determine if e and E can be unified. Our main result also uses a construction of an instance of the instantiation problem from a formula in propositional calculus. We note that the decidability of the general second-order unification problem remains open, even if the expressions can have only one argument. Some research into this monadic case (string unification) is presently in progress [30,34]. An obvious corollary of our work is that if the second-order unification problem is decidable, its complexity must be no less than that of recognizing tautologies.

The third-order unification problem is known to be undecidable [13,19] by a construction involving the undecidable Post Correspondence Problem. We present a new proof of this, which improves upon the previous results by restricting the degree (number of argments) that the third-order expressions may have. Whereas the earlier results made no such restriction, we prove the stronger result that the third-order second-degree unification problem is undecidable. Our construction in this proof uses the recently proved undecidability of Hilbert's Tenth Problem. We construct, from a Diophantine equation an instance of our unification problem with the property that the equation has a solution if and only if the constructed expressions can be unified.

In Chapter 1 we define our language of expressions and substitutions in order to define the unification problem. In Chapter 2 we present our practically linear first-order unification algorithm, prove its correctness and analyze its complexity. In Chapter 3 we show that the first-order subsumption problem is polynomially-complete and in Chapter 4 we do this for the second-order instantiation problem. In Chapter 5 we prove that the third-order second-degree unification problem is undecidable.

Just as this thesis was being completed, it was learnt that a linear first-order unification algorithm was discovered by Paterson and Wegman [22]. This algorithm combines our transformational and sorting stages. In the appendix we briefly outline this linear algorithm, explain why it is linear and compare an incremental property of our algorithm with that of the linear one. That is, we ask:

Having unified a set of expressions, how easy is it to unify the union of this set with an additional pair of expressions?

In the conclusion we outline possible directions for further research into the complexity of other unification problems.

CHAPTER 1

LANGUAGE

In this chapter we will define the notions of types, expressions and substitutions. This consolidates the notions found in earlier papers [6,14,24]. Finally, we will define the unification problem.

1.1 TYPES.

Informally, a function has type $(t_1,\ldots,t_n \rightarrow t_0)$ if it maps n-tuples of entities of type $t_1,\ldots,t_n$ into entities of type t_0. Thus the unique type of a symbol or expression can be regarded as a metalinguistic variable which indicates its position in a functional hierarchy.

Formally, a set T whose elements are called types is defined inductively from a fixed set T_0 of basic types by:

(1) $t \in T_0$ implies $t \in T$;

(2) $t_1,\ldots,t_n \in T$ $(n \geq 1)$ and $t_0 \in T_0$ imply $(t_1,\ldots,t_n \rightarrow t_0) \in T$.

Note that we exclude, for example, the type ((real→real)→(real→real)), which might be considered to be the type of the derivative operation in calculus. This is easily overcome by making the derivative have two arguments:

$$\text{DERIVATIVE}(f,x) = \lim_{\varepsilon \rightarrow 0} (f(x+\varepsilon)-f(x))/\varepsilon$$

so that

$$\text{type[DERIVATIVE]} = ((\text{real}\rightarrow\text{real}),\text{real}\rightarrow\text{real}).$$

1.1.1. Order.

We define the order of a type to reflect its "depth of nesting", that is, as a mapping from the set of types T to the set of integers by:

$$\text{order}[t] = \begin{cases} 1 & \text{if } t \in T_0 \\ 1 + \max\{\text{order}[t_i] \mid 1 \le i \le n\} & \text{if } t = (t_1, \ldots, t_n \to t_0) \end{cases}$$

1.1.2. Degree. We also define the degree of a type as a mapping from T to the set of integers by:

$$\text{degree}[t] = \begin{cases} 0 & \text{if } t \in T_0 \\ n & \text{if } t = (t_1, \ldots, t_n \to t_0) \end{cases}$$

We will later extend order and degree to the symbols which comprise our expressions. Informally, the order of a symbol indicates whether the symbol is an individual, function, functional,... and the degree of a symbol indicates the number of arguments it has.

1.2. EXPRESSIONS.

Our expressions are based upon the normal form of λ-calculus expressions of Church [7], where the formal arguments are given explicitly and also prefix notation is used.

Our presentation differs from that of Church's in the following way. Church composes expressions from atoms, applications and abstractions and subsequently shows, using the Church-Rosser theorem, that each expression has a normal form. The application of a substitution to an expression, e, can then be defined as the normal form of a certain expression constructed from e by abstraction and application. In contrast, we define an expression directly from the normal form, thus avoiding the notions of application and abstraction. However, our definition of substitution is now more complex. Our presentation is in fact self-contained, but we will occasionally relate our expressions with those of Church.

1.2.1. Symbols. Expressions are constructed from *symbols* which are either *variables* or *constants* and from the following five punctuation marks: λ , . ( ). Associated with each symbol is some type. For each type we have a denumerable set of variables of that type. We also have a set of constants of arbitrary given types.

Symbols are constructed from strings of letters: we use lower-case letters for variables and upper-case letters for constants.

We use s as a syntactical variable which ranges over symbols. As with all our syntactical variables we often add subscripts and/or superscripts, hence s_7 and s'' also denote symbols. We use u and v to denote variables and f to denote constants.

The type of a symbol s is written type[s].

1.2.2. Order and Degree. We extend order and degree to symbols by the definitions

order[s] = order[type[s]]

degree[s] = degree[type[s]].

1.2.3. Definition of expression. We now define our expressions and their types by induction on the number of occurrences of symbols.

An _expression_ is of the form

$$\lambda u_1,\ldots,u_m.s(e_1,\ldots,e_n)$$

where $u_1,\ldots,u_m$ are $m\geq 0$ distinct variables which denote "formal arguments";

s is a symbol, with degree[s] = n, followed by the $n\geq 0$ expressions:

$e_1,\ldots,e_n$ which are the "arguments" of s

and where for some basic type t_0:

$$\text{type}[s] = \begin{cases} t_0 & \text{if } n=0 \\ (\text{type}[e_1],\ldots,\text{type}[e_n]\to t_0) & \text{if } n\geq 1. \end{cases}$$

The _type_ of the expression is then defined to be:

$$\text{type}[\lambda u_1,\ldots,u_m.s(e_1,\ldots,e_n)] = \begin{cases} t_0 & \text{if } m=0 \\ (\text{type}[u_1],\ldots,\text{type}[u_m]\to t_0) & \text{if } m\geq 1. \end{cases}$$

We denote expressions by e and E.

1.2.4. _Writing conventions_. The above definition of an expression is unambiguous; however, we will often delete punctuation marks if it is clear from the context which expression is intended. In particular, if $m=0$ we delete "λ." and if $n=0$ we delete "()". For example, F(x) abbreviates $\lambda.F(\lambda.x())$. Since we allow strings of letters for symbols, the commas are generally necessary in "$\lambda u_1,\ldots,u_m$" in order to avoid ambiguity; however we will usually delete them.

If $n\geq 1$ then s is not an expression, but $\lambda u_1\ldots u_n.s(u_1,\ldots,u_n)$ is, where the variables u_i have appropriate types. We often abbreviate the latter by simply writing it as s. For example, if

type[x] = type[y] = s,

type[h] = (s→ t),

and type[g] = ((s→ t), s→ t)

then λhx.g(λy.h(y),x) can be abbreviated to λ hx.g(h,x) which, in turn, can be abbreviated to g.

1.2.5. Example. We will illustrate how an expression and its type can be constructed from symbols by the example:

λfn.SUM(ONE,n, λj.f(SQUARE(j))).

This is the formal counterpart of the mathematical expression:

$$\sum_{j=1}^{n} f(j^2).$$

Our symbols are the variables: f, n and j and the constants: SUM, ONE and SQUARE. The types of these symbols follow, where the basic set of types is {integer, real}.

type[j] = type[n] = type[ONE] = integer;

type[f] = (integer→real);

type[SQUARE] = (integer→integer);

type[SUM] = (integer,integer,(integer→real)→real).

Now, j is an expression with type[j] = integer and since type[SQUARE] = (integer→integer), SQUARE(j) is an expression of type: integer. Hence, since type[f] = (integer→real), λj.f(SQUARE(j)) is an expression of type: (type[j]→real), that is, (integer→real). ONE is an expression of type: integer, n is an expression with type[n] =

integer, hence, by considering the type of SUM, our example is indeed an expression whose type is ((integer→real),integer→real).

1.2.6. Length. The appropriate measure of the size of our expressions, length, is inductively defined from symbols by:

$$\text{length}[\lambda u_1 \ldots u_m . s(e_1,\ldots,e_n)] = 1 + \sum_{i=1}^{n} \text{length}[e_i].$$

(Note that when n=0 the summation is zero.)

1.2.7. Subexpression. We will now define the notion of subexpression by

(1) $s(e_1,\ldots,e_n)$, e_1, ... ,e_n are all subexpressions of $\lambda u_1 \ldots u_m . s(e_1,\ldots,e_n)$;

(2) e is a subexpression of e;

(3) e_1 is a subexpression of e_2 and e_2 is a subexpression of e_3 imply that e_1 is a subexpression of e_3.

1.2.8. Free and bound occurrences. We now use the above definition in order to precisely define the concepts of free and bound occurrences of variables in expressions.

An occurrence of the variable v is bound in the expression E if v occurs in a subexpression of E of the form $\lambda v_1 \ldots v_n . e$ where v is some v_i for i=1,...,n; otherwise the occurrence of v is free in E. We say that v is free (bound) in E if some occurrence of v is free (bound) in E. (Note that the variable x is both free and bound in the expression $F(x,\lambda x.g(x))$: the first occurrence of x is free and the second and third occurrences of x are bound in the expression.)

A pair of occurrences of a variable is linked in an expression E if these occurrences are bound in E and if for each subexpression e of E either both occurrences are bound in e or neither of them is.

1.2.9. Equality. Two expressions e' and e" of the same type are equal, written e'=e", if e" results from e' by changing only some bound occurrences of variables, providing that if a pair of occurrences is linked in e' then the corresponding pair is also linked in e" and vice versa, and that the type of corresponding variables remains the same.

As expected, equality between expressions is an equivalence relation on expressions.

For example,

$$\lambda xy.H(x,\lambda z.z,t,\lambda x.x,z)$$
$$\lambda uv.H(u,\lambda w.w,t,\lambda u.u,z)$$
$$\lambda uv.H(u,\lambda w.w,t,\lambda s.s,z)$$
$$\lambda uv.H(u,\ u.u,t,\lambda u.u,z)$$

are all equal expressions.

1.3. SUBSTITUTION.

A substitution is a finite set of ordered pairs

$$\{\langle v_1,e_1\rangle,\ldots,\langle v_n,e_n\rangle\}$$

where the v_i's are distinct variables and the e_i's are expressions such that for all $i=1,\ldots,n$ type[v_i] = type[e_i].

We also impose the condition that v_i does not abbreviate e_i (in the sense of section 1.2.4.). We write this substitution suggestively as:

$$\{v_1 \leftarrow e_1, \ldots, v_n \leftarrow e_n\}$$

and say that it pertains to the variables $v_1, \ldots, v_n$.

We denote substitutions by σ; the empty substitution is denoted by ε.

Informally, a substitution is applied to an expression by simultaneously replacing the variables, to which the substitution pertains, by the associated expressions. When doing so, we must be careful that no conflict of bound variables occurs. To ensure this, we choose a suitable expression which is equal to the original, but in which no conflicts arise.

1.3.1. Application. We now inductively define the application of a substitution σ to an expression e written $\sigma(e)$.

Let V be the set of variables which pertains to or which occurs free in some e^* where $v^* \leftarrow e^*$ belongs to σ, for some v^*. In order to avoid a conflict of bound variables, choose an expression e' which is equal to e, but in which no variable in V is bound:

$$e' = \lambda u_1 \ldots u_m . s(e_1, \ldots, e_n).$$

If the symbol s is some u_i $(i=1,\ldots,m)$ or if s is a constant or if σ does not pertain to s then define

$$\sigma(e) = \lambda u_1 \ldots u_m . s(\sigma(e_1), \ldots, \sigma(e_n)).$$

Otherwise, let

$$s \leftarrow \lambda v_1 \ldots v_n . E$$

belong to σ and define

$$\sigma(e) = \lambda u_1 \ldots u_m . \sigma'(E)$$

where

$$\sigma' = \{v_1 \leftarrow \sigma(e_1), \ldots, v_n \leftarrow \sigma(e_n)\}.$$

(By $\sigma'(E)$ we strictly mean $\sigma'(\lambda.E)$.)

Of course the application of the empty substitution to an expression, which is the basis of the inductive definition, is defined by:

$$\varepsilon(e) = e.$$

Note that substitutions preserve type, that is,

$$\text{type}[\sigma(e)] = \text{type}[e].$$

1.3.2. Well defined. It is not immediately obvious that the definition of application is well defined in the sense that the above method will always terminate. However, we can prove using generalized induction that it will. To do this we introduce a complexity measure for applications by:

$$\text{comp}[\sigma, e] = (\text{order}[\sigma], \text{length}[e])$$

where order is defined on substitutions by:

$$\text{order}[\sigma] = \max\{\text{order}[v] \mid \sigma \text{ pertains to } v\}.$$

To compute $\sigma(e)$ we must recursively compute $\sigma(e_1), \ldots, \sigma(e_n)$ and possibly $\sigma'(E)$. We will show that the complexity of these latter applications is less than that of the application $\sigma(e)$, hence by a generalized induction argument, the method will indeed terminate.

We order complexities by well-ordering pairs of natural numbers in the usual way:

$$(i',i'') < (j',j'') \quad \text{iff} \quad i'<j' \text{ or } (i'=j' \text{ and } i''<j'').$$

Using this ordering,

$$\text{comp}[\sigma,e_i] < \text{comp}[\sigma,e] \quad \text{for } i=1,\ldots,n$$

since

$$\text{length}[e_i] < \text{length}[e'] = \text{length}[e].$$

Also,

$$\text{comp}[\sigma',E] < \text{comp}[\sigma,e]$$

since

$$\begin{aligned} \text{order}[\sigma'] &= \max\{\text{order}[v_i] \mid 1\le i\le n\} \\ &= \text{order}[s] - 1 \\ &< \text{order}[s] \\ &\le \text{order}[\sigma]. \end{aligned}$$

1.3.3. Example. We illustrate the application of a substitution σ to an expression e by taking

$$e = \lambda xyz.f(g(A),f(g(x),g(y),x),H(z))$$

and

$$\sigma = \{f\leftarrow\lambda xyz.y,\ g\leftarrow\lambda u.F(u,x,w),\ z\leftarrow A\}.$$

Since σ pertains to f, g and z and since x and w are free in $\lambda u.F(u,x,w)$, V equals $\{f,g,z,x,w\}$. We choose

$$e' = \lambda x'yz'.f(g(A),f(g(x'),g(y),x'),H(z'))$$

because e=e' and no variable of V is bound in e'. Therefore,

$$\begin{aligned} \sigma(e) \quad &= \lambda x'yz'.\{x\leftarrow\sigma(g(A)),y\leftarrow\sigma(f(g(x'),g(y),x')), \\ &\qquad z\leftarrow\sigma(H(z'))\}(y) \end{aligned}$$

(since σ pertains to f)

$= \lambda x'yz'.\ \varepsilon(\sigma(f(g(x'),g(y),x')))$

(since the appropriate σ' is ε)

$= \lambda x'yz'.\ \sigma(f(g(x'),g(y),x'))$

(by the basis of the inductive definition).

In general, if $v \leftarrow e$ belongs to σ then $\sigma(v)$ equals e, therefore

$\sigma(e) \quad = \lambda x'yz'.\{x\leftarrow\sigma(g(x')),\ y\leftarrow\sigma(g(y)),\ z\leftarrow x'\}(y)$

(since σ pertains to f)

$= \lambda x'yz'.\ \sigma(g(y))$

$= \lambda x'yz'.\{u\leftarrow\sigma(y)\}(F(u,x,w))$

(since σ pertains to g)

$= \lambda x'yz'.F(\{u\leftarrow\sigma(y)\}(u),\{u\leftarrow\sigma(y)\}(x),\{u\leftarrow\sigma(y)\}(w))$

(since F is a constant)

$= \lambda x'yz'.F(\sigma(y),x,w)$

(since $\{u\leftarrow\sigma(y)\}$ pertains only to u)

$= \lambda x'yz'.F(y,x,w)$

(since σ does not pertain to y).

1.4 UNIFICATION.

We will define the concept of a set being unifiable, show that we can assume that this set contains only two expressions and finally, we define the unification problem.

1.4.1. Unifiable.

The substitution σ unifies a set of expressions $\{e_1,\ldots,e_n\}$ iff

$$\sigma(e_1) = \sigma(e_2) = \ldots = \sigma(e_n).$$

Since substitutions preserve types, this definition implies that $e_1,\ldots,e_n$ must all have the same type. Note that trivially, any substitution unifies the empty set and any set of one expression.

The substitution σ <u>unifies</u> a <u>collection</u> of sets of expressions iff σ unifies each set of expressions in the collection.

In the following definition, X may be a set of expressions or a collection of sets of expressions.

$$X \text{ is } \underline{\text{unifiable}} \text{ iff } \exists\sigma(\sigma \text{ unifies } X).$$

<u>1.4.2</u>. <u>Reduction</u>. We will show that the problem of determining if a set is unifiable can be reduced to the case when the set contains two expressions.

By introducing a suitable constant symbol, say F, of degree n, we can reduce the set of $n\geq 2$ expressions, $\{e_1,\ldots,e_n\}$ to the set of two expressions,

$$\{F(e_1,e_2,\ldots,e_{n-1},e_n),\ F(e_2,e_3,\ldots,e_n,e_1)\}.$$

Clearly σ unifies this set iff σ unifies $\{e_1,\ldots,e_n\}$.

A further reduction can replace the collection of sets of expressions

$$\{\ \{e_1',e_1''\},\ldots,\{e_m',e_m''\}\ \}$$

by the set of two expressions,

$$\{G(e_1',\ldots e_m'),\ G(e_1'',\ldots,e_m'')\}$$

where G is an appropriate constant of degree m. Again the reduction is valid.

1.4.3. <u>Unification Problem</u>. The unification problem is to determine, given two expressions e_1 and e_2, whether or not $\{e_1, e_2\}$ is unifiable.

We now define particular unification problems whose complexity we will be interested in.

1.4.3.1. <u>The first-order unification problem</u>. This problem is obtained by making the following restrictions on the general unification problem:

(1) the set of basic types is a singleton;

(2) the order of all variables is one;

(3) the order of all constants is one or two;

(4) all expressions have no formal arguments.

These restrictions are necessary in order to coincide with the language of first-order expressions as defined in the papers [12,27,33].

This is tantamount to defining a first-order expression inductively as either a variable, or a constant of degree n followed by n first-order expressions, and thereby abandoning the λ-notation.

1.4.3.2. <u>The n-th order unification problem</u> ($n \geq 2$). This problem is obtained by making the following restrictions on the general unification problem:

(1) the order of all variables is at most n;

(2) the order of all constants is at most n+1.

The second restriction is necessary to avoid a certain kind of instability, where during the unification a variable of order greater than n may have to be introduced. This phenomenon was noticed in [16].

<u>1.4.3.3. The n-th order p-th degree unification problem</u> ($\underline{n}\geq\underline{1}, \underline{p}\geq\underline{0}$). This problem is obtained by making the following restriction on the n-th order unification problem:

(1) the degree of all symbols is at most p.

<u>1.4.3.4. The n-th order instantiation problem</u> ($\underline{n}\geq\underline{1}$). This is the n-th order unification problem for two expressions in which one expression contains no free variables.

CHAPTER 2

A PRACTICALLY LINEAR UNIFICATION ALGORITHM

We present an algorithm for solving the first-order unification problem. We prove that it is correct and that the time taken by a suitable model of computation is of the order $nG(n)$ where n is the input size and G is a very slowly-growing function, related to the inverse of Ackermann's function. This complexity lies between $O(n)$ and $O(n\log\log\ldots\log n)$ (with any constant number of "log" factors).

2.1. INTRODUCTION.

We first present a unification algorithm which embodies the methods used by Robinson [27] and Guard [12]. It determines whether or not two first-order expressions are unifiable. (Note that the concept of composition of substitutions will be defined later.)

```
algorithm UNIFIABLE($e_1$, $e_2$):
begin
   $\sigma \leftarrow \varepsilon$; comment assign to $\sigma$ the empty substitution ;
   repeat until $\sigma(e_1) = \sigma(e_2)$ :
   begin
      let $e_1'$, $e_2'$ be occurrences of subexpressions in
      $\sigma(e_1)$, $\sigma(e_2)$ such that $e_1' \neq e_2'$ and that
      all corresponding symbols to the left of
      these occurrences are identical ;
      if $e_1'$ and $e_2'$ begin with different constants
      then return (false)
      else assume that $e_1'$ is a variable,
           otherwise swap $e_1'$ and $e_2'$ in:
           if $e_1'$ occurs in $e_2'$
           then return (false)
           else $\sigma \leftarrow \{e_1' \leftarrow e_2'\}\sigma$; comment compose $\{e_1' \leftarrow e_2'\}$ and $\sigma$;
   end ;
   return (true) ;
end.
```

This algorithm is abstract, in that we neither specify the data structures used to represent expressions and substitutions, nor the method of manipulating these data structures: of applying a substitution to expressions, of composing two substitutions and of determining if a variable occurs in an expression. However, Robinson and Guard clearly intended a simple representation using strings of symbols and simple, but costly, operations of physically manipulating such strings. It is easy to show that such a naive implementation has exponential complexity by considering the unification of the two expressions:

$$e_1 = F(x_1, x_2, \ldots, x_n)$$

and

$$e_2 = F(G(x_0, x_0), G(x_1, x_1), \ldots, G(x_{n-1}, x_{n-1})).$$

Venturini-Zilli [33] improved upon Robinson's more efficient algorithm [28] by representing expressions as trees, in which variables are shared, and by efficiently determining if a variable occurs in an expression. This improved implementation has quadratic complexity.

Unfortunately, it appears that all such implementations of this type of algorithm cannot have better than quadratic complexity. The reason for this is that the algorithm continually checks for the occurrence of a variable in an expression (in the case that one of e'_1 and e'_2 is a variable). An analogy can be made with the graphical problem of determining if a directed graph contains a circuit. If the entire graph is available, a linear algorithm can be used. However, if the edges of the graph are given to us edge by edge, and at each stage, we check for a circuit due to the additional edge, then the algorithm will have quadratic complexity. It appears that no such "on-line" method of determining if a graph contains a circuit has better than quadratic complexity. This explains our remarks made at the beginning of this paragraph.

2.2. MODEL OF COMPUTATION.

2.2.1. Random Access Machine.

We choose as our model of a computer the random access machine (RAM) [1]. The RAM consists of a read-only input tape, a write-only output tape and an infinite number of registers, each of which can hold an integer of arbitrary size. Since we assume that the unification problem is small enough to fit into the memory of a computer, and that the number of distinct symbols (represented as integers) is small enough so that each may fit into one computer word, our choice of model is justified.

2.2.2. Instructions.

The instructions of the RAM are typical of those found on any computer and include: arithmetic, comparison, logical, character, input/output, load/store, branching and indirect addressing operations. For example, if the registers are $R_0, R_1, R_2, \ldots$ then typical instructions are: "store the sum of the contents of R_i and of R_j into R_k", "if the contents of R_i is less than that of R_j then execute next the k-th instruction", "load R_i with the next input symbol", "print the contents of R_i", "store the number j in register R_i" and "load R_i with R_j where j is the content of R_k."

We will not actually present a RAM set of instructions to implement our algorithm, but will instead present a structured program which can readily be translated into an appropriate RAM program having the desired complexity.

2.2.3. Uniform cost. In assigning a complexity measure of time to a RAM, we apply the *uniform cost* criterion and assume that each instruction takes unit time. This is appropriate since we assume that the number of distinct symbols is small enough so that each may fit into one computer word.

2.2.4. Problem size. The complexity of algorithms is relative to the size of the problem. For the first-order unification problem of determining whether or not two expressions e_1 and e_2 are unifiable, we take the *size* to be the total number of occurrences of symbols in the expressions e_1 and e_2.

2.2.5. Complexity. By the complexity of an algorithm we mean the worst-case *time complexity*, that is, the maximum time taken by the RAM instructions over all input problems of size n. We will be interested only in the *asymptotic* time complexity which expresses the functional variation of complexity rather than an exact estimate. We use the "O" notation accordingly: $g(n)$ is $O(f(n))$ if there exists a constant c such that $g(n) \leq cf(n)$, except for a finite number of values.

2.3. FIRST-ORDER UNIFICATION.

We will define the notions of term, composition of substitutions and most general unifier.

2.3.1. Term. We define a term to be a first-order expression which is not a variable.

2.3.2. Composition. We define the composition of substitutions σ_1 and σ_2, written $\sigma_1\sigma_2$, as follows.

If $\sigma_1 = \{v'_1 \leftarrow e'_1, \ . \ . \ . \ , v'_m \leftarrow e'_m\} \quad m \geq 0$

and if $\sigma_2 = \{v''_1 \leftarrow e''_1, \ . \ . \ . \ , v''_n \leftarrow e''_n\} \quad n \geq 0$

then $\sigma_2\sigma_1 = \{v'_1 \leftarrow \sigma_2(e'_1), \ . \ . \ . \ , v'_m \leftarrow \sigma_2(e'_m)\}$

$\cup \ \{v''_i \leftarrow e''_i \mid \sigma_1$ pertains to v''_i and

σ_2 does not pertain to $v''_i\}$.

The meaning of $\sigma_2\sigma_1$ is to first apply the substitution σ_1 and then to apply σ_2.

The following properties of substitutions and expressions allow us to dispense with parentheses:

$$\sigma_1(\sigma_2\sigma_3) = (\sigma_1\sigma_2)\sigma_3 \quad \text{and} \quad (\sigma_1\sigma_2)(e) = \sigma_1(\sigma_2(e))$$

for all substitutions σ_1, σ_2, σ_3 and for all expressions e.

2.3.3. Most general unifier. In this definition, X is either a set of expressions or a collection of sets of expressions.

σ is a <u>most general unifier</u> (<u>mgu</u>) of X iff
σ unifies X and $\forall \sigma' [\sigma'$ unifies X implies $\exists \sigma'' (\sigma' = \sigma'' \sigma)]$.

Not only do we wish a unification algorithm to determine if two expressions are unifiable, but also, when they are, we would like to obtain their most general unifier. In fact, all such algorithms do indeed provide the mgu. For example, in the algorithm in section 2.1, the mgu can be obtained from σ. The mgu will be expressed in the form of a composition of singleton substitutions (corresponding to successive replacements) rather than a general substitution (corresponding to simultaneous replacements) since the latter form could have exponential length.

<u>2.4.</u> <u>DESCRIPTION OF THE ALGORITHM</u>. Our algorithm consists of two stages: a transformational stage followed by a sorting stage. The transformational stage inputs the two given expressions and outputs a partition of expressions; this stage may detect failure of unification due to the attempt at unifying two expressions beginning with different constant symbols. The sorting stage constructs from the output of the transformational stage a directed graph (digraph) and determines if it contains a circuit by trying to topologically sort the digraph. If a circuit is found, unification fails because we cannot unify a variable and an expression in which the variable occurs. If no circuit is found, the topological ordering induces a substitution which

is a most general unifier.

2.4.1. Transformational Stage. This stage inputs, in general, a set of pairs of expressions to be unified, S_I and transforms them into a partition of classes of expressions, F_0. The two main sets used in this stage are S, a set of unordered pairs of expressions, and F, a partition of expressions. We allow S to contain repetitions. F, in fact, is a partition of the subexpressions occurring in S_I. We allow repetitions in each class of the partition, F, and we allow repetitions of the classes in F. As usual, two classes are either equal or disjoint. We insist that if the same variable is an element of two classes, then these classes must be equal, and that all the terms belonging to a class must begin with the same constant.

Initially, S is S_I and F_I, the initial value of F, consists of all the subexpressions occurring in S_I, each in a class of its own. Since we are not interested in the equality of two subexpressions, unless they are variables, we allow F_I to contain several identical classes containing the same term.

At the end of the transformational stage, S is empty and F is the output partition, F_0. We can intuitively consider that we are unifying S subject to the constraints represented by F.

2.4.1.1. Abstract algorithm. We present the transformational stage in the form of an abstract structured program. This will be expanded in more detail later.

```
algorithm TRANSFORM:
begin
   Initialize S to $S_I$ and F to $F_I$;
   repeat until S is empty:
   begin
      Delete a pair of expressions, $(e_1,e_2)$, from S;
      if $e_1 \neq e_2$
      then begin
              Find classes $T_1,T_2 \in F$
              such that $e_1 \in T_1$ and $e_2 \in T_2$ ;
              if $T_1 \neq T_2$
              then begin
                      if $T_1$ contains a term $f'(e_1',\ldots,e_m')$
                      and $T_2$ contains a term $f''(e_1'',\ldots,e_n'')$
                      then if $f' \neq f''$
                           then UNIFICATION FAILS
                           else Add to S the pairs:
                                $(e_1',e_1''),\ldots,(e_n',e_n'')$ ;
                       Merge $T_1$ and $T_2$ , that is,
                       replace $T_1$ and $T_2$ by $T_1 \cup T_2$ ;
                   end;
           end;
   end;
end.
```

We will now specify the data structures used to represent expressions, sets and partitions and the method of manipulating them in order to obtain an efficient algorithm.

2.4.1.2. Data structures. We represent expressions by trees (strictly, directed acyclic graphs), in which variables are shared; the set of pairs of expressions, S, by a stack; and the partition F by a forest of trees.

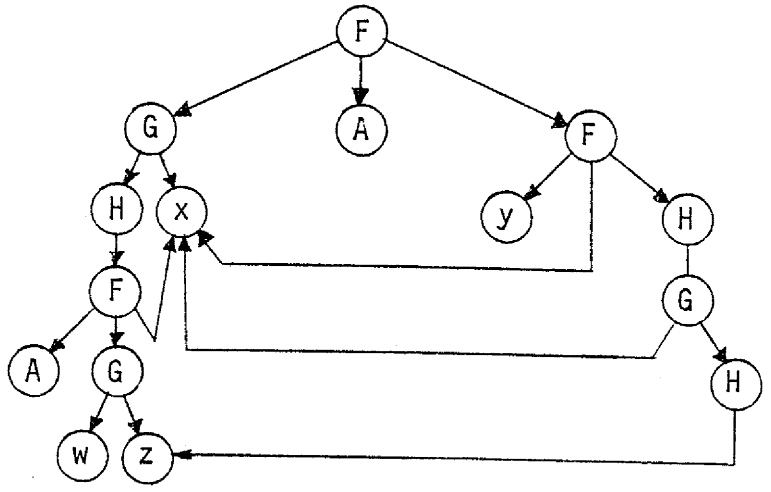
2.4.1.2.1. Expressions. The expressions are represented by trees in which different occurrences of the same variable are represented by different pointers to the same vertex of the tree. Each vertex of the tree corresponds to some symbol occurring in the expression. If a vertex corresponds to a constant symbol of degree n, then each son of the vertex corresponds to an argument. Figure 2.1 illustrates the representation of an expression.

2.4.1.2.2. Set of pairs of expressions. The set S, consisting of pairs of expressions, is represented by a stack of pairs which indicate the corresponding tree representations of the expressions. Figure 2.2 illustrates such a representation.

2.4.1.2.3. Partitions. The partition F is represented as a forest of trees. Each class in the partition is represented as a tree, each vertex of which points to an expression. Since we must know if a class contains a term, the root of a tree points to some arbitrary term within that tree. This term is known as the designated term of a class. Also, the root of a tree is effectively the name of the class which it represents. Figure 2.3 illustrates this representation. Strictly, the stack entries of S point to the corresponding vertices in F.

2.4.1.3. Data manipulation. We will describe the important operations on these data structures occurring during the transformational stage.

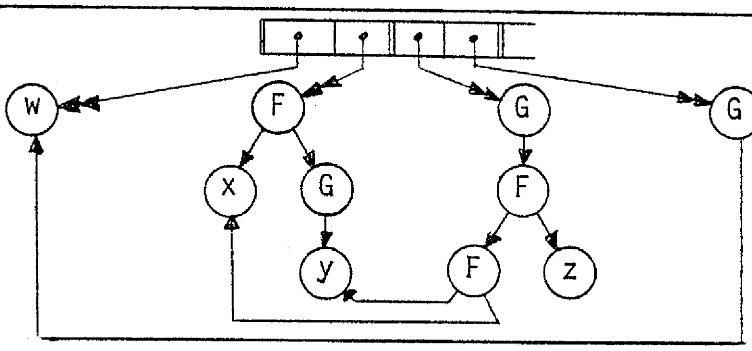
2.4.1.3.1. Expressions. In the algorithm, TRANSFORM, rather than checking if e_1 and e_2 are equal expressions, we only



The tree representation of the expression:

F(G(H(F(A,G(w,z),x)),x),A,F(y,x,H(G(x,H(z))))).

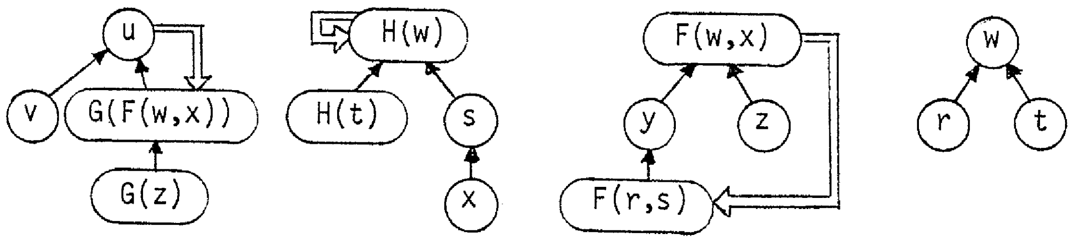
Figure 2.1.



The stack representation of S, the set of pairs of expressions:

{ {w, F(x,G(y))}, {G(F(F(y,x),z)), G(w)} }

Figure 2.2.



The forest representation of the partition:

{[u, v, G(F(w,x)), G(z)], [x,H(w),H(t),s], [F(w,x),y,z,F(r,s)], [w,r,t]}.

The big arrow indicates the designated term of each class.

Figure 2.3.

check if the pointers representing e_1 and e_2 are equal. The tree-like data structure for expressions facilitates finding the arguments of a term, that is, given a pointer to a term, $f(e_1,\ldots,e_n)$, we can easily find the pointers to its arguments, $e_1,\ldots,e_n$.

2.4.1.3.2. Manipulating S. Since S is represented as a stack, it is easy to determine if S is empty, to delete a pair from S (by "popping off" the top pair from the stack) and to add a list of pairs to S (by "pushing" them onto the top of the stack).

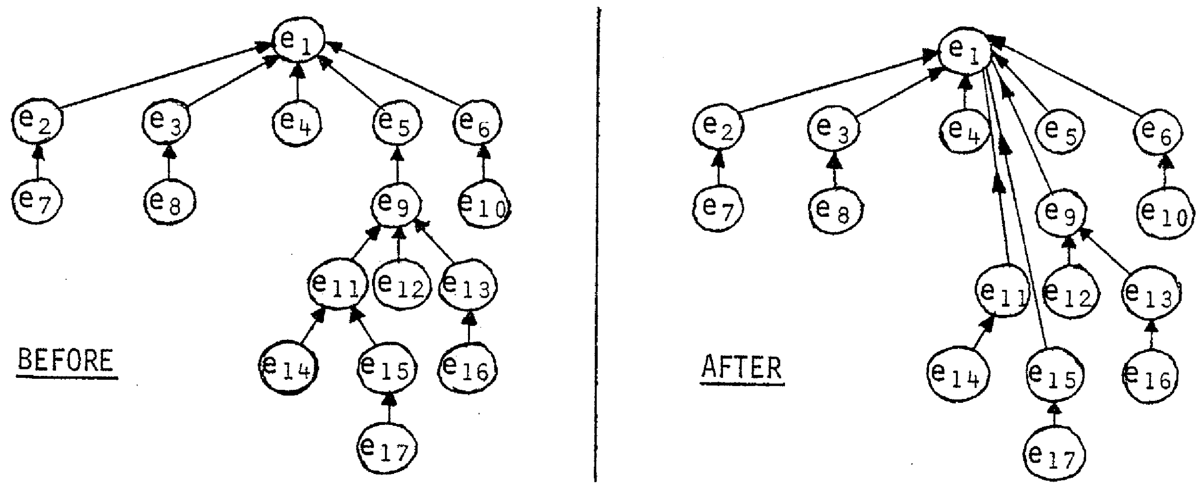
2.4.1.3.3. Manipulating F. The efficiency of the transformational stage depends upon the method of performing two operations on F: to FIND which class in the partition an expression belongs to and to MERGE two classes of the partition. These operations were analyzed by Tarjan [31].

To FIND which class an expression belongs, we traverse a path from the vertex of the tree corresponding to the expression to the root; this root is effectively the name of the required class. The cost of a FIND is proportional to the length of the path from the vertex to the root. This will be reduced if we employ a collapsing heuristic: after finding the root, we collapse the path directly onto the root. Formally, if $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{n-1} \rightarrow v_n$ is the unique path from the vertex v_1 to the root v_n then we replace the edge $v_i \rightarrow v_{i+1}$ by the edge $v_i \rightarrow v_n$ for $i=1,\ldots,n-2$. Figure 2.4 illustrates an example of a FIND.

To MERGE two classes, we make one tree representing one of the classes a subtree of the tree representing the other class. To decrease the average path length and hence the cost of subsequent FINDs, we employ a <u>balancing heuristic</u>: make the "light" tree a subtree of the "heavy" tree, where the comparatives refer to the number of vertices in the tree. In the case when the "heavy" tree contains only variables and the "light" tree contains some term, we have to ensure that the new root points to the designated term. Figure 2.5 illustrates the process of merging.

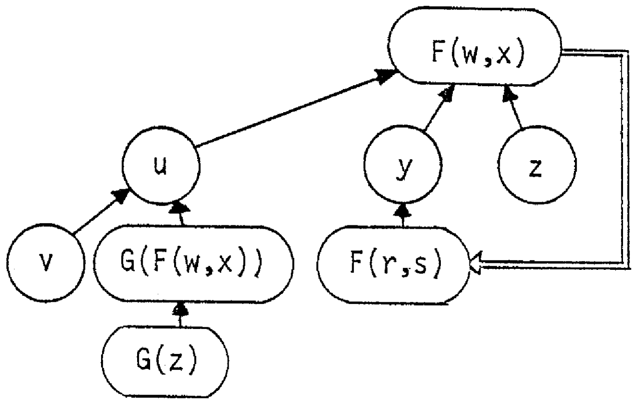
<u>2.4.2.</u> <u>Sorting stage</u>. The output partition F_0 of the transformational stage has the special property that the unifiers of F_0 and of S_I are the same. (We will formally prove this later.) It is now easy to determine if F_0 is unifiable by examining a naturally induced graph for the existence of a circuit. We will attempt to topologically sort this digraph by embedding its vertices in a linear order. If this is not possible, then the digraph contains a circuit and F_0 is therefore not unifiable. If it is possible, the sorted digraph indicates the most general unifier of F_0.

<u>2.4.2.1.</u> <u>Topological sorting</u>. Given a digraph, that is, a set E of relations of the form $v \rightarrow w$ (directed edges) on a finite set of vertices, V, the problem of <u>topological sorting</u> is to embed the digraph in a linear order, that is,



Both trees represent the class of expressions: $[e_1, e_2, \ldots, e_{17}]$, before and after FINDing the class which contains the expression e_{15}. The vertices e_{15}, e_{11}, e_9 and e_5 on the path from the given expression e_{15} to the root are collapsed directly onto the root.

Figure 2.4.



This tree is obtained after MERGing the first and third trees of Figure 2.3. It represents the class:

[u, v, G(F(w,x)), G(z), F(w,x), y, z, F(r,s)]

in which the designated term is F(r,s).

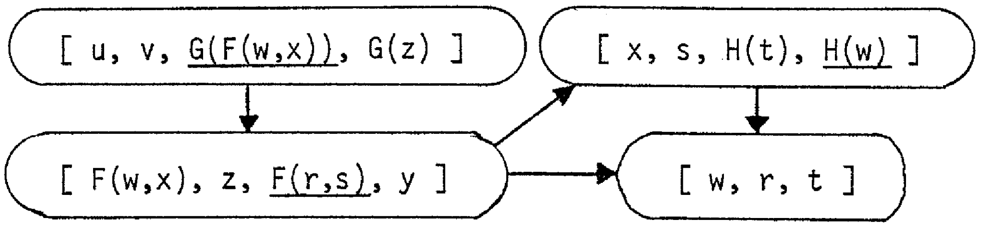
Figure 2.5.

to arrange the vertices into a linear sequence $v_1, \ldots, v_n$ such that whenever $v_i \rightarrow v_j$ we have $i<j$.

There is a well-known algorithm [18] which determines if a digraph can be topologically sorted and, if so, outputs the vertices in linear order. The time complexity of this algorithm is $O(\text{length}[V]+\text{length}[E])$ where length[V] is the number of vertices and length[E] is the number of directed edges in the digraph.

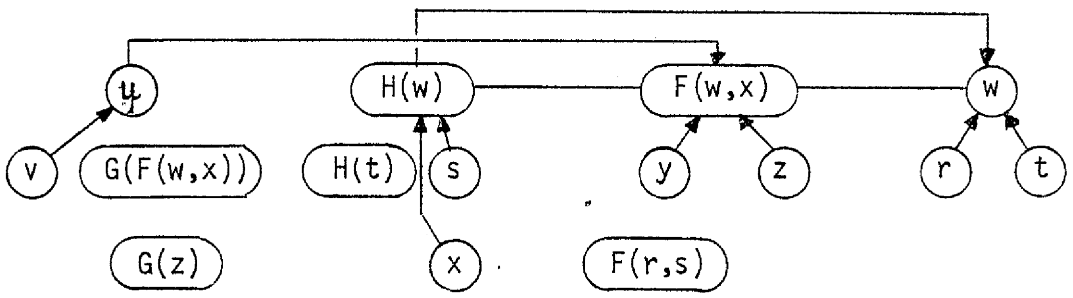
2.4.2.2. Digraph Construction. From F_0 we will construct a digraph. First we will construct an abstract intermediate digraph which is naturally induced by F_0. The final digraph is constructed from this by examining the representation of F_0 as a forest of trees.

The intermediate digraph has as vertices the classes in the partition F_0. Its edges are constructed by examining each class in F_0. Given a class T in F_0, let e be any term in T. (If no such term exists, then T contributes nothing to the set of directed edges.) Let e equal $f(e_1, \ldots, e_n)$ and let e_i belong to the class T_i $(i=1, \ldots, n)$. Then T contributes the set of directed edges: $T \rightarrow T_1, \ldots, T \rightarrow T_n$. The digraph is well-defined because it is independent of the particular choice of a term in a class. The reason is that F_0 has the special "hereditary" property: if $f(e'_1, \ldots, e'_n)$ and $f(e''_1, \ldots, e''_n)$ belong to the same class in F_0 then for each $i=1, \ldots, n$ e'_i and e''_i also belong to the same class. This will be proved later. Figure 2.6 illustrates this constructed



The directed graph associated with the partition:
{ [u,v,G(F(w,x)),G(z)], [x,H(w),H(t),s], [F(w,x),z,F(r,s),y], [w,r,t] }.
The underlined expressions denote the designated term of a class.

Figure 2.6.



The directed graph associated with Figure 2.6., using the forest representation of Figure 2.3.

Figure 2.7.

digraph.

In practice, we must construct a related digraph directly from the forest representation of F_0. From this final digraph, the mgu, if it exists, can be constructed. The vertices and edges of this digraph are obtained as follows. For each vertex, v, in the forest, which corresponds to a variable and which is not a root, let r be the root of the tree to which v belongs; add the directed edge: $v \to r$. Also, for each root, r, let $f(e_1, \ldots, e_n)$ be the designated term of the tree having root r and let r_i $(i=1,\ldots,n)$ be the root of the tree to which e_i belongs; add the directed edges: $r \to r_i$. Figure 2.7 illustrates this digraph.

A detailed algorithm, CONSTRUCT DIGRAPH, is given in the next section to construct from F_0 the digraph, which is then used by the topological sorting algorithm.

It is evident that the number of vertices and the number of edges in this constructed digraph are linear, relative to the size of the input set, S_I.

If this digraph can not be topologically sorted then the input set S_I cannot be unified. If it can be sorted, then its vertices can be embedded into a linear order. Let $v_1, \ldots, v_n$ be the subsequence of this linear order which corresponds to variables only. Then the mgu is

$$\{v_1 \leftarrow e_1\} \ . \ . \ . \ \{v_n \leftarrow e_n\}$$

where e_i is the designated term of the class to which v_i belongs; if no such designated term exists then e_i is the variable which corresponds to the root of the tree to which v_i belongs. The algorithm, OUTPUT UNIFIER, inputs the vertices of the digraph in their linear order and outputs the mgu. If $v_1,\ldots,v_n$ is the output of the topological sorting algorithm, then OUTPUT UNIFIER(v_1),...,OUTPUT UNIFIEP(v_n) gives the required unifier.

2.4.3. <u>Flowchart</u>. We present the transformational stage of our algorithm in the form of a flowchart. This flowchart will be used later in the proof of correctness and in the timing analysis. The flowchart inputs a set, S_I, of pairs of expressions to be unified and either exits with failure due to the attempt at unifying two expressions beginning with different constant symbols, or exits with the output partition F_0. See Figure 2.8 for the flowchart.

2.4.4. <u>Example</u>. We illustrate the transformational stage of the algorithm for the case when $S_I=\{\{e_1,e_2\}\}$ where

$$e_1 = P(x,G(F(x,w)),v,F(F(u,u),t),x,G(w))$$

and

$$e_2 = P(F(G(y),G(z)),u,G(F(r,s)),y,F(u,v),G(w)).$$

Figure 2.9 gives the status of the program variables S and F for each cycle of the transformational stage.

Here, the stage exits with output partition F_0 equal to

$\{[e_1,\ e_2],$

$[G(F(x,w)),\ u,\ G(y),\ v,\ G(F(r,s)),\ G(z)],$

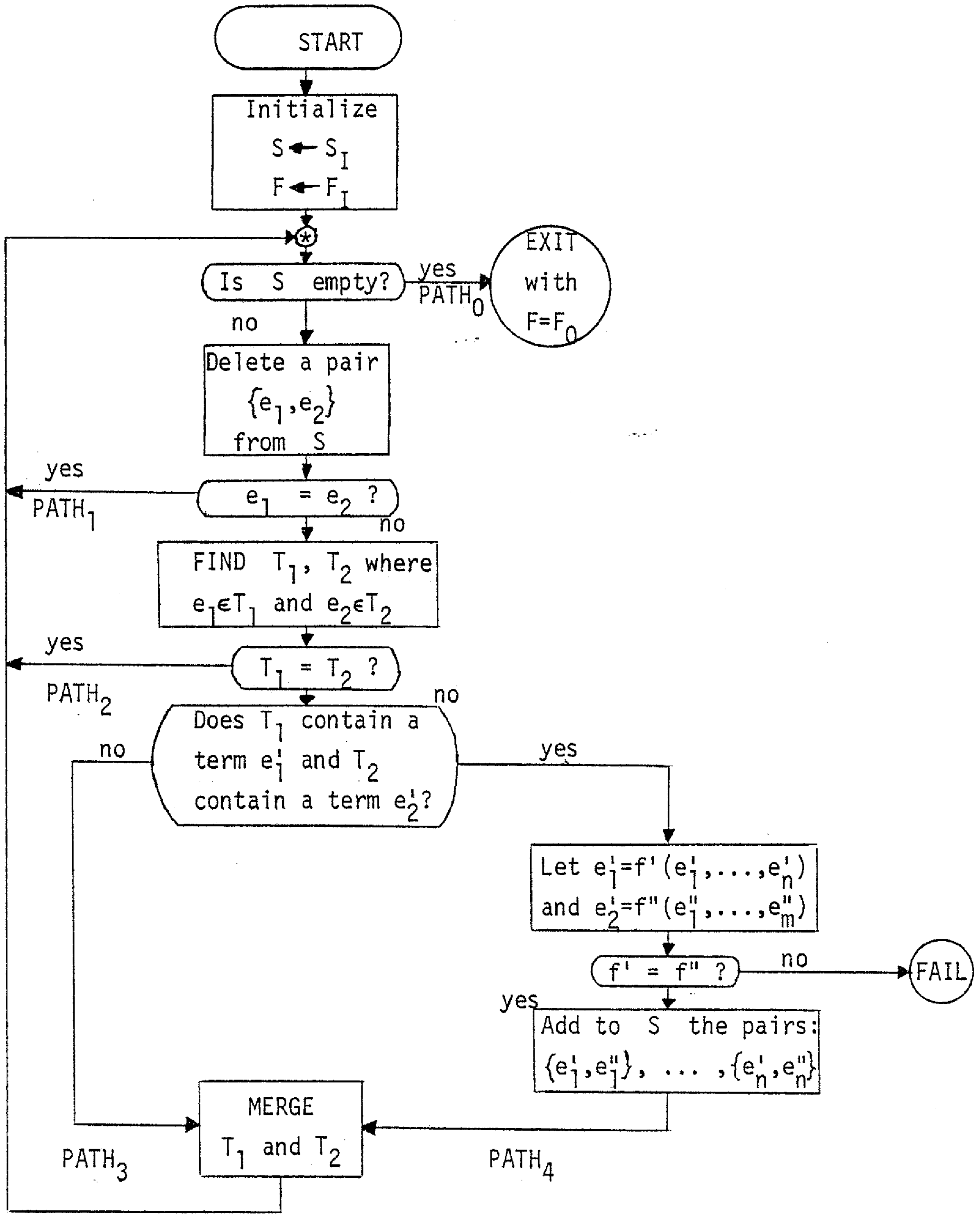
$[F(F(u,u),t),\ y,\ F(x,w),\ F(r,s),\ z],$

[F(u,u), x, F(G(y),G(z)), F(u,v), r],
[t, w, s] }.

2.4.5. Programs. Since the flowchart we presented was abstract, in that the data structures were not specified and the method of performing the operations was not described, we also give a structured program, TRANSFORM. This uses several subprograms: DECOMPOSE, which attempts to unify two terms by examining their corresponding arguments, FIND, which determines the class of the partition a given expression belongs to, and MERGE, which constructs the union of two classes in the partition.

Each expression is, in fact, a pointer to a list of its attributes: SYMBOL, VARIABLE, ARGLIST; PARENT, TERM and WEIGHT. The first three describe the expressions and hence never change and the last three attributes are used in the representation of F as a forest of trees.

SYMBOL is the first symbol of an expression, VARIABLE indicates whether the expression is a variable or a term and if it is a term, ARGLIST points to a list of its arguments. In the forest representation of the partition, PARENT indicates the next vertex towards the root of a tree. If PARENT is null, then the expression is at the root of a tree in which case, if the class associated with the tree contains a term, TERM indicates the designated term. WEIGHT is the number of vertices in the tree and is used to balance trees during a merge operation. Initially, for all subex-


START
Initialize
$S \leftarrow S_I$
$F \leftarrow F_I$
Is S empty?
yes
$PATH_0$
EXIT with $F=F_0$
no
Delete a pair $\{e_1,e_2\}$ from S
$e_1 = e_2$?
yes
$PATH_1$
no
FIND T_1, T_2 where $e_1 \epsilon T_1$ and $e_2 \epsilon T_2$
$T_1 = T_2$?
yes
$PATH_2$
no
Does T_1 contain a term e_1' and T_2 contain a term e_2'?
no
yes
Let $e_1'=f'(e_1',\ldots,e_n')$ and $e_2'=f''(e_1'',\ldots,e_m'')$
$f' = f''$?
no
FAIL
yes
Add to S the pairs: $\{e_1',e_1''\}, \ldots ,\{e_n',e_n''\}$
MERGE T_1 and T_2
$PATH_3$
$PATH_4$

Flowchart for the transformational stage of the practically linear unification algorithm.

Figure 2.8.

C	D	F
0	1	$[\underline{e_1}]$
0	2	$[x]$
0	3	$[\underline{GFxw}]$
0	11	$[\underline{Fxw}]$
0	14	$[w]$
0	4	$[v]$
0	5	$[\underline{FFuut}]$
0	13	$[\underline{Fuu}]$
0	3	$[u]$
0	14	$[t]$
0	7	$[\underline{Gw}]$
0	1	$[\underline{e_2}]$
0	2	$[\underline{FGyGz}]$
0	8	$[\underline{Gy}]$
0	5	$[y]$
0	9	$[\underline{Gz}]$
0	12	$[z]$
0	4	$[\underline{GFrs}]$
0	12	$[\underline{Frs}]$
0	18	$[r]$
0	19	$[s]$
0	6	$[\underline{Fuv}]$
0	7	$[\underline{Gw}]$

C	D	F
1		$[\underline{e_1},e_2]$
2	6	$[x,\underline{FGyGz}]$
3	8	$[\underline{GFxw},u]$
4	9	$[v,\underline{GFrs}]$
5	11	$[\underline{FFuut},y]$
6	13	$[x,\underline{FGyGz},Fuv]$
7		$[\underline{Gw},Gw]$
8	16	$[\underline{GFxw},u,Gy]$
9	16	$[v,\underline{GFrs},Gz]$
11	17	$[\underline{FFuut},y,Fxw]$
12	17	$[\underline{Frs},z]$
13	18	$[Fuu,x,\underline{FGyGz},Fuv]$
14	19	$[t,w]$
16		$[GFxw,u,Gy,v,\underline{GFrs},Gz]$
17		$[\underline{FFuut},y,Fxw,Frs,z]$
18		$[Fuu,x,\underline{FGyGz},Fuv,r]$
19		$[t,w,s]$

C=0 indicates $F = F_I$
D=" " indicates $F = F_0$

C	D	P	S
0	1	4	$\{e_1,e_2\}$
1	2	3	$\{x,FGyGz\}$
1	3	3	$\{GFxw,u\}$
1	4	3	$\{v,GFrs\}$
1	5	3	$\{FFuut,y\}$
1	6	4	$\{x,Fuv\}$
1	7	4	$\{Gw,Gw\}$
6	8	4	$\{Gy,u\}$
6	9	4	$\{Gz,v\}$
7	10	1	$\{w,w\}$
8	11	4	$\{Fxw,y\}$
9	12	3	$\{Frs,z\}$
11	13	4	$\{Fuu,x\}$
11	14	3	$\{t,w\}$
13	15	2	$\{u,Gy\}$
13	16	4	$\{u,Gz\}$
16	17	4	$\{Fxw,Frs\}$
17	18	3	$\{Fuu,r\}$
17	19	3	$\{t,s\}$
	20	0	

C=0 indicates $S = S_I$
P=0 indicates $S = \emptyset$

"C" indicates at what cycle a set was created.
"D" indicates at what cycle a set was deleted.
"P" refers to the particular path in the flowchart taken during a cycle. Underlined expressions are the designated terms of classes in F.

The left table indicates F_I, the middle table indicates the rest of F during the algorithm and the right table indicates S.

Figure 2.9.

pressions of S_I, PARENT is null, WEIGHT is one and for terms, TERM points to itself, otherwise for variables, TERM is null.

```
algorithm TRANSFØRM:
begin
   Initialize STACK and FØREST;
   repeat until STACK is empty:
   begin
      PØP a pair of expressions, {expr',expr"} off STACK;
      if expr' ≠ expr" (by virtue of different pointers)
      then begin
              root'←FIND(expr');  root"←FIND(expr");
              if root' ≠ root"
              then begin
                      if TERM[root'] ≠ null and
                         TERM[root"] ≠ null
                      then DECØMPØSE(TERM[root'],
                                     TERM[root"]) ;
                      MERGE(root',root");
                   end;
           end;
   end;
end.


algorithm DECØMPØSE(expr',expr"):
begin
   if SYMBØL[expr'] ≠ SYMBØL[expr"]
   then EXIT with FAILURE
   else for each argument pair: arg', arg"
               found from ARGLIST[expr'], ARGLIST[expr"]
        do: PUSH arg' and arg" onto STACK;
end.


algorithm FIND(vertex):
begin
   comment use the top of STACK as a temporary LIST;
   v ← vertex ;
   repeat until PARENT[v] = null:
   begin
      add v to LIST;  v ← PARENT[v];
   end;
   return(v); comment since it is now the root;
   comment collapse the path directly onto the root;
   for each w on LIST do: PARENT[w] ← v;
end.


algorithm MERGE(root',root"):
```

```
begin
   assume that WEIGHT[root'] ≤ WEIGHT[root"]
   otherwise, swap root' and root" in:
   begin
      light ← root';  heavy ← root";
      PARENT[light] ← heavy ;
      WEIGHT[heavy] ← WEIGHT[heavy] + WEIGHT[light];
      comment if necessary, update the new designated term;
      if TERM[light] ≠ null
      then TERM[heavy] ← TERM[light];
   end;
end.
```

```
algorithm CONSTRUCT DIGRAPH:
begin
   for all variables, v, in $F_0$ do:
   if PARENT[v] ≠ null
   then add the directed edge, v → FIND(v);
   for all roots, r, in $F_0$ do:
   if TERM[r] ≠ null
   then for each argument of TERM[r] do:
        add the directed edge, r → FIND(argument);
end.
```

```
algorithm OUTPUT UNIFIER(vertex):
begin
   if VARIABLE[vertex]
   then comment CONSTRUCT DIGRAPH has previously made all
                variables point directly to the root;
        if PARENT[vertex] = null
        then if TERM[vertex] ≠ null
             then output {vertex ← TERM[vertex]};
        else if TERM[PARENT[vertex]] ≠ null
             then output {vertex ← TERM[PARENT[vertex]]}
             else output {vertex ← PARENT[vertex]};
end.
```

2.5. CORRECTNESS. We will prove that if our algorithm comprising the transformational stage and sorting stages halts, it does so correctly. That is, if failure of unification is reported then indeed the input set S_I is not unifiable and if a substitution is output then indeed this is a mgu for S_I. That our algorithm halts will be evident from the complexity analysis of the next section.

We will prove that if failure is reported during the transformational stage then S_I is not unifiable. If this stage exits successfully, we will prove that F_0 and S_I have the same mgu. During the sorting stage, if a circuit is detected (that is, the induced digraph cannot be topologically sorted) then we will prove that F_0 is not unifiable, and hence S_I is not unifiable. If a substitution is output we will prove that it is a mgu for F_0 and hence also for S_I.

2.5.1. Transformational stage. Our correctness proof uses the techniques of program verification [10]. We will show:

$$\forall\sigma(\sigma \text{ is a mgu of } S_I \text{ iff } \sigma \text{ is a mgu of } F_0).$$

This is a consequence of the simpler condition:

$$\forall\sigma(\sigma \text{ unifies } S_I \text{ iff } \sigma \text{ unifies } F_0)$$

by virtue of simple properties of expressions.

To prove this simpler condition we consider the following assertion, $A(S,F)$, concerning the state of the algorithm given by the values of S and F:

$A(S,F)$ iff $\forall\sigma(\sigma$ unifies S_I iff σ unifies S and σ unifies $F)$.

We will prove that A is an invariant assertion at the point of the transformational flowchart labelled "*", that is, if A is true at "*" then A remains true if control returns to "*".

Initially, A is true since $A(S,F)$ iff $A(S_I,F_I)$ and since each class of F_I is a singleton. If we can prove that A is invariant, then if the exit is successful, S is empty and F equals F_0 hence $A(S,F)$ iff $A(\emptyset,F_0)$ which is equivalent to our simpler condition.

To prove that A remains invariant it suffices to prove $B(S',F',S'',F'')$ where this is defined as

$$\forall\sigma(\sigma \text{ unifies } S' \ \& \ \sigma \text{ unifies } F' \text{ iff } \sigma \text{ unifies } S'' \ \& \ \sigma \text{ unifies } F'')$$

where S' and F' are the old values of S and F at "*" and S'' and F'' are the new values of S and F when "*" is next reached. The reason is that:

$B(S',F',S'',F'')$ implies $[A(S',F')$ implies $A(S'',F'')]$.

To prove that B holds, we must examine each possible path in the flowchart which starts and finishes at "*".

Proof: $PATH_1$: Consider $PATH_1$ of the flowchart. Here $S' = S'' + \{e_1,e_2\}$ (that is, $S'' \cup \{\{e_1,e_2\}\}$) and $F'=F''$. $B(S'' + \{e_1,e_2\},F',S'',F')$ is true since $e_1=e_2$ implies that σ unifies $\{e_1,e_2\}$.

$PATH_2$: Again $B(S''+\{e_1,e_2\},F',S'',F')$ is true since $e_1 \in T_1$ and $e_2 \in T_2$ and $T_1 = T_2$.

$PATH_3$: Here, $S' = S'' + \{e_1,e_2\}$ and if $F'=F \cup \{T_1,T_2\}$ then $F'' = F \cup \{T_1 \cup T_2\}$. $B(S''+\{e_1,e_2\},F\cup\{T_1,T_2\},S'', F\cup\{T_1 \cup T_2\})$ holds since σ unifies $\{e_1,e_2\}$ and σ unifies T_1 and σ unifies T_2 iff σ unifies $T_1 \cup T_2$.

$PATH_4$: Let $S' = S + \{e_1,e_2\}$ then $S'' = S \cup \{\{e'_1,e''_1\},\ldots,\{e'_n,e''_n\}\}$ where $e'=f'(e'_1,\ldots,e'_n)$ and $e''=f''(e''_1,\ldots,e''_n)$. Let $F' = F \cup \{T_1,T_2\}$ then $F'' = F \cup \{T_1 \cup T_2\}$. $B(S',F',S'',F'')$ holds since

σ unifies $\{e_1, e_2\}$ and σ unifies T_1 and σ unifies T_2
iff σ unifies $\{\{e'_1, e''_1\}, \ldots, \{e'_n, e''_n\}\}$ and σ unifies $T_1 \cup T_2$.

(end of proof)

Having proved that A is invariant at the point "*" we can now prove that if any path ends in failure, then S_I is not unifiable because:

$f' \neq f''$ implies σ does not unify $\{f'(e'_1, \ldots, e'_n), f''(e''_1, \ldots, e''_m)\}$

2.5.2. Sorting stage. We will first show that if a circuit exists in the constructed digraph (that is, the induced digraph cannot be topologically sorted) then F_0 is not unifiable. By the construction of the digraph, a circuit implies a sequence of trees in the forest F_0: $T_0, T_1, \ldots, T_n$ and a sequence of roots of these trees: $r_0, r_1, \ldots, r_n$ $(n \geq 0)$ with the following property: the class corresponding to T_i contains a term e_i which has some argument, e'_{i+1}, which belongs to the class corresponding to T_{i+1}. (Addition is modulo n.)

Therefore, if σ is a unifying substitution:

σ unifies F_0

implies σ unifies each tree in F_0

implies in particular, σ unifies $T_0, T_1, \ldots, T_n$

implies $\sigma(e'_i) = \sigma(r_i) = \sigma(e_i)$ for $i=0,1,\ldots,n$

implies $\text{length}[\sigma(e'_i)] = \text{length}[\sigma(e_i)] > \text{length}[\sigma(e'_{i+1})]$

implies $\text{length}[\sigma(e'_0)] > \ldots > \text{length}[\sigma(e'_n)] > \text{length}[\sigma(e'_0)]$.

The contradiction implies that F_0 is not unifiable.

We will now prove that if a substitution is output by the sorting stage, then it is a mgu of F_0.

First, we prove that a hereditary property holds of the output partition, F_0: if $f(e^*_1,\ldots,e^*_n)$ and $f(e^{**}_1,\ldots,e^{**}_n)$ belong to the same class of F_0 then, for all $i=1,\ldots,n$ e^*_i and e^{**}_i belong to the same class.

This will be a corollary of a subsequent result. First, let us introduce notation which abbreviates the above concepts.

$$e_1 \equiv e_2 \text{ mod } X \text{ iff } \exists T(T \in X \text{ and } e_1 \in T \text{ and } e_2 \in T).$$

When X is a partition, F, then $e_1 \equiv e_2$ mod F means that e_1 and e_2 belong to the same class of F. When X is a set of pairs of expressions, S, then $e_1 \equiv e_2$ mod S means that $\{e_1, e_2\}$ belongs to S.

Also, define $e' \equiv e''$ mod*(S,F) by

$\exists e_1 \ldots \exists e_n [e' = e_1$ & $e_n = e''$ & $1 \le i < n$ implies

$(e_i \equiv e_{i+1}$ mod S or $e_i \equiv e_{i+1}$ mod F$)]$.

Note that $e_1 \equiv e_2$ mod*($\emptyset$,F) iff $e_1 \equiv e_2$ mod F.

To prove the hereditary property, consider the following assertion, H(S,F), concerning the state of the transformational stage, given by the values of S and F, defined by

H(S,F) iff

$f(e^*_1,\ldots,e^*_n) \equiv f(e^{**}_1,\ldots,e^{**}_n)$ mod F

implies $\forall i [e^*_i \equiv e^{**}_i$ mod*(S,F)$]$.

We will prove that H is an invariant assertion at the point of the flowchart labelled "*". Initially, $H(S_I, F_I)$

holds, since each class in F_I is a singleton. After proving the invariance, when the flowchart exits successfully, $H(\phi,F_0)$ becomes:

$f(e_1^*,\ldots,e_n^*) \equiv f(e_1^{**},\ldots,e_n^{**}) \bmod F_0$
implies $\forall i[e_i^* \equiv e_i^{**} \bmod F_0]$.

The proof of the invariance of H is similar to that of A (in section 2.5.1.). For each path, starting and finishing at "*", we prove that $H(S',F')$ implies $H(S'',F'')$ where S' and F' are the starting values and S'' and F'' are the finishing values of S and F.

Proof: $PATH_1$: The invariance depends upon the following simple properties.

$e' \equiv e'' \bmod S + \{e_1,e_2\}$ implies $e' \equiv e'' \bmod S$ or $e' \equiv e'' \bmod \{\{e_1,e_2\}\}$, and $e_1 = e_2$ implies $e' \equiv e'' \bmod \{\{e_1,e_2\}\}$ implies $e' = e''$ implies $e' \equiv e'' \bmod F$. From which, $H(S+\{e_1,e_2\},F)$ implies $H(S,F)$.

$PATH_2$: The relevant result is :
$e_1 \equiv e_2 \bmod S$ implies $e' \equiv e'' \bmod \{\{e_1,e_2\}\}$ implies $e' \equiv e'' \bmod S$.

$PATH_3$: Since not both of the classes T_1 and T_2 contain terms,

$f(e_1^*,\ldots,e_n^*) \equiv f(e_1^{**},\ldots,e_n^{**}) \bmod \{T_1 \cup T_2\}$ implies
$f(e_1^*,\ldots,e_n^*) \equiv f(e_1^{**},\ldots,e_n^{**}) \bmod \{T_i\}$ for i=1 or 2.

Also, $e' \equiv e'' \bmod \{\{e_1,e_2\}\}$ and $e' \equiv e'' \bmod \{T_1,T_2\}$ both imply $e' \equiv e'' \bmod \{T_1 \cup T_2\}$.

$PATH_4$: The difficult case is when $f(e_1^*,\ldots,e_n^*) \in T_1$ and $f(e_1^{**},\ldots,e_n^{**}) \in T_2$ (or vice versa). Here,

$f(e^*_1,\ldots,e^*_n) \equiv f'(e'_1,\ldots,e'_n)$ and $f=f'$ so, for all i,

$e^*_i \equiv e'_i \bmod{}^*(S+\{e_1,e_2\}, F\cup\{T_1,T_2\})$

and $e^{**}_i \equiv e''_i \bmod{}^*(S+\{e_1,e_2\}, F\cup\{T_1,T_2\})$.

Now, $e' \equiv e'' \bmod \{\{e_1,e_2\}\}$ implies $e' \equiv e'' \bmod \{T_1 \cup T_2\}$ hence $e^*_i \equiv e'_i \bmod{}^*(S, F\cup\{T_1\cup T_2\})$ and similarly for e^{**}_i, e''_i. Consequently,

$$e^*_i \equiv e^{**}_i \bmod{}^*(S \cup \bigcup_{i=1}^{n} \{e'_i, e''_i\}, F\cup\{T_1\cup T_2\}).$$

(end of proof)

If the digraph can be topologically sorted, we will show

$$\forall\sigma(\sigma \text{ is a mgu of } F_0 \text{ iff } \sigma \text{ is a mgu of } F^*_0)$$

where $F^*_0 = \{T^*_0 \mid T \in F_0\}$

and $T^* = \{e \mid e \in T$ and (e is a variable or

e is the designated term of T)$\}$.

Clearly, σ^*, the substitution induced by the output of the topological sorting stage is the most general unifier of F^*_0. We will prove that σ^* is the mgu of F_0 and, by the preceding section, the mgu of S_I. It suffices to prove the easier condition:

$$\forall\sigma(\sigma \text{ unifies } F_0 \text{ iff } \sigma \text{ unifies } F^*_0).$$

Proof: The "only if" part is trivial; the "if" part will be proved by considering the statements, $D_i(\sigma)$, defined by:

$D_i(\sigma)$ iff $\forall r(1 \le r < i$ implies σ unifies $T^*_r)$ and

$\forall r(i \le r \le m$ implies σ unifies $T_r)$

for $i=0,1,\ldots,m$ where $T_1,\ldots,T_m$ is a linear ordering of the classes of F_0. In order to prove

$$\forall\sigma[D_m(\sigma) \text{ implies } D_0(\sigma)]$$

we will prove

$$\forall\sigma[D_i(\sigma) \text{ implies } D_{i-1}(\sigma)] \quad \text{for } i=1,\ldots,m.$$

The main result used to prove this is:

$$\forall\sigma\{[\sigma \text{ unifies } T_r^* \ \& \ \forall s(r<s\leq m \text{ implies } \sigma \text{ unifies } T_s)] \text{ implies } \sigma \text{ unifies } T_r\}$$

and a consequence of our hereditary property:

$$f(e_1',\ldots,e_n') \equiv f(e_1'',\ldots,e_n'') \bmod \{T_r\}$$

implies

$$\forall i\exists s>r[e_i' \equiv e_i'' \bmod \{T_s\}] .$$

To prove the former, if T_r contains no terms, then the result is trivial, else let $e^*=f(e_1^*,\ldots,e_n^*)$ be the designated term in T_r. Now σ unifies T_r iff

$$\sigma \text{ unifies } T_r^* \text{ and for all terms, } e, \ \sigma(e)=\sigma(e^*).$$

Let $e=f(e_1,\ldots,e_n)$ then $\sigma(e)=\sigma(e^*)$ iff, for all i, $\sigma(e_i)=\sigma(e_i^*)$. We have $r<s\leq m$ implies

$$\sigma \text{ unifies } T_s \text{ implies } \sigma(e_i)=\sigma(e_i^*) \text{ implies } \sigma(e)=\sigma(e^*).$$

2.6. TIMING ANALYSIS. We will show that the time complexity of the transformational stage is $O(nG(n))$ and similarly for the time taken to construct the digraph. The time complexity of the sorting stage is $O(n)$. Consequently, the time complexity for the unification algorithm is $O(nG(n))$. Throughout this section n is the length of the input set, S_I.

2.6.1. A slowly-growing function.

Define the function A, related to Ackermann's function, for pairs of non-negative integers by:

$A(0,x) = 2x \quad \text{for } x \geq 0$

$A(i,0) = 0 \quad \text{for } i \geq 1$

$A(i,1) = 2 \quad \text{for } i \geq 1$

$A(i,x) = A(i-1, A(i,x-1)) \text{ for } i \geq 1 \text{ and } x \geq 2.$

It is easily shown that $A(1,x)=2$ for $x \geq 1$ and $A(2,x) = 2\uparrow 2 \uparrow \ldots \uparrow 2$ where "$\uparrow$" denotes exponentiation and there are x occurrences of 2. Thus $A(3,4) = 2\uparrow 2 \uparrow \ldots \uparrow 2$ (65536 occurrences of two).

Define α as a kind of "inverse" of A by:

$\alpha(m,n) = \min\{z \geq 1 \mid A(z,4\lceil m/n \rceil) > \log_2 n\}$ for $m,n \geq 1$.

Finally define $G(n) = \alpha(n,n)$. G grows extremely slowly: $G(n) \leq 3$ for all "practical" values of n, that is, for all $n < 2^{A(3,4)}$.

2.6.2. Transformational stage.

By neglecting the cost of FIND and MERGE instructions, we will show that the number of steps required by the transformational stage is linear relative to the input length. Consequently, there can only be a linear number of FIND and MERGE instructions.

To show this linearity, we examine each step of the flowchart. Most operations require a constant time. For example, to determine if a class contains a term requires merely the inspection of the TERM field of the tree which represents the class. The exceptional case is the operation: "add to S all the pairs of arguments", since the number of pairs depends upon the degree of the constant symbol. However, we can easily "absorb" this additional cost by effectively counting the edges rather than the vertices in the tree representation of an expression.

Proceeding formally, let TIME[S,F] be the additional time taken to process the sets S and F when the algorithm reaches the point labelled "*" in the flowchart. We now set up a system of recursive equations involving TIME, by considering all paths in this flowchart.

PATH$_0$: If the exit is taken after finding S empty, we have

$$\mathrm{TIME}[\emptyset;F_0] = c_1$$

where c_1 is the constant time taken to determine if S is empty.

PATH$_1$: Here

$$\mathrm{TIME}[S+\langle e_1, e_2\rangle;F] = \mathrm{TIME}[S;F] + c_2$$

where c_2 is a constant.

PATH$_2$: Similarly,

$$\mathrm{TIME}[S+\langle e_1, e_2\rangle;F] = \mathrm{TIME}[S;F] + c_3$$

where we initially neglect the cost of a FIND.

PATH$_3$: Where the cost of a MERGE is ignored,

$$\mathrm{TIME}[S+\langle e_1, e_2\rangle;F\cup\langle T_1, T_2\rangle] = \mathrm{TIME}[S;F\cup\langle T_1 \cup T_2\rangle] + c_4 .$$

PATH$_4$:

$$\mathrm{TIME}[S+\langle e_1, e_2\rangle;F\cup\langle T_1, T_2\rangle] =$$

$$\mathrm{TIME}[S\cup\langle\langle e_1', e_1''\rangle, \ldots, \langle e_m', e_m''\rangle\rangle;F\cup\langle T_1 \cup T_2\rangle + c_5 m + c_6$$

if $f'=f''$

c_7 if $f' \neq f''$.

Here, $c_5 m$ is the time taken to add to S the m pairs of arguments and c_6, c_7 are constants.

To simplify these equations, we absorb the $c_5 m$ term into the TIME measure by the transformation:

$$\text{TIME}^*[S;F] = \text{TIME}[S;F] + c_5 . \text{pairs}[S]$$

where pairs[S] is the number of pairs in the set S. Also in each equation, we will replace each equality by "≤" and each c_i by their maximum, c_{max}. Finally, the transformation

$$T[S;F] = \text{TIME}^*[S;F] \,/\, c_{max}$$

transforms each c_i into unity. Without loss of generality we can assume that $c_1=c_7=0$. Since

$$\text{TIME}[S;F] \leq \text{TIME}^*[S;F] = c_{max} . T[S;F]$$

once we show that T is linear, it follows that TIME is also.

Our set of equations become

PATH_0: $T[\emptyset;F_0] = 0$

PATH_1: $T[S+\langle e_1,e_2\rangle;F] \leq T[S;F] + 1$

PATH_2: $T[S+\langle e_1,e_2\rangle;F] \leq T[S;F] + 1$

PATH_3: $T[S+\langle e_1,e_2\rangle;F\cup\langle T_1,T_2\rangle] \leq T[S;F\cup\langle T_1 \cup T_2\rangle] + 1$

PATH_4: $T[S+\langle e_1,e_2\rangle;F\cup\langle T_1,T_2\rangle]$

$$\begin{cases} \leq T[S\cup\{\langle e_1',e_1''\rangle,\ldots,\langle e_m',e_m''\rangle\};F\cup\langle T_1 \cup T_2\rangle] + 1 & \text{if } f'=f'' \\ = 0 & \text{if } f'\neq f''. \end{cases}$$

We now prove that T[S;F] is linear relative to the lengths of S and F. In fact, we will prove that

$$T[S;F] \leq \text{pairs}[S] + \text{degree}[F]$$

where we first define "degree" for expressions by

$$\text{degree}[f(e_1,\ldots,e_m)] = m.$$

We extend degree to sets of expressions in which all terms, if any, begin with the same constant symbol:

$$\text{degree}[T] = \begin{cases} \text{degree}[e] & \text{if } T \text{ contains a term, } e \\ 0 & \text{if } T \text{ contains only variables.} \end{cases}$$

Finally, we extend degree to a partition containing such sets:

$$\text{degree}[F] = \sum_{T\in F} \text{degree}[T].$$

We now prove our time estimate by induction on pairs[S] + degree[F].

Proof: PATH_0: $T[\emptyset;F_0]=0 \le \text{pairs}[\emptyset] + \text{degree}[F_0]$ is trivially true for our basis of induction.

PATH_1 or PATH_2:

$$\begin{aligned} T[S+\langle e_1,e_2\rangle;F] &\le T[S;F] + 1 \\ &\le \text{pairs}[S] + \text{degree}[F] + 1 \\ &= \text{pairs}[S+\langle e_1,e_2\rangle] + \text{degree}[F]. \end{aligned}$$

PATH_3: $T[S+\langle e_1,e_2\rangle;F\cup\langle T_1,T_2\rangle]$

$$\begin{aligned} &\le T[S;F\cup\langle T_1\cup T_2\rangle] + 1 \\ &\le \text{pairs}[S] + \text{degree}[F\cup\langle T_1\cup T_2\rangle] + 1 \\ &= \text{pairs}[S+\langle e_1,e_2\rangle] + \text{degree}[F] + \text{degree}[T_1] \\ &\qquad + \text{degree}[T_2] \end{aligned}$$

(since one of T_1 or T_2 contains no terms)

$$= \text{pairs}[S+\langle e_1,e_2\rangle] + \text{degree}[F\cup\langle T_1,T_2\rangle].$$

PATH_4: The result is trivial when $f'\neq f''$, otherwise,

$$T[S+\langle e_1,e_2\rangle;F\cup\langle T_1,T_2\rangle]$$

$$\leq T[S \cup \{\{e_1', e_1''\}, \ldots, \{e_m', e_m''\}\}; F \cup \{T_1 \cup T_2\}] + 1$$
$$\leq \text{pairs}[S \cup \{\{e_1', e_1''\}, \ldots, \{e_m', e_m''\}\}]$$
$$+ \text{degree}[F \cup \{T_1 \cup T_2\}] + 1$$
$$= \text{pairs}[S] + m + \text{degree}[F] + m + 1$$
$$= \text{pairs}[S + \{e_1, e_2\}] + \text{degree}[F \cup \{T_1, T_2\}]$$

(since degree$[T_1]$ = m = degree$[T_2]$).

Hence, in particular,

$$T[S_I; F_I] \leq \text{pairs}[S_I] + \text{degree}[F_I]$$

which is clearly proportional to the length of S_I. Therefore, ignoring the cost of FIND and MERGE instructions, the time taken during the transformational stage is linear, relative to the length of S_I. Note that T[S;F] can be interpreted as the number of cycles the flowchart takes, and since

$$\text{pairs}[S_I] + \text{degree}[F_I] \leq \text{length}[S_I]$$

the number of cycles taken altogether is bounded above by length$[S_I]$.

We must now include the cost of the FIND and MERGE instructions. We will use the following result of Tarjan, taken from [31], in which FIND and MERGE operations are processed by path-compression on balanced trees.

"If T(m,n) is the maximum time required by a sequence of $m \geq n$ FINDs and n-1 intermixed MERGEs on a universe of n elements, then there exist constants k_1 and k_2 such that

$$k_1 m\alpha(m,n) \leq T(m,n) \leq k_2 m\alpha(m,n)."$$

We must justify using this result for our special case. For the upper bound, the time required to process the FIND and MERGE instructions is bounded above by $k_2 n\alpha(n,n)$ where n is the length of S_I.

To establish a lower bound, we note that Tarjan's result applies to an arbitrary sequence of FINDs and MERGEs, whereas our sequences have a special character.

The lower bound result of Tarjan implies that, given n, there exists a set of n elements and a sequence of n FIND and n-1 MERGE instructions requiring $\mathcal{O}(n\alpha(n,n))$ time. This enables us to construct a unification problem of size 4n which requires $\mathcal{O}(n\,\alpha(n,n))$ time, implying that given n there exists a unification problem requiring $\mathcal{O}(nG(n))$ time. The unification problem will be to unify $F(u_{i_1},\ldots,u_{i_{2n-1}})$ and $F(v_{i_1},\ldots,v_{i_{2n-1}})$ where the u's and v's belong to a set of n variables (corresponding to the set of n elements) and are defined as follows. Consider the above sequence of FIND and MERGE instructions. If the r-th instruction is FIND(e) then let u_{i_r} correspond to e and let v_{i_r} correspond to the element which is at the root of the tree containing e. If the r-th instruction is to MERGE the classes T_1 and T_2 then let u_{i_r} correspond to the root of the tree representing T_1 and let v_{i_r} correspond to the root of the tree representing T_2. To solve this unification problem our algorithm essentially processes the original sequence of FIND and MERGE instructions and hence takes $\mathcal{O}(nG(n))$ time.

Consequently, the total time taken by the transformational stage is $O(nG(n))$, that is, practically linear.

2.6.3. _Sorting stage_. The algorithm to construct the directed graph also involves some FIND instructions. The number of edges E and vertices V in the digraph are linear relative to the length of S_I since V is, at most, the number of subexpressions in S_I and E is proportional to the length of S_I. Hence the total time taken during this construction is also practically linear.

The topological sorting stage requires linear time, that is, $O(V+E)$ time. The algorithm which constructs the mgu from the sorted digraph is also linear, since it examines the vertices V.

In conclusion, the total time required by our algorithm, comprising the transformational stage, the digraph construction, the topological sorting stage and the unifier construction is practically linear, that is, requires $O(nG(n))$ time.

CHAPTER 3

THE COMPLEXITY OF SUBSUMPTION

In this chapter we will prove that the problem of subsumption is polynomially-complete, that is, has computational complexity equivalent to the problem of determining if a formula of propositional calculus is satisfiable.

3.1. PROPOSITIONAL CALCULUS.

In order to define the SATISFIABILITY problem we first present our notation for propositional formulas.

3.1.1. Notation.

We use the following logical connectives in our formulas: $\bar{\ }$ (negation), v (disjunction) and & (conjunction).

An atom is the basic constituent of all our propositional formulas. We will use P, Q, R, P_1, Q_1, R_1,... for atoms.

A literal is either an atom or a negated atom.

A clause is a disjunction of literals.

A system is a conjunction of clauses.

A pdnf (perfect disjunctive normal formula) is a disjunction of conjunctions of literals such that each atom occurring in the formula occurs exactly once in each conjunction. Although there are systematic methods of converting a formula into a logically equivalent pdnf, we are only interested in converting clauses containing at most three literals, that is, in the following valid equivalences, where L_1, L_2 and L_3 are literals.

L_1 iff L_1

$L_1 \vee L_2$ iff $(L_1 \& L_2) \vee (L_1 \& \overline{L_2}) \vee (\overline{L_1} \& L_2)$

$L_1 \vee L_2 \vee L_3$ iff $(L_1 \& L_2 \& L_3) \vee (L_1 \& L_2 \& \overline{L_3}) \vee (L_1 \& \overline{L_2} \& L_3) \vee (L_1 \& \overline{L_2} \& \overline{L_3})$
$\vee (\overline{L_1} \& L_2 \& L_3) \vee (\overline{L_1} \& L_2 \& \overline{L_3}) \vee (\overline{L_1} \& \overline{L_2} \& L_3)$.

Here $\overline{L_i}$ equals $\overline{A_i}$ if L_i equals the atom A_i otherwise equals A_i if L_i equals a negated atom, $\overline{A_i}$. For example, the pdnf which is equivalent to the clause $\overline{P} \vee Q$ is:

$$(\overline{P} \& Q) \vee (\overline{P} \& \overline{Q}) \vee (P \& Q).$$

3.1.2. Truth-valuation. The semantic notion of satisfiability is made precise by defining a truth-valuation as a mapping from the set of atoms to the truth-set, {true, false}, and which is extended to any formula of propositional calculus by the usual truth-table method. We use τ to denote truth-valuations. The definitions of satisfies and satisfiable follow.

τ satisfies the formula F iff $\tau(F)$=true.

F is satisfiable iff $\exists \tau$ (τ satisfies F).

3.1.3. SATISFIABILITY. The SATISFIABILITY problem is to determine whether or not a given system is satisfiable.

The SATISFIABILITY$_3$ problem is to determine whether or not a given system, in which each clause contains at most three literals, is satisfiable.

3.1.4. Length. We will later need the following measure of size for formulas. We define the length of a formula as the total number of occurrences of atoms in the formula. In particular, if S is the system $C_1 \& \ldots \& C_n$ where clause C_i contains l_i literals then

$$\text{length}[S] = \sum_{i=1}^{n} \text{length}[C_i] = \sum_{i=1}^{n} l_i.$$

3.2. COMPUTATIONAL COMPLEXITY.

In order to make precise statements about the complexity of problems and the equivalence in complexity of problems we summarize the definitions found in the literature[8,17].

3.2.1. Problem. A problem can be rigorously defined as a particular set of strings from some alphabet; an instance of the problem is then a specified element of that set of strings. A Turing Machine (or algorithm) solves a problem if, given an instance of the problem as input, it halts in an accepting state (or returns the value, "true").

3.2.2. P and NP. We are only interested in the time complexity of problems, the time being the number of steps taken by a multitape Turing Machine to solve the problem.

P is the class of problems which can be solved within time $t(n)$ on a deterministic Turing Machine where t is some polynomial and n is the length of an instance of the problem. For example, the problem of determining whether or not a truth-valuation satisfies a formula, belongs to P since there is an algorithm for solving this in linear time, using the well-known method for compiling and evaluating infix formulas.

NP is the class of problems which can be solved within polynomial time on a non-deterministic Turing Machine. A non-deterministic Turing Machine differs from a deterministic one in that, in general, the next state in a computation is chosen from a set of states. A non-deterministic machine accepts an input if some sequence of choices of states results in an accepting state. The time taken by a non-deterministic Turing Machine is the number of steps in some sequence of choices. For example, SATISFIABILITY belongs to NP since the following non-deterministic algorithm solves this problem by "guessing" a truth-valuation and determining if it satisfies the system.

```
algorithm SATISFIABLE(S):
begin
   for each atom A in S do
       choose from one of:
              choice$_T$: let $\tau(A)$ = true;
              choice$_F$: let $\tau(A)$ = false ;
   return (SATISFIES($\tau$,S));
end.
```

The non-deterministic feature of this algorithm is given by the keyword, choose, which specifies that the statement to be executed is chosen from the alternatives given by choice$_T$ and choice$_F$. SATISFIES(τ,S) determines whether or not τ satisfies S and requires only linear time. Every sequence of choices has length equal to the length of S therefore it should be clear that every sequence takes polynomial time. Consequently, SATISFIABILITY belongs to

NP.

3.2.3. Polynomial-reducibility. A problem P_1 is polynomial-time reducible to problem P_2, written $P_1 \leq_p P_2$, if an instance of problem P_1 can be solved in polynomial time on a deterministic Turing Machine which consults an external oracle to solve problem P_2; the time taken to consult the oracle is unity and the time required by the oracle is ignored.

If $P_1 \leq_p P_2$ then loosely speaking, an efficient algorithm to solve P_1 implies an efficient algorithm to solve P_2.

In all our constructions we will present a method, requiring polynomial time, which converts an instance I_1 of problem P_1 into an instance I_2 of a problem P_2 such that I_1 iff I_2. This is a sufficient condition that $P_1 \leq_p P_2$.

3.2.4. NP-complete problems. A problem P is NP-complete if it satisfies:

(1) P belongs to NP

and (2) If Q belongs to NP then $Q \leq_p P$.

By a theorem of Cook [8], (2) can be replaced by

(2') SATISFIABILITY $\leq_p$ P.

For example, the following problems are known to be NP-complete[8,17]: HAMILTON - to determine if a graph has a cycle which includes each vertex exactly once; TAUTOLOGY - to determine if a formula of propositional calculus is satisfied by every truth-valuation; SATISFIABILITY$_3$ - hence condition (2') can be changed to

(2") SATISFIABILITY$_3$ $\leq_p$ P.

3.3. SUBSUMPTION.

The SUSUMPTION problem is to determine whether or not a set of expressions, A, subsumes another set of expressions B. The expressions in A and B are of first-order and B contains no variables.

A subsumes B iff $\exists\sigma\,\sigma(A) \subseteq B$ where $\sigma(A) = \{\sigma(e) \mid e \in A\}$.

Theorem 3: SUBSUMPTION is an NP-complete problem.

Proof: We will show that SUBSUMPTION belongs to NP and that SATISFIABILITY$_3$ $\leq_p$ SUBSUMPTION.

3.3.1. Non-deterministic algorithm.

To show that SUBSUMPTION belongs to NP we present a non-deterministic algorithm for determining whether or not a set of expressions A subsumes a set of expressions B which contain no variables.

```
algorithm SUBSUMES(A,B):
begin
   if A is empty
   then return (true)
   else begin
           delete any expression e from A;
           choose an expression E from B;
           if there exists σ such that σ(e)=E
           then return (SUBSUMES(σ(A),B)
           else return (false);
        end;
end.
```

A subsumes B if there is some sequence of choices of expressions from B resulting in "true" being returned.

3.3.1.1. Correctness. The correctness of the algorithm is justified by the results that $\emptyset$ subsumes B and that $A \cup \langle e \rangle$ subsumes B iff

$$\exists E \langle E \in B \;\&\; \exists \sigma \, [\sigma(e) = E \;\&\; \sigma(A) \text{ subsumes } B] \rangle.$$

3.3.1.2. Efficiency. Each sequence of choices in the non-deterministic algorithm, SUBSUMES, takes polynomial time in relation to the input length. This is due to the length of each sequence being no greater than the number of elements in the set A and to the time taken by each operation being polynomial. In particular the time taken to find a unifier for $\langle e, E \rangle$ is linear. Consequently, SUBSUMPTION belongs to NP.

3.3.2. Reducibility. We will prove SATISFIABILITY$_3 \le_p$ SUBSUMPTION by constructing an instance of the SUBSUMPTION problem from an instance of the SATISFIABILITY$_3$ problem. From a system, S, in which each clause has at most three literals, we will construct sets of expressions E_1 and E_2 such that

(1) E_1 subsumes E_2 iff S is satisfiable

and (2) E_1 and E_2 can be constructed efficiently from S.

3.3.2.1. Construction. Let S be a system in which each clause contains at most three literals. We will construct an instance of the SUBSUMPTION problem, that is, sets of expressions E_1 and E_2 in which variables occur only in E_1. We will now define the symbols occurring in E_1 and E_2.

3.3.2.1.1. Symbols. Let $\{P_1,\ldots,P_m\}$ be the set of atoms occurring in S. Then $\{p_1,\ldots,p_m\}$ will be the set of variables occurring in E_1. We define a bijective mapping, α, which establishes a correspondence between these sets by:

$$\alpha(P_i) = p_i \quad \text{for } i=1,\ldots,m.$$

T and F will be constants of degree zero, corresponding to "true" and "false". We define another bijection, β, which establishes this correspondence by:

$$\beta(\text{true}) = T \quad \text{and} \quad \beta(\text{false}) = F.$$

Associated with each clause of the system is a constant. Let the system, S, have n clauses, $S = C_1\&\ldots\&C_n$, in which the i-th clause, C_i, contains l_i (=1,2 or 3) literals. Introduce G_i as a constant symbol of degree l_i for $i=1,\ldots,n$.

3.3.2.1.2. Expressions. Using these symbols ($p_1,\ldots,p_m$, $T,F,G_1,\ldots,G_n$) we will now construct the sets of expressions E_1 and E_2. Only E_1 will contain variables.

Let the system be $S = C_1\&\ldots\&C_n$ and let $P_{i,1},\ldots,P_{i,l_i}$ be some fixed ordering of the atoms which occur in the i-th clause, C_i, of S. (For example, let the order be induced by the left to right order of the atoms in C_i.)

We construct E_1 by first constructing expressions $E_i^!$ by:

$$E_i^! = G_i(p_{i,1},\ldots,p_{i,l_i}) \quad i=1,\ldots,n$$

where $p_{i,j} = \alpha(P_{i,j})$ for $j=1,\ldots,l_i$. Then the set of expressions E_1 is

$$E_1 = \langle E'_1, \ldots, E'_n \rangle.$$

We now construct E_2 according to:

$$E_2 = E_C[1, C_1] \cup \ldots \cup E_C[n, C_n].$$

Let clause C_i have the equivalent pdnf given by

$$C_i \text{ iff } D_{i,1} \vee \ldots \vee D_{i,s_i} \quad (s_i = 2^{l_i} - 1)$$

then we define the sets of expressions $E_C[i, C_i]$ by:

$$E_C[i, C_i] = \langle E_D[i, D_{i,1}], \ldots, E_D[i, D_{i,s_i}] \rangle \quad i=1,\ldots,n.$$

Let the disjunct $D_{i,j}$ equal $L_{i,j,1}$ & ... & L_{i,j,l_i} where the literal $L_{i,j,k}$ is either $P_{i,k}$ or $\overline{P}_{i,k}$ for $k=1,\ldots,l_i$, then we define the expressions $E_D[i, D_{i,j}]$ using the symbols G_i :

$$E_D[i, D_{i,j}] = G_i(E_L[L_{i,j,1}], \ldots, E_L[L_{i,j,l_i}])$$

for $i=1,\ldots,n$; $j=1,\ldots,s_i$. Finally, we define E_L by:

$$E_L[L_{i,j,k}] = \begin{cases} T & \text{if } L_{i,j,k} = P_{i,k} \\ F & \text{if } L_{i,j,k} = \overline{P}_{i,k}. \end{cases}$$

Note that E_2 does not contain variables.

<u>3.3.2.1.3</u>. <u>Example</u>. We will now illustrate the construction by examining the system:

$$S = P \;\&\; (\overline{P} \vee Q) \;\&\; \overline{R} \;\&\; (\overline{Q} \vee R \vee \overline{P}).$$

Let p, q and r be the variables corresponding to the atoms P, Q and R. Then

$$E_1 = \langle G_1(p),\ G_2(p,q),\ G_3(r),\ G_4(q,r,p) \rangle$$

and

$$E_2 = E_C[1, P] \cup E_C[2, \overline{P} \vee Q] \cup E_C[3, \overline{R}] \cup E_C[4, \overline{Q} \vee R \vee \overline{P}],$$

where for example, since the pdnf equivalent to $\overline{P} \vee Q$ is

$$(\overline{P} \;\&\; Q) \vee (\overline{P} \;\&\; \overline{Q}) \vee (P \;\&\; Q)$$

we have

$E_C[2,\overline{P} \vee Q]$

$= \langle E_D[2,\overline{P}\&Q],\ E_D[2,\overline{P}\&\overline{Q}],\ E_D[P\&Q]\rangle$

$= \langle G_2(E_L[\overline{P}],E_L[Q]),\ G_2(E_L[\overline{P}],E_L[\overline{Q}]),\ G_2(E_L[P],E_L[Q])\rangle$

$= \langle G_2(F,T),\ G_2(F,F),\ G_2(T,T)\rangle.$

Altogether, E_2 equals

$\langle G_1(T),$

$G_2(F,T),\ G_2(F,F),\ G_2(T,T),$

$G_3(F),$

$G_4(F,T,F),\ G_4(F,T,T),\ G_4(F,F,F),\ G_4(F,F,T),$

$G_4(T,T,F),\ G_4(T,T,T),\ G_4(T,F,F)\rangle.$

3.3.2.2. Validity. We will show that our construction is valid.

Lemma: Let S be a system in which each clause contains at most three literals and let E_1 and E_2 be the sets of expressions constructed from S. Then

E_1 subsumes E_2 iff S is satisfiable,

that is,

$$\exists\sigma[\sigma(E_1) \subseteq E_2 \quad \text{iff} \exists\tau[\tau \text{ satisfies } S].$$

Proof: A simple correspondence between the substitution of variables by T or F and truth-valuations establishes the lemma. The relation is given by:

$$\tau = \beta^{-1}\sigma\alpha \quad \text{and} \quad \sigma = \beta\tau\alpha^{-1}$$

where α and β are bijections defined in section 3.3.2.1.1. For example, the truth-valuation:

$\tau(P) = \text{true},\ \tau(Q) = \text{false},\ \tau(P) = \text{true}$

is associated with the substitution:

$$\langle p \leftarrow T,\ q \leftarrow F,\ r \leftarrow T\rangle.$$

The following equivalences establish that

$$\tau \text{ satisfies } S \text{ iff } \sigma(E_1) \subseteq E_2$$

where σ and τ are related as above. The lemma then easily follows. The following equivalences are due to simple properties of truth-valuations, substitutions and the definitions used in the construction.

τ satisfies $S = C_1 \& \ldots \& C_n$

iff $\forall i\ \tau$ satisfies $C_i = D_{i,1} \vee \ldots \vee D_{i,s_i}$

iff $\forall i \exists j\ \tau$ satisfies $D_{i,j} = L_{i,j,1} \& \ldots \& L_{i,j,l_i}$

iff $\forall i \exists j \forall k\ \tau$ satisfies $L_{i,j,k}$

iff $\forall i \exists j \forall k\ \tau(L_{i,j,k}) = \text{true}$

iff $\forall i \exists j \forall k \langle [L_{i,j,k} = P_{i,k} \ \& \ \tau(P_{i,k}) = \text{true}] \vee$
$\qquad [L_{i,j,k} = \overline{P}_{i,k} \ \& \ \tau(P_{i,k}) = \text{false}]\rangle$

iff $\forall i \exists j \forall k\ \sigma(p_{i,k}) = E_L[L_{i,j,k}]$ where $p_{i,k} = \alpha(P_{i,k})$

iff $\forall i \exists j\ \sigma(p_{i,1}) = E_L[L_{i,j,1}] \ \& \ldots \& \ \sigma(p_{i,l_i}) = E_L[L_{i,j,l_i}]$

iff $\forall i \exists j\ \sigma(G_i(p_{i,1}, \ldots, p_{i,l_i})) =$
$\qquad G_i(E_L[L_{i,j,1}], \ldots, E_L[L_{i,j,l_i}])$

iff $\forall i \exists j\ \sigma(E_i') = E_D[i, D_{i,j}]$

iff $\forall i\ \sigma(E_i') \in \langle E_D[i, D_{i,1}], \ldots, E_D[i, D_{i,s_i}]\rangle$

iff $\forall i\ \sigma(E_i') \in E_C[i, C_i]$

iff $\forall i\ \sigma(E_i') \in E_C[1, C_1'] \cup \ldots \cup E_C[n, C_n]$
$\qquad$ (since $i \neq j$ implies $\forall \sigma \sigma(E_i') \notin E_C[j, C_j]$)

iff $\forall i\ \sigma(E_i') \in E_2$

iff $\sigma(\langle E_1', \ldots, E_n'\rangle) \subseteq E_2$

iff $\sigma(E_1) \subseteq E_2$.

3.3.2.3. Efficiency of construction. We will now show that the construction can be performed efficiently, that is, within polynomial time relative to the length of the input system, S. The main point to note is that, although the conversion of a clause to an equivalent pdnf generally requires exponential time, this is not the case here since each clause of the system has at most three literals. In fact, since there are only three cases, we will construct directly from tables the equivalent pdnf, rather than use a general conversion algorithm.

The construction can be performed in linear time relative to the length of S. This is evident from examining our construction. We will only elaborate upon the construction of $E_C[i,C_i]$ when clause C_i contains $l_i=3$ literals. For example, let the clause be $\overline{Q} \vee R \vee \overline{P}$. (Q,R,P is then our fixed ordering of the atoms in this clause.) Since this clause contains 3 literals, the following table will be used. It indicates the pdnf which is equivalent to any clause containing three literals, $L_1 \vee L_2 \vee L_3$. (We will also have a table used for clauses containing two literals; the one literal case is trivial.)

+	+	+	+	-	-	-
+	+	-	-	+	+	-
+	-	+	-	+	-	+

The i-th row of the j-th column of this table indicates if L_i is to appear negated (-) or not negated (+) in the j-th disjunct of the pdnf which is:

$(L_1 \& L_2 \& L_3) \vee (L_1 \& L_2 \& \overline{L}_3) \vee (L_1 \& \overline{L}_2 \& L_3) \vee (L_1 \& \overline{L}_2 \& \overline{L}_3) \vee$
$(\overline{L}_1 \& L_2 \& L_3) \vee (\overline{L}_1 \& L_2 \& \overline{L}_3) \vee (\overline{L}_1 \& \overline{L}_2 \& L_3)$.

Using this schema we can directly construct the pdnf that is equivalent to $\overline{Q} \vee R \vee \overline{P}$:

$(\overline{Q} \& R \& \overline{P}) \vee (\overline{Q} \& R \& P) \vee (\overline{Q} \& \overline{R} \& \overline{P}) \vee (\overline{Q} \& \overline{R} \& P) \vee$
$(Q \& R \& \overline{P}) \vee (Q \& R \& P) \vee (Q \& \overline{R} \& \overline{P})$

and hence the set of expressions, $E_C[i, \overline{Q} \vee R \vee \overline{P}]$ which equals:

$\langle G_i(F,T,F),\ G_i(F,T,T),\ G_i(F,F,F),\ G_i(F,F,T),$
$G_i(T,T,F),\ G_i(T,T,T),\ G_i(T,F,F)\rangle$.

Note that the length of $E_C[i, C_i]$ is only a constant multiple of the length of C_i since C_i contains a bounded number of literals.

The time taken by the construction is essentially that required to output the constructed sets E_1 and E_2. We will conclude the demonstration that the construction can be performed efficiently by showing that the lengths of E_1 and E_2 are linear relative to the length of S.

$$\begin{aligned}
\text{length}[E_1] &= \text{length}[\langle E'_1, \ldots, E'_n \rangle] \\
&= \sum_{i=1}^{n} \text{length}[E'_i] \\
&= \sum_{i=1}^{n} \text{length}[G_i(p_{i,1}, \ldots, p_{i,l_i})] \\
&= \sum_{i=1}^{n} (1 + l_i) \\
&\leq 2 \sum_{i=1}^{n} l_i \\
&= 2 \sum_{i=1}^{n} \text{length}[C_i] \\
&= 2\ \text{length}[S].
\end{aligned}$$

Also,

$$\begin{aligned}
\text{length}[E_2] &= \Sigma_{i=1}^{n} \ \text{length}[E_C[i, C_i]] \\
&= \Sigma_{i=1}^{n} \ \Sigma_{j=1}^{s_i} \ \text{length}[\ _D[i, D_{i,j}]] \\
&= \Sigma_{i=1}^{n} \ \Sigma_{j=1}^{s_i} \ \text{length}[\ G_i(E_L[L_{i,j,1}] \ , \ldots , \\
&\qquad\qquad\qquad E_L[L_{i,j,l_i}]) \ ] \\
&= \Sigma_{i=1}^{n} \ \Sigma_{j=1}^{s_i} \ (1 + \Sigma_{k=1}^{l_i} \ \text{length}[E_L[L_{i,j,k}]] \ ) \\
&= \Sigma_{i=1}^{n} \ \Sigma_{j=1}^{s_i} \ (1 + \Sigma_{k=1}^{l_i} \ 1) \\
&\le 2 \, \Sigma_{i=1}^{n} \ \Sigma_{j=1}^{s_i} \ l_i \\
&\le 14 \ \Sigma_{i=1}^{n} \ l_i \qquad \text{since } s_i = 2^{l_i} - 1 \le 7 \\
&= 14 \ \text{length}[S].
\end{aligned}$$

CHAPTER 4

THE COMPLEXITY OF INSTANTIATION

In this chapter we will prove that the problem of second-order instantiation is NP-complete. In order to relate our definition of a second-order expression with others, we note that we could define a second-order term as a first-order variable or constant, or as a second-order variable or constant followed by the appropriate number of second-order terms. A second-order expression could then be defined as a second-order term, possibly preceded by λ and a sequence of distinct first-order variables. The following are examples of second-order expressions: w, B, G(A,y,f(x,y)), λxy.x, λxyz.F(x,g(y,z)w,y).

4.1. INSTANTIATION.

The INSTANTIATION problem is the second-order unification problem in which one expression contains no variables. That is, it is to determine, given two expressions e and E in which E contains no variables, whether or not there exists a substitution σ such that $\sigma(e)=E$.

4.2. COMPLETENESS.

Theorem 4: INSTANTIATION is an NP-complete problem.

Proof: We will show that INSTANTIATION belongs to NP and that SATISFIABILITY $\leq_p$ INSTANTIATION.

4.2.1. Non-deterministic algorithm.

To show that

INSTANTIATION belongs to NP we present a non-deterministic algorithm for solving a special case of the INSTANTIATION problem. We will ignore the types of the expressions and we will assume that the expressions do not contain formal arguments, that is, formally, every "λ" is immediately followed by ".". We do this in order to simplify the algorithm, however, we can extend the algorithm to solve the general case by checking types to determine if certain substitutions are applicable and by including simple tests when expressions contain formal arguments. For example, none of $\lambda uv.v$, $F(A)$, $\lambda uv.F(u)$ is an instance of $\lambda xy.x$. Having shown that the special algorithm operates in non-deterministic polynomial time, it is easy to show this for the general algorithm.

4.2.1.1. Elementary unifiers. Pietrzykowski [23] presented a method for enumerating the unifiers of two second-order expressions and Huet [14] gave a procedure for determining if two expressions are unifiable, however his procedure does not, in general, halt. We apply their methods to the special case of second-order instantiation. First we define the notion of elementary unifier.

Let $e = f(e_1, \ldots, e_m)$ and $E = F(E_1, \ldots, E_n)$ be two second-order expressions in which E contains no variables and f is a variable. If $\text{order}[f]=2$ then define PROJECTION(e,E) as the following set of m substitutions:

$$\text{PROJECTION}(e,E) = \{ \{f \leftarrow \lambda u_1 \ldots u_m . u_1\}, \ldots, \{f \leftarrow \lambda u_1 \ldots u_m . u_m\} \}.$$

Also define IMITATION(e,E) as the singleton set:

$$\{\{f \leftarrow \lambda u_1 \ldots u_m . F(g_1(u_1,\ldots,u_m),\ldots,g_n(u_1,\ldots,u_m))\}\}.$$

where g_i is a "new" m-th degree variable ($i=1,\ldots,n$). A "new" variable is one which is different from all others in the appropriate context. Note that when order[f]=1, m equals zero and the set becomes $\{\{f \leftarrow F(g_1,\ldots,g_n)\}\}$.

We can now define the set of elementary unifiers for e and E:

$$EU(e,E) = PROJECTION(e,E) \cup IMITATION(e,E).$$

The usefulness of this definition is evident from the following lemma.

Lemma: Let S be any set of pairs of expressions $\{\langle e_1,E_1\rangle,\ldots,\langle e_n,E_n\rangle\}$. Variables do not occur in the expressions E, $E_1,\ldots,E_n$ and e begins with a variable. All expressions are second-order. Then

$$S \cup \{\langle e, E\rangle\} \text{ is unifiable}$$

iff

$$\exists\sigma[\sigma \text{ belongs to } EU(e,E) \;\&\; \sigma(S \cup \{\langle e,E\rangle\}) \text{ is unifiable }],$$

where

$$\sigma(S \cup \{\langle e,E\rangle\}) = \{\langle\sigma(e_1),E_1\rangle,\ldots,\langle\sigma(e_n),E_n\rangle, \langle\sigma(e),E\rangle\}.$$

Proof: The lemma is a special case of a more general completeness result found in [14].

4.2.1.2. Algorithm. We now present the non-deterministic algorithm which determines if the expression E_I is an instance of the expression e_I.

```
algorithm INSTANCE($e_I$,$E_I$) :
begin
   S ← { <$e_I$, $E_I$> };
   repeat until S is empty :
   begin
      delete a pair <e, E> from S ;
      let e=f($e_1$,...,$e_m$) and E=F($E_1$,...,$E_n$) ;
      if f is a variable
      then begin
              choose from one of:
              choice$_I$:begin
                         let $\sigma$ belong to IMITATION(e,E) ;
                         apply $\sigma$ to S ;
                         add to S the n pairs:
                            <$g_i$($\sigma$($e_1$),...,$\sigma$($e_m$)), $E_i$>
                            ( for i=1,...,n ) ;
                     end ;
              choice$_P$:if order[f] ≠ 1
                     then begin
                             choose $\sigma$ from PROJECTION(e,E);
                             apply $\sigma$ to S;
                             add to S the pair: <$\sigma$(e),E>;
                          end;
           end
      else if f=F
           then add to S the m pairs:<$e_1$,$E_1$>,...,<$e_m$,$E_m$>
           else return (false) ;
   end;
   return (true) ;
end.
```

E_I is an instance of e_I if there is some sequence of choices which causes this algorithm to return the value, "true".

4.2.1.3. Correctness. The correctness of the algorithm is justified by the lemma and the results: $\emptyset$ is unifiable and if f is a constant symbol equal to F then

$$\{ \langle f(e_1,\dots,e_m),\ F(E_1,\dots,E_n)\rangle \} \text{ is unifiable iff}$$
$$\bigcup_{i=1}^{n} \{\langle e_i, E_i\rangle\} \text{ is unifiable.}$$

4.2.1.4. Efficiency. To show that INSTANTIATION belongs to NP we will show that each sequence of choices requires only polynomial time relative to the input length, length$[e_I]$+length$[E_I]$. We will show that each operation of the algorithm requires polynomial time in the length of the input to the algorithm and that the length of a sequence of choices is bounded by a polynomial of the input length.

Many of the basic operations of the algorithm require constant time, for example, "delete a pair from S", "is f a variable?" and "apply $\{f \leftarrow \lambda u_1 \ldots u_m . u_i\}$" if suitable data structures are used. "Add to S the m pairs" requires linear time and "apply the substitution (given by choice_I)" requires quadratic time to introduce m.n new pointers in a tree-like data structure for expressions in which subexpressions are shared.

We must also show that the size of this compact representation of S cannot increase exponentially during the algorithm. We first note an invariant of S: the number of occurrences of any variable in S is always bounded (by the length of e_I). This is initially true. Also the application of an imitation to an occurrence of a variable f introduces m new variables $g_1, \ldots, g_m$. Each such application increases the size of the compact representation of S by a bounded amount (m vertices representing the variables $g_1, \ldots, g_m$ plus m.n edges indicating the n arguments for each g_i, where m and n are always bounded by the input length). Since we will shortly show that the search depth is bounded polynomially, it follows that the size of the data structure representation of S is bounded polynomially relative to the input length.

It remains to show that the depth of a search for "true" (that is, the length of a sequence of choices ending in "true") is of polynomial order. By considering two features of expressions: the number of variables in S and the lengths of the second components of the pairs in S, we can show that the depth is bounded quadratically relative to the input length.

We now define two such measures which reflect these features. Let S be a set of pairs of expressions in which variables occur only in the first component. Let vars[S] equal the total number of distinct free variables occurring in S. Define the length of S by:

$$\text{length}[S] = \Sigma_{<e,E>\in S}\text{length}[E].$$

Define M[e] to be the maximum degree of all subexpressions occurring in e by the inductive definition:

$$M[\lambda u_1\ldots u_m.s(e_1,\ldots,e_n)] = \max(1, n, M[e_1],\ldots,M[e_n]).$$

We now set up a system of recursive equations involving depth[S], which is the maximum number of choices which have to be made to reach either "true" or "false" in the non-deterministic algorithm. The equations are constructed from examining the algorithm.

D_0: depth $[\emptyset] = 0$

D_1: depth $[S \cup \{<f(e_1,\ldots,e_m), E>\}]$

$\leq \max(1 + \text{depth}[\sigma_0(S \cup \bigcup_{i=1}^{n} \{<g_i(e_1,\ldots,e_m), E_i>\})],$

$$1 + \mathrm{depth}[\sigma_1(S \cup \{\langle e_1, E\rangle\})],$$

$$\vdots$$

$$1 + \mathrm{depth}[\sigma_m(S \cup \{\langle e_m, E\rangle\})] \;)$$

where $E=F(E_1,\ldots,E_n)$,

$$\sigma_0 = \{f \leftarrow \lambda u_1\ldots u_m . F(g_1(u_1,\ldots,u_m),\ldots,g_n(u_1,\ldots,u_m))\},$$

and $\sigma_i = \{f \leftarrow \lambda u_1\ldots u_m . u_i\}$ for $i=1,\ldots,n$.

D_2: $\mathrm{depth}[S \cup \{\langle F(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}]$

$$\le 1 + \mathrm{depth}[S \cup \bigcup_{i=1}^{m} \{\langle e_i, E_i\rangle\}]$$

where F is a constant symbol.

Note that if order(f)=1 then in D_1, m=0 so we ignore $\sigma_1,\ldots,\sigma_m$.

Using these equations we now prove by induction that:

$$\mathrm{depth}[S] \le M.\mathrm{length}[S] + \mathrm{vars}[S]$$

where $M = M[E_I]$.

The basis for induction, equation D_0, is trivially true. We now prove, using the induction hypothesis on the right sides of equations D_1 and D_2, that it is also true for the left sides.

D_1: $1 + \mathrm{depth}[\sigma_0(S \cup \bigcup_{i=1}^{n} \{\langle g_i(e_1,\ldots,e_m), E_i\rangle\})]$

$$\le 1 + M\ \mathrm{length}[\sigma_0(S \cup \bigcup_{i=1}^{n} \{\langle g_i(e_1,\ldots,e_m), E_i\rangle\})]$$

$$+ \quad \mathrm{vars}[\sigma_0(S \cup \bigcup_{i=1}^{n} \{\langle g_i(e_1,\ldots,e_m), E_i\rangle\})]$$

$$\le 1 + M.(\mathrm{length}[S] + \Sigma_{i=1}^{n}\mathrm{length}[E_i])$$

$$+ \mathrm{vars}[S \cup \{\langle f(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}] + n - 1$$

(since σ_0 replaces f by the $n \le M$ variables $g_1,\ldots,g_n$)

$$\le M.\mathrm{length}[S \cup \{\langle f(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}]$$

$$+ \quad \mathrm{vars}[S \cup \{\langle f(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}].$$

Also for $i=1,\ldots,m$

$$1 + \text{depth}[\sigma_i(S \cup \{\langle e_i, E\rangle\})]$$
$$\leq 1 + M.\text{length}[\sigma_i(S \cup \{\langle e_i, E\rangle\})]$$
$$+ \quad \text{vars}[\sigma_i(S \cup \{\langle e_i, E\rangle\})]$$
$$\leq M.(\text{length}[S] + \text{length}[E]) +$$
$$\text{vars}[\sigma_i(S \cup \{\langle e_i, E\rangle\})] + 1$$
$$\leq M.\text{length}[S \cup \{\langle f(e_1,\ldots,e_m), E\rangle\}]$$
$$+ \text{vars}[S \cup \{\langle f(e_1,\ldots,e_m), E\rangle\}]$$

(since σ_i is a projection which removes the variable f).

By considering the maximum of all these m+1 results we prove the induction result for equation D_1.

D_2: $\text{depth}[S \cup \{\langle F(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}]$

$$\leq 1 + \text{depth}[S \cup \bigcup_{i=1}^{m} \{\langle e_i, E_i\rangle\}]$$
$$\leq 1 + M.\text{length}[S \cup \bigcup_{i=1}^{m} \{\langle e_i, E_i\rangle\}]$$
$$+ \quad \text{vars}[S \cup \bigcup_{i=1}^{m} \{\langle e_i, E_i\rangle\}]$$
$$\leq 1 + M.(\text{length}[S] + \Sigma_{i=1}^{m}\text{length}[E_i])$$
$$+ \text{vars}[S \cup \bigcup_{i=1}^{m} \{\langle e_i, E_i\rangle\}]$$
$$\leq M.\text{length}[S \cup \{\langle F(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}]$$
$$+ \quad \text{vars}[S \cup \{\langle F(e_1,\ldots,e_m), F(E_1,\ldots,E_n)\rangle\}].$$

Having proved $\text{depth}[S] \leq M.\text{length}[S] + \text{vars}[S]$, in particular, initially when $S=\{\langle e_I, E_I\rangle\}$

$$\text{depth}[\{\langle e_I, E_I\rangle\}]$$
$$\leq M.\text{length}[\{\langle e_I, E_I\rangle\}] + \text{vars}[\{\langle e_I, E_I\rangle\}]$$
$$= M.\text{length}[E_I] + \text{vars}[e_I]$$
$$\leq (\text{length}[e_I] + \text{length}[E_I])^2$$

(since $M \leq \text{length}[e_I]$ and $\text{vars}[e_I] \leq \text{length}[e_I]$).

Thus the depth of the search is bounded quadratically by the length of the input.

4.2.2. Reducibility. We will prove SATISFIABILITY $\leq_p$ INSTANTIATION by constructing an instance of the INSTANTIATION problem from an instance of the SATISFIABILITY problem. From a system, S, we will construct two expressions e and E such that

(1) E is an instance of e iff S is satisfiable.

(2) e and E can be constructed efficiently from S.

4.2.2.1. Construction. Let S be a system. We will construct an instance of the INSTANTIATION problem, that is, two expressions e and E in which variables occur only in e. We will now define the symbols occurring in e and E. We let the basic set of types be a singleton, hence we effectively ignore types.

4.2.2.1.1. Symbols. Let $\{P_1,\ldots,P_m\}$ be the set of atoms occurring in S, then just as in section 3.3.2.1.1., $p_1,\ldots,p_m$ will be some of the first-order variables where $p_i = \alpha(P_i)$ for $i=1,\ldots,m$. Let S be the system $C_1 \& \ldots \& C_n$ in which the i-th clause C_i contains l_i literals. Then we introduce the remaining first-order variables:

$$x_{i,j} \text{ for } i=1,\ldots,n;\ j=1,\ldots,l_i.$$

The second-order variables are g_i $(i=1,\ldots,n)$ where g_i has degree l_i.

The constants are T and F (first-order, zero-degree) corresponding to "true" and "false"; and second-order symbols: G_i (l_i-degree) for $i=1,\ldots,n$, H of degree two and K of degree 2n.

4.2.2.1.2. Expressions. Using these symbols (p_i, $x_{i,j}$, g_i, T, F, G_i, H and K) we will now construct the expressions e and E. Only e will contain variables. e will be constructed from expressions e'_i and e''_i and E will be constructed from expressions E'_i and E''_i, defined as follows.

$$e'_i = H(g_i(T,\ldots,T),\ g_i(F,\ldots,F)) \text{ for } i=1,\ldots,n$$

where there are l_i T's and l_i F's as arguments of g_i.

$$E'_i = H(T,F) \text{ for } i=1,\ldots,n.$$

Let the i-th clause of S be

$$C_i = L_{i,1} \vee \ldots \vee L_{i,l_i}$$

where the literal $L_{i,j}$ equals either the atom $P_{i,j}$ or the negated atom $\overline{P}_{i,j}$ for $j=1,\ldots,l_i$ and $P_{i,j}$ is an atom occurring in S. We then construct e''_i from expressions $e'''_{i,j}$ by:

$$e''_i = G_i(e'''_{i,1},\ldots,e'''_{i,l_i}) \text{ for } i=1,\ldots,n$$

where

$$e'''_{i,j} = g_i(x_{i,j},\ldots,p_{i,j},\ldots,x_{i,j}) \text{ for } i=1,\ldots,n;\ j=1,\ldots l_i$$

where all of the l_i arguments of g_i are $x_{i,j}$ except for the j-th argument which is $p_{i,j} = \alpha(P_{i,j})$.

Expressions E''_i are constructed from $E'''_{i,j}$ by:

$$E''_i = G_i(E'''_{i,1},\ldots,E'''_{i,l_i}) \text{ for } i=1,\ldots,n$$

where

$$E'''_{i,j} = T \text{ if } L_{i,j} = P_{i,j}$$

$$F \text{ if } L_{i,j} = \overline{P}_{i,j}$$

for $i=1,\ldots,n;\ j=1,\ldots,l_i$.

Finally, our constructed expressions are:

$$e = K(e'_1,\ldots,e'_n,e''_1,\ldots,e''_n)$$

and

$$E=K(E'_1,\ldots,E'_n,E''_1,\ldots,E''_n).$$

4.2.2.1.3. Example. We will illustrate the construction by examining the system:

$$S = P \ \& \ (\overline{P} \vee Q) \ \& \ \overline{R} \ \& \ (\overline{Q} \vee R \vee \overline{P}).$$

Let p, q and r be the variables corresponding to the atoms P, Q and R. Then

$$\begin{aligned} e = K(&H(g_1(T),\ g_1(F)),\\ &H(g_2(T,T),\ g_2(F,F)),\\ &H(g_3(T),\ g_3(F)),\\ &H(g_4(T,T,T),\ g_4(F,F,F)),\\ &G_1(g_1(p)),\\ &G_2(g_2(p,x_{2,1}),\ g_2(x_{2,2},q)),\\ &G_3(g_3(r)),\\ &G_4(g_4(q,x_{4,1},x_{4,1}),g_4(x_{4,2},r,x_{4,2}),\\ &\quad g_4(x_{4,3},x_{4,3},p))\quad) \end{aligned}$$

and

$$\begin{aligned} E = K(&H(T,F),\\ &H(T,F),\\ &H(T,F),\\ &H(T,F),\\ &G_1(T), \end{aligned}$$

$G_2(F,T)$,

$G_3(F)$,

$G_4(F,T,F)$).

4.2.2.2. Validity. We will show that our construction is valid.

Lemma: Let S be a system and let e and E be the expressions constructed from S. Then

E is an instance of e iff S is satisfiable

that is,

$$\exists \sigma\ [\sigma(e) = E\ ] \text{ iff } \exists \tau [\ \tau \text{ satisfies } S\ ].$$

Proof: First we motivate some aspects of the construction in order to intuitively understand the formal proof. E is an instance of e iff E'_i is an instance of e'_i and E''_i is an instance of e''_i for all i. Now E'_i is an instance of e'_i iff $H(T,F)$ is an instance of $H(g_i(T,\ldots,T),g_i(F,\ldots,F))$. This forces g_i to be some projection, $\lambda u_1 \ldots u_{l_i}.u_j$ rather than an imitation $\lambda u_1 \ldots u_{l_i}.T$ or $\lambda u_1 \ldots u_{l_i}.F$. (This could have been achieved by forcing $\lambda u.u$ to be an instance of $\lambda u.g_i(u,\ldots,u)$ however for simplicity we prefer to avoid expressions containing formal arguments.) Since E''_i must be an instance of e''_i, the substitution $\{g_i \leftarrow \lambda u_1 \ldots u_{l_i}.u_j\}$ forces the instantiation of $p_{i,j}$ with T or F according to whether or not the literal $L_{i,j}$ is or is not negated. (The role of the variables $x_{i,j}$ is to trivially satisfy the remaining instantiations.) The instantiation of $p_{i,j}$ corresponds to the truth-valuation, $\tau(L_{i,j})$=true. By considering all such instantiations, we effectively test S for satisfiability.

Before we formally prove the lemma, we first present a fact which justifies why we can virtually ignore the "dummy" variables $x_{i,j}$.

Fact: Let A, B_i be formulas containing expressions and the substitution σ but which do not contain the variable, f. Let E_i be expressions which contain no variables, then

$$\exists\sigma\,(A \;\&\; \exists i\,[\sigma(f)=E_i \;\&\; B_i]) \quad \text{iff} \quad \exists\sigma\,(A \;\&\; \exists i B_i).$$

The proof of this depends upon a similar result:

$$\exists\sigma\;(A \;\&\; \sigma(f) = E) \quad \text{iff} \quad \exists\sigma\, A.$$

The following equivalences establish the lemma.

E is an instance of e

iff $\exists\sigma\;\sigma(e) = E$

iff $\exists\sigma\;\sigma(K(e'_1,\ldots,e'_n,e''_1,\ldots,e''_n))=K(E'_1,\ldots,E'_n,E''_1,\ldots,E''_n)$

(by the definition of e and E)

iff $\exists\sigma K(\sigma(e'_1),\ldots,\sigma(e'_n),\sigma(e''_1),\ldots,\sigma(e''_n)) =$
$K(E'_1,\ldots,E'_n,E''_1,\ldots,E''_n)$

(since K is a constant)

iff $\exists\sigma\forall i\;[\sigma(e'_i) = E'_i \;\&\; \sigma(e''_i) = E''_i\,]$

iff $\exists\sigma\forall i\;[\sigma(H(g_i(T,\ldots,T),g_i(F,\ldots,F))) = H(T,F) \;\&$
$\sigma(G_i(e'''_{i,1},\ldots,e'''_{i,l_i}))=G_i(E'''_{i,1},\ldots,E'''_{i,l_i})\,]$

(by the definitions of e'_i, E'_i e''_i and E''_i)

iff $\exists\sigma\;\forall i\;[\sigma(g_i(T,\ldots,T)) = T \;\&\; \sigma(g_i(F,\ldots,F)) = F \;\&$
$\forall k\;\;\sigma(e'''_{i,k}) = E'''_{i,k}\,]$

(since H and G_i are constants)

iff $\exists\sigma\;\forall i\;[\exists j\;\sigma(g_i) = \lambda u_1\ldots u_{l_i}.u_j \;\;\&$

$\forall k\ \sigma(e'''_{i,k}) = E'''_{i,k}\ ]$

(since g_i cannot be an imitation)

iff $\exists\sigma\forall i\ \exists j\ [\ \sigma(g_i) = \lambda u_1 \ldots u_{l_i}.u_j\ \ \&\ \forall k\ \sigma(e'''_{i,k})=E'''_{i,k}\ ]$

iff $\exists\sigma\forall i\ \exists j\ [\ \sigma(g_i) = \lambda u_1 \ldots u_{l_i}.u_j\ \ \&$

$\forall k\ \sigma(g_i(x_{i,k}, \ldots, p_{i,k}, \ldots, x_{i,k}))=E'''_{i,k}\ ]$

(by the definition of $e'''_{i,k}$)

iff $\exists\sigma\ \forall i\ \exists j\ [\sigma(g_i) = \lambda u_1 \ldots u_{l_i}.u_j\ \ \&\ \sigma(p_{i,j}) = E'''_{i,j}\ \ \&$

$\forall k\neq j\ \ \sigma(x_{i,k}) = E'''_{i,k}\ ]$

(by applying σ to g_i)

iff $\exists\sigma\ \forall i\ \exists j\ [\ \sigma(p_{i,j}) = E'''_{i,j}]$

(by using the Fact for the variables g_i or $x_{i,j}$)

iff $\exists\sigma\ \forall i\ \exists j\ [(L_{i,j}=P_{i,j}\ \ \&\ \sigma(p_{i,j}) = T)\ \vee$

$(L_{i,j}=\overline{P}_{i,j}\ \ \&\ \sigma(p_{i,j}) = F)\ \ ]$

(by the definition of $E'''_{i,j}$).

We now use the relation between σ and τ (see 3.3.2.2.):

$$\tau = \beta^{-1}\sigma\alpha \qquad \text{and} \qquad \sigma = \beta\tau\alpha^{-1}$$

to obtain the next equivalence:

iff $\exists\tau\ \forall i\ \exists j[\ (L_{i,j}=P_{i,j}\ \ \&\ \tau(P_{i,j}) = \text{true})\ \vee$

$(L_{i,j}=\overline{P}_{i,j}\ \ \&\ \tau(P_{i,j}) = \text{false})\ \ ]$

iff $\exists\tau\forall i\ \exists j[\ \ \tau(L_{i,j}) = \text{true}\ ]$

iff $\exists\tau\forall i\ \exists j\ \ \tau$ satisfies $L_{i,j}$

iff $\exists\tau\forall i\ \ \ \tau$ satisfies $L_{i,1} \vee \ldots \vee L_{i,l_i}$

iff $\exists\tau\forall i\ \ \ \tau$ satisfies C_i

(by the definition of C_i)

iff $\exists\tau\ \tau$ satisfies $C_1\ \&\ \ldots\ \&\ C_n$

iff $\exists\tau\ \tau$ satisfies S

(by the definition of S)

iff S is satisfiable.

4.2.2.3. Efficiency of construction. We will now show that the construction can be performed efficiently, that is, within polynomial time relative to the length of the input system, S. It should be obvious from the construction that a suitable scanning algorithm suffices to construct the expressions e and E. The time required will simply be proportional to the lengths of e and E which we will now show to be quadratic relative to the length of S.

$$
\begin{aligned}
&\mathrm{length}[e] \\
&= \mathrm{length}[K(e'_1,\ldots,e'_n,e''_1,\ldots,e''_n)] \\
&= 1 + \Sigma_{i=1}^{n}\mathrm{length}[e'_i] + \Sigma_{i=1}^{n}\mathrm{length}[e''_i] \\
&= 1 + \Sigma_{i=1}^{n}\mathrm{length}[H(g_i(T,\ldots,T),g_i(F,\ldots,F))] \\
&\quad + \Sigma_{i=1}^{n}\mathrm{length}[G_i(e'''_{i,1},\ldots,e'''_{i,l_i})] \\
&= 1 + \Sigma_{i=1}^{n}(3 + 2l_i) + \Sigma_{i=1}^{n}(1 + \Sigma_{j=1}^{l_i}\mathrm{length}[e'''_{i,j}]) \\
&= 1 + 3n + 2\Sigma_{i=1}^{n}l_i + n \\
&\quad + \Sigma_{i=1}^{n}\Sigma_{j=1}^{l_i}\mathrm{length}[g_i(x_{i,j},\ldots,p_{i,j},\ldots,x_{i,j})] \\
&= 1 + 4n + 2\Sigma_{i=1}^{n}l_i + \Sigma_{i=1}^{n}\Sigma_{j=1}^{l_i}(1 + l_i) \\
&= 1 + 4n + 2\Sigma_{i=1}^{n}l_i + \Sigma_{i=1}^{n}l_i(1 + l_i) \\
&\leq 9\Sigma_{i=1}^{n}l_i^2 \\
&\leq 9(\Sigma_{i=1}^{n}l_i)^2 \\
&= 9(\Sigma_{i=1}^{n}\mathrm{length}[C_i])^2 \\
&= 9(\mathrm{length}[S])^2.
\end{aligned}
$$

Also,

$\mathrm{length}[E]$

$$= \text{length}[K(E'_1,\ldots,E'_n,E''_1,\ldots,E''_n)$$
$$= 1 + \Sigma_{i=1}^{n}\text{length}[E'_i] + \Sigma_{i=1}^{n}\text{length}[E''_i]$$
$$= 1 + \Sigma_{i=1}^{n}\text{length}[H(T,F)] + \Sigma_{i=1}^{n}\text{length}[G_i(E'''_{i,1},\ldots,E'''_{i,l_i})]$$
$$= 1 + 3n + \Sigma_{i=1}^{n}(1 + \Sigma_{j=1}^{l_i}\text{length}[E'''_{i,j}])$$
$$= 1 + 4n + \Sigma_{i=1}^{n}\Sigma_{j=1}^{l_i}1$$
$$\leq 6\Sigma_{i=1}^{n}l_i$$
$$= 6\,\text{length}[S].$$

Thus the sum of the lengths of the constructed expressions e and E is at most $15(\text{length}[S])^2$ and therefore the time taken to construct e and E from S is polynomial (in fact, quadratic) relative to the length of S.

CHAPTER 5

THE UNDECIDABILITY OF 3RD-ORDER 2-ND DEGREE UNIFICATION.

In this chapter we will show that the third-order second-degree unification problem (in which all degrees are at most two) is undecidable. This refines the results of Huet[13] and Lucchesi [19] who independently proved the undecidability of third-order unification, but under the condition that the degree is arbitrary. Whereas they reduced the Post Correspondence Problem to the unification problem, we will reduce the Diophantine Problem of number theory.

The results of Huet and Lucchesi imply that there exists a $k \geq 4$ such that the third-order k-th degree unification problem is undecidable. This follows from the fact that there exits a $k \geq 4$ such that the Post Correspondence Problem with k pairs of strings is undecidable and that Huet and Lucchesi construct from an instance of the Post Correspondence Problem with k pairs of strings an instance of the third-order k-th degree unification problem.

5.1. 3-RD ORDER 2-ND DEGREE UNIFICATION PROBLEM.

This problem is the third-order unification problem in which the degree of all symbols occurring in the expressions is at most two.

5.2. DIOPHANTINE PROBLEM.

The Diophantine Problem is to determine whether or not a diophantine equation has a solution in natural numbers, that is, to determine if

$$\exists m_1 \ldots \exists m_N \; P[m_1,\ldots,m_N] = Q[m_1,\ldots,m_N]$$

where P and Q are multinomials (multivariable polynomials) in the variables $m_1,\ldots,m_N$ and have non-negative integer coefficients.

5.2.1. Multinomial. A multinomial in the variables $m_1,\ldots,m_N$ is defined inductively by:

(1) 1 is a multinomial

(2) $m_1,\ldots,m_N$ are multinomials

(3) if P and Q are multinomials then $(P+Q)$ is a multinomial

(4) if P and Q are multinomials then $(P.Q)$ is a multinomial.

The diophantine problem was first posed as Hilbert's Tenth Problem in 1900 and was recently found to be undecidable by Matijasevic [20].

5.3. CONSTRUCTION.

From a diophantine problem we will construct a third-order second-degree unification problem by first constructing from a multinomial in $m_1,\ldots,m_N$ an expression.

5.3.1. Symbols. Our expressions will be constructed using the following symbols: the variables: u, f, $\phi_1,\ldots,\phi_N$ and the constant: C. The types of these symbols are given by:

$\text{type}[u] = t$

$\text{type}[f] = (t\rightarrow t)$

$\text{type}[C] = (t,t\rightarrow t)$

$\text{type}[\phi_i] = ((t\rightarrow t),t\rightarrow t)$ for $i=1,\ldots,N$.

Each third-order variable ϕ_i corresponds to an integer m_i and C is used to make a list of expressions.

The order of all symbols is at most three and the degree is at most two, hence the constructed unification problem will be of third-order and second-degree.

5.3.2. Expressions. Using the above symbols we will define by induction an expression E_P associated with each multinomial P. The type of E_P is t and the variables f and u occur free in E_P. Let $E_P[e',e'']$ denote the application of the substitution $\{f \leftarrow e',\ u \leftarrow e''\}$ to E_P.

The definition of E_P is by induction on multinomials:

$$
\begin{aligned}
E_1 &= f(u)\\
E_{m_i} &= \phi_i(\lambda u.f(u),u) \quad \text{for } i=1,\dots,N\\
E_{(P+Q)} &= E_P[\lambda u.f(u),\ E_Q]\\
E_{(P.Q)} &= E_P[\lambda u.E_Q,u].
\end{aligned}
$$

Note that $E_{(P+Q)} \neq E_{(Q+P)}$, however this is not important.

We can now construct from the Diophantine Problem:

$$\exists m_1 \dots \exists m_N\ P[m_1,\dots,m_N] = Q[m_1,\dots,m_N]$$

the two expressions e_1 and e_2 given by:

$$e_1 = \lambda fu.C(\phi_1(\lambda u.u,u),C(\phi_2(\lambda u.u,u),\dots,C(\phi_N(\lambda u.u,u),E_P)\dots))$$

$$e_2 = \lambda fu.C(u,C(u,\dots,C(u,E_Q)\dots)).$$

5.3.3. Example. We will illustrate the construction by considering the Diophantine Problem:

$$\exists m_1 \exists m_2 (\ m_2^3 + m_1 m_2 = 3m_1^2 + 2m_2 + 3\ ).$$

We must first bracket the multinomials to fit our definition. One such bracketing for $m_2^3 + m_1 m_2$ is:

$$P = ((m_2.(m_2.m_2))+(m_1.m_2))$$

and for $3m_1^2 + 2m_2 + 3$ is:

$$Q = (((m_1.m_1).(1+(1+1)))+((m_2.(1+1))+((1+1)+1))).$$

We find that

$$E_{(m_2.(m_2.m_2))} = \phi_2(\lambda u.\ \phi_2(\lambda u.\ \phi_2(\lambda u.f(u),u),u),u)$$
$$E_{(m_1.m_2)} = \phi_1(\lambda u.\ \phi_2(\lambda u.f(u),u),u)$$
$$E_{(1+(1+1))} = E_{((1+1)+1)} = f(f(f(u)))$$
$$E_{(m_2.(1+1))} = \phi_2(\lambda u.f(f(u)),u).$$

Altogether the constructed expressions are:

$$e_1 = \lambda fu.C(\phi_1(\lambda u.u,u),C(\phi_2(\lambda u.u,u),E_P))$$

and

$$e_2 = \lambda fu.C(u,C(u,E_Q))$$

where

$$E_P = \phi_2(\lambda u.\phi_2(\lambda u.\phi_2(\lambda u.f(u),u),u),\phi_1(\lambda u.\phi_2(\lambda u.f(u),u),u))$$

and

$$E_Q = \phi_1(\lambda u.\phi_1(\lambda u.f(f(f(u))),u),\phi_2(\lambda u.f(f(u)),f(f(f(u))))).$$

5.4. VALIDITY. We will prove that the construction is valid, from which our main theorem follows:

Theorem 5: The third-order second-degree unification problem is undecidable.

Proof: Given a Diophantine Problem, we construct, as above, two expressions e_1 and e_2. We will prove that:

$\exists m_1 \ldots \exists m_N\ P[m_1,\ldots,m_N] = Q[m_1,\ldots,m_N]$ iff

$\langle e_1, e_2 \rangle$ is unifiable.

Since the Diophantine Problem is undecidable, so also is the third-order second-degree unification problem because "is $\{e_1, e_2\}$ unifiable?" is an instance of it.

Before showing the validity of the construction, we introduce a definition and prove a lemma.

Let $f^m(u)$ denote $f(f(\ldots f(u)\ldots))$ where f occurs m times, for $m \geq 0$.

5.4.1. Lemma.

σ unifies $\{\lambda u.\phi(\lambda u.u,u), \lambda u.u\}$ iff $\exists m[\sigma(\phi) = \lambda fu.f^m(u)]$.

Proof: (If) This is trivial since

$$\{\phi \leftarrow \lambda fu.f^m(u)\}$$

unifies

$$\{\lambda u.\phi(\lambda u.u,u), \lambda u.u\} \quad \text{for } m \geq 0.$$

(Only if) Rather than defining and using the method given by Huet [14], we give a self-contained proof based on analyzing the symbols which occur in $\sigma(\phi)$.

Let $\phi \leftarrow \lambda fu.e_0$ belong to σ where $e_0 = s_0(e_1,\ldots,e_n)$ and s_0 is a symbol, then

$$\begin{aligned}\sigma(\lambda u.\phi(\lambda u.u,u)) &= \lambda u.\{f \leftarrow \sigma(\lambda u.u), u \leftarrow \sigma(u)\}(s_0(e_1,\ldots,e_n))\\ &= \lambda u.\{f \leftarrow \lambda u.u,\ u \leftarrow u\}(s_0(e_1,\ldots,e_n))\end{aligned}$$

since by renaming, we assume σ does not pertain to u. Now, s_0 cannot be a constant since the right side of the equation would be of the form $\lambda u.s_0(\ldots)$ which cannot equal $\lambda u.u$.

Similarly, s_0 cannot be a variable different from f or u. If s_0 is the variable, u, then indeed:

$$\sigma(\lambda u.\phi(\lambda u.u,u)) = \lambda u.\{f \leftarrow \lambda u.u,\ u \leftarrow u\}(u) = \lambda u.u.$$

If s_0 is the variable, f, which has degree one, then

$$\sigma(\lambda u.\phi(\lambda u.u,u)) = \lambda u.\{f \leftarrow \lambda u.u,\ u \leftarrow u\}(f(e_1))$$
$$= \lambda u.\{f \leftarrow \lambda u.u,\ u \leftarrow u\}(e_1).$$

Let $e_0 = s_1(e_1', \ldots, e_{n_1}')$ then by repeating the above argument either s_1 is the variable u or $s_1 = f$ and $e_1 = s_2(e_1'', \ldots, e_{n_2}'')$, etc. Collecting these results, we have:

$$\phi \leftarrow \lambda fu.f^m(u) \text{ belongs to } \sigma, \text{ for some } m \geq 0.$$

We could give a rigorous proof by induction on m in

$\{\phi \leftarrow e\}$ unifies $\{\lambda u.\phi(\lambda u.u,u),\ \lambda u.u\}$

and length$[e] = m$ implies $e = \lambda fu.f^m(u)$.

5.4.2. Validity proof. We will now prove

$$\exists m_1 \ldots \exists m_N\ P[m_1, \ldots, m_N] = Q[m_1, \ldots, m_N]$$

iff

$$\{e_1, e_2\} \text{ is unifiable.}$$

Now $\{e_1, e_2\}$ is unifiable

iff $\exists\sigma\ \sigma$ unifies $\{e_1, e_2\}$

iff $\exists\sigma\ \sigma$ unifies $\bigcup_{i=1}^{N} \{\{\lambda u.\ \phi_i(\lambda u.u,u), \lambda u.u\}\}$

$\cup\ \{\{\lambda fu.E_P,\ \lambda fu.E_Q\}\}$

(by the definition of e_1 and e_2 and simple properties of substitutions (C is a constant))

iff $\exists\sigma\ [\forall i\ \sigma$ unifies $\{\lambda u.\phi_i(\lambda u.u,u),\ \lambda u.u\}$ &

σ unifies $\{\lambda fu.E_P,\ \lambda fu.E_Q\}$]

iff $\exists\sigma\ [\forall i \exists m_i\ \sigma(\phi_i) = \lambda fu.f^{m_i}(u)$ &

σ unifies $\{\lambda fu.E_P,\ \lambda fu.E_Q\}$]

iff $\exists m_1 \ldots \exists m_N\ \sigma^* = \{\phi_1 \leftarrow \lambda fu.f^{m_1}(u), \ldots, \phi_N \leftarrow \lambda fu.f^{m_N}(u)\}$

unifies $\{\lambda fu.E_P,\ \lambda fu.E_Q\}$.

We will now show, by induction on multinomials, that

$\sigma*(E_P) = f^P(u)$ where P is a multinomial in $m_1,\ldots,m_N$.

(1) $\sigma*(E_1) = \sigma*(f(u)) = f^1(u)$

(since $\sigma*$ does not pertain to f or u).

(2) $\sigma*(E_{m_i}) = \sigma*(\phi_i(\lambda u.f(u),u))$

(by the definition of E_{m_i}).

$= f^{m_i}(u)$ (by applying $\sigma*$ to ϕ_i).

for all $i=1,\ldots,N$.

(3) $\sigma*(E_{(P+Q)}) = \sigma*(E_P[\lambda u.f(u), E_Q])$

(by the definition of $E_{(P+Q)}$).

$= f^P(\sigma*(E_Q))$ (by hypothesis for E_P)

$= f^P(f^Q(u))$ (by hypothesis for E_Q)

$= f^{P+Q}(u)$.

(4) $\sigma*(E_{(P.Q)}) = \sigma*(E_P[\lambda u.E_Q,u])$ (by definition of $E_{(P.Q)}$)

$= (\sigma*(\lambda u.E_Q))^P(u)$ (by hypothesis for E_P)

$= (f^Q)^P(u)$ (by hypothesis for E_Q)

$= f^{P.Q}(u)$.

Hence $\{e_1,e_2\}$ is unifiable iff $\exists m_1\ldots\exists m_N$ such that:

$\sigma*$ unifies $\{\lambda fu.E_P, \lambda fu.E_Q\}$

iff $\sigma*(\lambda fu.E_P) = \sigma*(\lambda fu.E_Q)$

iff $\lambda fu.\sigma*(E_P) = \lambda fu.\sigma*(E_Q)$

iff $\lambda fu.f^P(u) = \lambda fu.f^Q(u)$ (by the above result)

iff $P = Q$.

We have thus proved the validity of the construction.

Note that the result is independent of the actual bracketing chosen in the multinomials. Because, if P and Q are "equivalent" multinomials (that is, $P=Q$ is true in number theory) then

$$\sigma^*(E_P) = f^P(u) = f^Q(u) = \sigma^*(E_Q).$$

APPENDIX

We will briefly describe a recent linear first-order unification algorithm discovered by Paterson and Wegman [22]. This is of interest because it theoretically improves our practically linear time bound and also because of similarities with our algorithm. Succintly, the linear algorithm combines our transformational and sorting stages.

In order to present the linear algorithm, we first present an algorithm, which determines if a digraph is acyclic (contains no circuits), based upon the linear topological sorting algorithm [18]. Our sorting algorithm is based on this algorithm, and also is the integral part of the linear algorithm.

First, let us define a <u>root vertex</u> of a digraph as a vertex whose only edges are outgoing. Using this notion, we present the algorithm, ACYCLIC, which determines if a digraph G is acyclic.

```
algorithm ACYCLIC(G):
begin
   while G contains a root vertex, r, do:
   Delete vertex r and all of its outgoing edges from G ;
   if G contains no vertices
   then return (true)
   else return (false) ;
end.
```

The linear unification algorithm constructs an equivalence relation ("≡") on the vertices of the tree representation of the expressions to be unified. This is similar to our construction of the partition in section

2.4.1. This relation is "propagated" from terms to their corresponding arguments. (Refer to the hereditary property in section 2.5.2.) However, the propagation is performed whilst sorting the digraph induced by the equivalence relation.

Before we present the algorithm, UNIFIABLE, which is the basis for the linear algorithm, let us define a root class to be an equivalence class of vertices (of the tree representation of expressions) containing only root vertices.

```
algorithm UNIFIABLE($e_1$,$e_2$):
begin
   Set $e_1 \equiv e_2$ ;
   while there exists a root class, R, do:
   begin
      if R contains two terms beginning with
         different function symbols
      then return (false) ;
      if R contains a term
      then let $e'=f(e'_1,\ldots,e'_n)$ be a term in R ;
      for all expressions $e''$ ($\neq e'$) in R do:
      if $e''$ is a variable
      then print $\{e'' \leftarrow e'\}$
      else begin
              let $e''=f(e''_1,\ldots,e''_n)$
              comment: propagate the equivalence relation;
              Set $e'_1 \equiv e''_1,\ldots,e'_n \equiv e''_n$ ;
           end ;
      Delete the vertices in R and their outgoing edges ;
   end ;
   if there are no classes remaining
   then return (true)
   else return (false) ;
end.
```

We will not give details of the actual implementation which results in a linear algorithm, but will point out a few salient features. Our algorithm is practically linear because we perform FINDs and MERGEs on a partition. The reason why the relatively expensive FINDs (which are responsible for the non-linearity) are performed is because we frequently merge two classes of the partition and then find a term from each in order to propagate the equivalence relation. In contrast, in the linear algorithm, by using stacks, equivalence classes are constructed by adding only one element to them at a time. Further, only when a class has been selected as a root class, is the equivalence relation propagated and then the root class is deleted.

Practically, it would be difficult to choose between our practically-linear algorithm and the linear algorithm. A more useful criterion may be based on the "on-line" measure of complexity. The reason for this is that, in some theorem-proving contexts, it is appropriate to repeatedly unify a set of expressions incrementally. That is, having successfully unified a set of expressions, E, we then add a pair of expressions to E and unify this union, E'. Consequently, we are intereseted in the additional time required relative to the size of the additional pair.

It appears that our practically linear algorithm is more suitable for this incremental type of processing than the linear algorithm. We cannot theoretically justify this, since, for example, an efficient "on-line" digraph sorting algorithm is unknown, however we can indicate our reasoning. Already our transformational stage has the incremental property since it is based upon an on-line processing of equivalence relations. In our algorithm, having unified a set of expressions, an additional pair of expressions is easily added to S and further transformations made. If we do not detect failure of unification, it is necessary to re-sort the new induced digraph. In contrast, in the linear algorithm, since the transforming and sorting occur simultaneously, it is necessary to unify E' by reapplying the entire algorithm, without being able to use information about the prior unification of E. Specifically, having unified E, it is difficult to quickly merge two classes in E', because, in the linear implementation the expressions in a class are identified by labels. Due to the sorting process during the unification of E, these labels, once assigned, are never changed. However, if we have to later merge two classes due to the additional pair of expressions, the relabelling will be time consuming.

Finally, we will illustrate the linear algorithm by attempting to unify the pair of expressions :

$\{P(F(H(A,w,B),x),\ z,\ G(F(u,x),w)),\quad P(y,\ G(y,A),\ z)\}$.

The tree representation of these expressions is given in Figure A.1., where an (undirected) edge joins the two vertices corresponding to the two expressions to be unified. Generally, the connected components given by such edges represent the equivalence classes.

Initially, there is only one class and this is a root class. The equivalence relation is propagated to the corresponding arguments, that is, edges are drawn between the vertices corresponding to the first arguments: F(H(A,w,B),x) and y; similarly between z and G(y,A) and between G(F(u,x),w) and z. After deleting the root class and its outgoing edges the situation is given by Figure A.2.

Since the vertex associated with y is not a root, the unique root class is [z, G(F(u,x),w), G(y,A)]. The equivalence relation is propagated so that F(u,x) and y are made equivalent and w and A are also. After deletion, Figure A.3. is obtained.

Now the only root class is [F(H(A,w,B),x), y, F(u,x)] and after propagation and deletion, Figure A.4. is obtained.

Now, either [x] or [u, H(A,w,B)] is a root class; we choose the latter which is quickly disposed of since it contains only one term. Figure A.5. illustrates the current state.

Both [x] and [w, A] are now root classes; by choosing [x] we obtain Figure A.6.; by next choosing [w, A] we finally get Figure A.7. Since there are no classes left, unification succeeds.

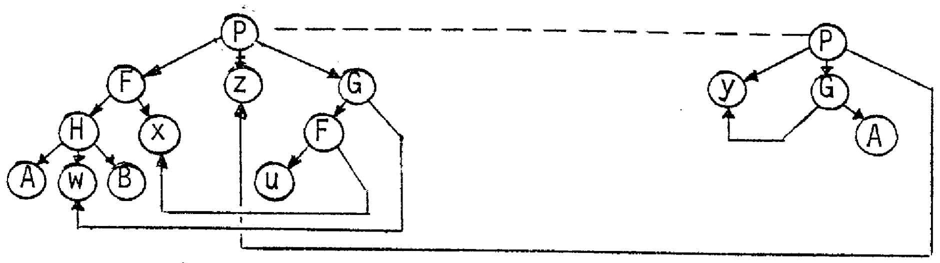


Figure A.1.

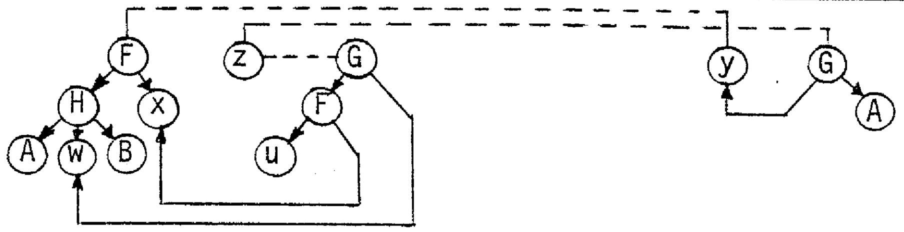


Figure A.2.

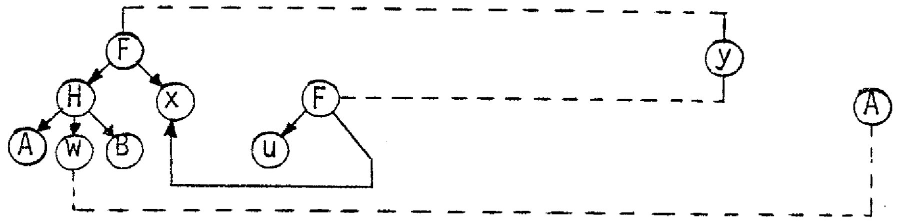


Figure A.3.

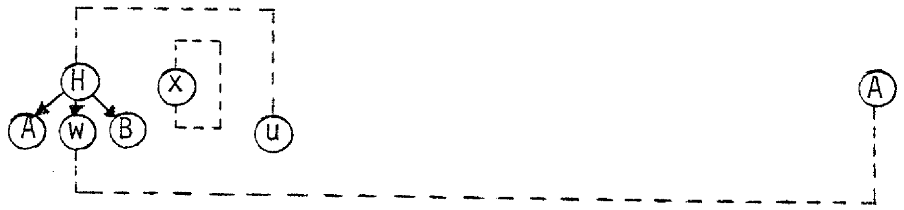


Figure A.4.

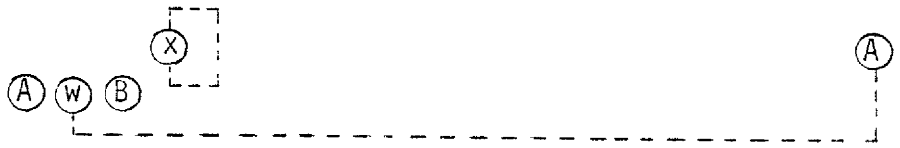


Figure A.5.

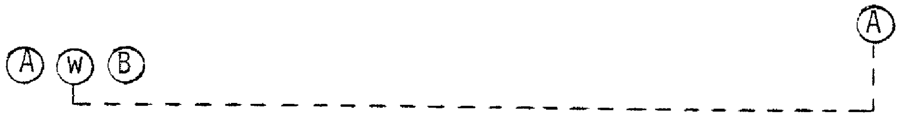


Figure A.6.

Figure A.7.

Illustrating the linear unification algorithm. Dotted lines represent undirected edges.

C O N C L U S I O N

We have examined several unification problems from a computational complexity viewpoint and we see that their complexities range from linear for the first- order unification problem to undecidable for the third-order second-degree unification problem with the conjectured exponential complexity of second-order instantiation in between.

Since our first-order unification algorithm is based upon maintaining a partition on-line, and since the best algorithm to do this, using the technique of path-compression on balanced trees, is practically linear, so also is ours. The existence of a linear algorithm for maintaining an on-line partition remains an open question.

As mentioned in the appendix, the context in which unification appears can determine the usefulness of a unification algorithm. Currently, research is under way into a certain linear-resolution scheme in theorem-proving, which uses unification in a global manner rather than the local philosophy of previous deduction systems [9]. In addition to the useful incremental property referred to in the appendix, such a system incorporates a "back-tracking" search: if a certain line of reasoning is to be discontinued, it may be necessary to back-track to some earlier decision point. This requires "undoing" parts of the unification process. Due to our representation of a partition of expressions by trees and the simple merging of classes in the partition, this "undoing" is efficiently performed. In contrast, due to the identification of classes by labels in the linear algorithm, this "undoing" cannot be easily performed.

There remain a few gaps in the classification of some unification problems according to their complexity. The decidability of second-order unification is an open problem, even for the first-degree case. This problem may be related to certain algebraic problems of specialized semigroups. For the second-degree case there appears to be no corresponding algebraic theory dealing with binary trees. The decidability of the third-order first-degree problem is also unresolved.

The instantiation problem, in which one expression contains no variables, has been solved completely for first-order. It is easy to design a linear algorithm for solving the first-order instantiation problem, based on a simple examination of the tree representations of the expressions. A corollary of our result for the first-order unification is that our algorithm is linear in the special case when one expression contains no variables.

The second-order instantiation problem we examined had no restriction on the degree of the expressions; however our construction can be easily modified, with some loss of clarity, so that the degree is at most two. The first-degree case is trivial. Further refinements may be suggested by restricting the number of and degrees of variables and constants occurring in the expressions to be unified.

R E F E R E N C E S

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

[2] Allen J. and Luckham D. (1969)
"An interactive theorem-proving program" in Machine Intelligence 5, 321-336.

[3] Baxter L.D. (1973)
"An efficient unification algorithm," Research Report CS-73-23, Department of Computer Science, University of Waterloo.

[4] Baxter L.D. (1976)
"A practically linear unification algorithm," Research Report CS-76-13, Department of Computer Science, University of Waterloo.

[5] Chang C. and Lee R.C. (1973)
Symbolic Logic and Mechanical Theorem Proving, Academic Press.

[6] Church A. (1940)
"A formulation of the simple theory of types," Journal of Symbolic Logic 5, 1, 56-68.

[7] Church A. (1941)
"The calculi of λ-conversion," Annals of Mathematical Studies, No. 6, Princeton University Press.

[8] Cook S.A. (1971)
"The complexity of theorem-proving procedures," in

Proceedings of the Third Annual Symposium on Theory of Computing, 151-158.

[9] Cox P.T. and Pietrzykowski T. (1976)
"A graphical deduction system," Research Report CS-76-35, Department of Computer Science, University of Waterloo.

[10] Elspas B., et alia (1972)
"An assessment of techniques for proving program correctness," Computing Surveys, 4, 2.

[11] Gould W.E. (1966)
"A matching procedure for ω-order logic," Ph.D. Thesis, Princeton University.

[12] Guard J.R. et alia (1969)
"Semiautomated mathematics," Journal of the Association for Computing Machinery, 16, 1, 49-62.

[13] Huet G.P. (1973)
"The undecidability of unification in third-order logic," Information and Control 22, 3, 257-267.

[14] Huet G.P. (1975)
"A unification algorithm for typed λ-calculus," Theoretical Computer Science 1, 27-57.

[15] Huet G.P. (1976)
"Simplification sets," a seminar on Automatisches Beweisen, Oberwolfach.

[16] Jensen D. and Pietrzykowski T. (1973)
"Mechanizing ω-order type theory through unification,"

Research Report CS-73-16, Department of Computer Science, University of Waterloo.

[17] Karp R. (1972)
"Reducibility among combinatorial problems," Computer Science Technical Report 3, University of California, Berkeley.

[18] Knuth D.E. (1968)
The Art of Computer Programming, Volume I: Fundamental Algorithms, Addison-Wesley.

[19] Lucchesi C.L. (1972)
"The undecidability of the unification problem for third-order language," Research Report CSRR 2059, Department of Computer Science, University of Waterloo.

[20] Matijasevic Y. (1970)
"Enumerable sets are diophantine," Dokl. Akad. Nauk SSSR, 191, 279-282.

[21] Newell A., Shaw C. and Simon H. (1956)
"The Logic Theory Machine: a complex information processing system," Institute of Radio Engineers Transactions on Information Theory, IT-2, 61-79.

[22] Paterson M.S. and Wegman M.N. (1976)
"Linear Unification," Proceedings of the Eighth Annual Association for Computing Machinery Symposium on Theory of Computing, 181-186.

[23] Pietrzykowski T. (1971)
"A complete mechanization of second-order logic," Jour-

nal of the Association for Computing Machinery, 20, 2, 333-364.

[24] Pietrzykowski T. and Jensen D. (1972)
"A complete mechanization of ω-order type theory," Association for Computing Machinery National Conference, 82-92.

[25] Post E. (1940)
"Absolutely unsolvable problems and relatively undecidable propositions," in The Undecidable, (M. Davis editor), Raven Press.

[26] Robinson G.A. and Wos L.T. (1969)
"Paramodulation and theorem-proving in first-order theories with equality," in Machine Intelligence 4, American Elsevier, 135-150.

[27] Robinson J.A. (1965)
"A machine-oriented logic based on the resolution principle," Journal of the Association for Computing Machinery, 12, 1, 23-41.

[28] Robinson J.A. (1970)
"Computational logic: the unification computation," in Machine Intelligence 6, American Elsevier, 63-72.

[29] Robinson J.A. (1976)
"Fast Unification," a seminar on Automatisches Beweisen, Oberwolfach.

[30] Siekman J. (1975)
"String unification," Research Report CSM-7, Computing

Centre, University of Essex.

[31] Tarjan, R.E. (1975)
"Efficiency of a good but not linear set union algorithm," Journal of the Association for Computing Machinery, 22, 2, 215-225.

[32] Van Emden M.H. (1975)
"Programming with resolution logic," in Machine Representation of Knowledge ,(eds: E.W.Elcock & D.Michie)

[33] Venturini-Zilli M. (1975)
"Complexity of the unification algorithm for first-order expressions," Research Report, Consiglio Nazionale Delle Ricerche Istituto per le applicazioni del calcolo.

[34] Winterstein G. (1976)
"Unification in second-order logic," Research Report, Fachbereich Informatik, Universitat Kaiserslautern.