THESIS

TYPE INERENCE
WITH
OVERLOADING
USING AN
ATTRIBUTE GRAMMAR

by
Bruce James Bull
March, 1994

Thesis Advisor: Dennis M. Volpano
Thesis Second Reader: Craig W. Rasmussen

Approved for public release; distribution is unlimited.

94-22597
Interactive programming environments for languages offer many advantages over traditional batch-oriented ones, such as immediate static analysis. One form of analysis is type checking, yet type checking in this setting for languages with common features like overloading has received little attention.

We implement an interactive type checker for the polymorphic type system of ML with overloading. The implementation was produced automatically from an attribute grammar using the Synthesizer Generator, an attribute evaluator generator. Type inference then is accomplished via attribute evaluation so that if the evaluation is done incrementally, then type inference becomes incremental as well.
ABSTRACT

Interactive programming environments for languages offer many advantages over traditional batch-oriented ones, such as immediate static analysis. One form of analysis is type checking, yet type checking in this setting for languages with common features like overloading has received little attention.

We implement an interactive type checker for the polymorphic type system of ML with overloading. The implementation was produced automatically from an attribute grammar using the Synthesizer Generator, an attribute evaluator generator. Type inference then is accomplished via attribute evaluation so that if the evaluation is done incrementally, then type inference becomes incremental as well.
# TABLE OF CONTENTS

I. INTRODUCTION ................................................. 1  
   A. IMPLEMENTING $W_o$ .................................. 3  
II. TYPE SYSTEMS .................................................. 4  
   A. WHAT IS A TYPE? ........................................ 4  
   B. POLYMORPHISM ......................................... 5  
   C. OVERLOADING ......................................... 7  
III. THE $ML$ TYPE SYSTEM AND OVERLOADING .............. 10  
   A. $ML_o$ .................................................. 12  
IV. TYPE INFERENCE IN SYSTEM $ML_o$ ...................... 16  
   A. PARAMETRIC OVERLOADING AND SATISFIABILITY ........ 20  
   B. SATISFIABILITY ALGORITHM .......................... 24  
V. IMPLEMENTATION OF $W_o$ .................................. 27  
   A. THE ROLE OF ATTRIBUTE GRAMMARS .................. 27  
VI. CONCLUSIONS .................................................. 39  
   A. FUTURE WORK ......................................... 40  
LIST OF REFERENCES ............................................. 41  
INITIAL DISTRIBUTION LIST ................................. 44
LIST OF FIGURES

2.1 Polymorphic length function ........................................ 6
2.2 Pascal min function .................................................... 6
2.3 Ada generic min function ............................................... 7

3.1 System $ML_0$ .......................................................... 13

4.1 Algorithm $W_o$ .......................................................... 17
4.2 Infinite and recursive overloadings .................................. 21
4.3 Type constructors of various arities .................................. 24

5.1 Context-free grammar for $ML_0$ type inference ................... 29
5.2 Inherited and synthesized attributes of implementation grammar .. 30
5.3 Attribute equations for $ML_0$ type inference ..................... 31
5.4 A partial derivation tree and dependence graph for $pr (x, \lambda y . \lambda z . y z)$ 32
5.5 Implementation initial screen ........................................... 34
5.6 An assumption set defined ............................................. 35
5.7 Type inference of three expressions .................................. 37
5.8 Representation of attribute $typeGrammar$ .......................... 38
ACKNOWLEDGMENTS

I owe an immense debt of gratitude to Dr. Dennis Volpano for offering timely and valuable assistance and insight into every aspect of my thesis. His tireless pursuit of excellence and high ideals have been an inspiring example. He has made my efforts at the Naval Postgraduate School both rewarding and memorable.

I wish to also thank Dr. Craig Rasmussen for lending his time and expertise in ensuring the correctness of this thesis. His ability to make difficult concepts understandable and genuine concern have greatly benefited many students at NPS, including myself.

Most of all, I would like to express my gratitude to my family. To my wife Tacie; for her encouragement, thoughtfullness, and untiring patience. To my children; for providing reminders of what is truly important in life.
I. INTRODUCTION

In this thesis, we assume the reader is familiar with basic type theory and its associated notational conventions. We also assume a general familiarity with the concepts and notation of the lambda-calculus. A comprehensive presentation of these concepts can be found in the texts of Thompson [Tho91] and Gunter [Gun92].

The advantages of interactive programming environments to increase programmer effectiveness and maximize utilization of system resources are significant. For example, during program development, extensive context-sensitive type checking is a valuable tool. The immediate recognition of type errors at this stage could yield vast improvements to the quality and reliability of today's software products. Valuable system resources would be preserved through decreased waste due to unnecessary re-compilations. Perhaps more significantly, the advantages of providing an environment where programmers can focus on the fundamental aspects of a problem with a much higher degree of continuity are clear.

The study of type inference is integral to this effort. Though significant advances have been made in this research area, further work needs to be done. This thesis considers a suitable type system for implementing a polymorphic programming language with overloading. Utilizing this type system, an implementation is produced that performs incremental type inference in an interactive environment.

One can argue that system ML represents the current state of the art in type systems. It is a polymorphic type system but prohibits the use of overloading. Yet the need for overloading in programming languages is well known. Current imperative languages, such as Ada and C++, and even the functional language standard ML, allow an identifier to represent different types but the resulting programs merely
contain monomorphic instances overloaded on an identifiers name. A process called overloading resolution is required to assign a particular type to an identifier based on its context. Consider the following expressions, where + is defined over integers and reals:

(a) $1 + 2$
(b) $1.0 + 2.0$

What is the type of +? We only know that it can have the type $int \rightarrow int \rightarrow int$ or the type $real \rightarrow real \rightarrow real$. But we can reliably assign neither of these types to + without first examining its context. In a polymorphic language, like $ML$, we can assign + the type $\forall \alpha . \alpha \rightarrow \alpha \rightarrow \alpha$ but this results in + having too many types. On the other hand, if we assign + the type $real \rightarrow real \rightarrow real$ we preclude its use in expression (a). We will examine these issues in more detail in Chapter II.

What is needed is a means to express a type for + which encompasses all of its possible types and no more. We can do this with the use of constrained type schemes. We can then assign to any occurrence of +, regardless of its context, the type $\forall \alpha$ with (+ : $\alpha \rightarrow \alpha \rightarrow \alpha$). $\alpha \rightarrow \alpha \rightarrow \alpha$. This means that + can assume any finite type $\alpha \rightarrow \alpha \rightarrow \alpha$, with $\alpha$ instantiated to any particular type for which + is defined.

Using the concept of constrained type schemes, an extension to system $ML$ has been developed incorporating overloading called $ML_o$. The associated type inference algorithm $\omega$ infers principal types for expressions in $ML_o$. It turns out that, unless we place restrictions on the kinds of overloading we can express using constrained type schemes, typability in $ML_o$ is undecidable. In Chapter IV we consider a form of overloading called parametric overloading which makes typability in $ML_o$ decidable and present an algorithm which determines satisfiability of constraints with respect to a parametric assumption set.
A. IMPLEMENTING $W_0$

$W_0$ performs batch type inference. In this respect, it is unsuitable for direct incorporation into a useful interactive programming environment. What is needed is an incremental approach to type inference which will provide immediate feedback to the programmer when type errors are encountered.

One might attempt to rewrite $W_0$ to achieve incremental type inference. Our approach is to utilize the formalism of attribute grammars to express $W_0$. In this setting type inference is performed via attribute evaluation. As expressions are input a corresponding change is reflected in the attribution. If we are able to perform attribute evaluation incrementally, type inference can also be done incrementally. Furthermore, it is implicit in the formalism.

We present an implementation of $W_0$ utilizing an attribute grammar in SSL, the language of the Synthesizer Generator of Grammatech. It is an attribute evaluator generator that takes as input a set of attribute equations and returns as output an attribute evaluator, or in our case, a type-checker. By utilizing the Synthesizer Generator for our implementation we are not only able to produce an attribute evaluator, but one in which attribute evaluation is done incrementally. As a result, we are able to achieve both attribute evaluation and type inference in an incremental setting. Chapters IV and V discuss details of the implementation and the algorithms used.
II. TYPE SYSTEMS

The concept of type systems in programming languages deals with a set of rules which, when applied to terms of a language, produce types for those terms. The notion of types in programming languages has been given steadily increasing importance over the past several years. It is clear that languages with rich type classes offer programmers more flexibility in modeling real-world objects. Yet, there remains a significant lack of consensus as to what types are. As consensus in this area is critical to the successful application of type theory to practical implementations of new programming environments, this chapter outlines the most important aspects of type systems and their application to this thesis.

A. WHAT IS A TYPE?

When discussing types, there exists a tendency to confuse the distinction between implementation issues and the underlying nature of types in general. Actual machines, for example, provide relatively few types (i.e. integers, floating-point numbers, pointers, etc.). The implementation of types in a high-level language, while posing some very real problems in the area of compiler design, should remain distinct from a discussion of type correctness in the higher context of the meaning of types. With reference to implementation issues, referred to as Reductionist type correctness, Smith states:

The key issue is how to protect the representation from misuse. [Smi91]

In this thesis, we will not concern ourselves with the reductionist view of types. Rather, we will view a type as an algebra, a set of values and operations such that
the set is closed under these operations. For example, type int is the set of integers together with the usual arithmetic operations, but the set of natural numbers and the predecessor operation do not form a type. This view gives us a fundamental basis from which to discuss the meaning and usage of types in programming languages unencumbered by implementation issues. Operations of an algebra are axiomatized, providing then a semantics that one can use to reason about programs in which they occur. In order to use the axioms, however, it may be necessary to restrict the types of certain program arguments to the algebras in question. For example, if we are to prove that a function adds 1 to its argument then we might wish to fix the type of its argument to int, say. For some programs, though, reasoning can proceed without fixing argument types. Such programs are called polymorphic.

B. POLYMORPHISM

Polymorphic means to have many forms. With respect to programming languages, this refers to programs or terms which have many types, or can operate on values of many types. Perhaps more intuitively, we can state that the purpose of polymorphism is to allow programs which use a single name to operate on many different types of inputs and, perhaps, produce different types of output.

We will first be concerned with a form of polymorphism called parametric polymorphism, where polymorphic entities can be described by a universally quantified formula with all quantification at the outermost level (e.g. $\forall \alpha. \alpha \rightarrow \alpha$). In Figure 2.1, we give an example of a function, length, defined in a generic polymorphic programming language. We can ascribe to length type $\forall \alpha. list(\alpha) \rightarrow \text{int}$. It's meaning is a function which given a list computes its length.

Languages which do not support polymorphism put unnecessary restrictions on the use of a function. Consider the Pascal program in Figure 2.2. Procedure min has the type: $\text{int} \rightarrow \text{int} \rightarrow \text{int}$. Yet there is nothing inherent in min which depends
function length(x)
{
    if not null(x) then
        1 + length(tail(x))
    else
        0
}

Figure 2.1: Polymorphic length function

on integer. Replacing integer with char would yield a correct Pascal program with meaning corresponding to the lexicographic ordering of characters.

It is not uncommon for the claim to be made that Ada is a polymorphic programming language, as in [ASU86]. One might argue that it is, but really only weakly so. Through the use of generics, one can define a template for representing what appears to be a polymorphic function. In the example of Figure 2.3, one might wish to ascribe the type ∀α.α → α → α to the Ada function min within the generic package MIN_PKG. This would indicate that min is defined over all instantiations of α, including int and char. This is obviously not the case, for a generic package cannot be used directly in Ada. It must first be instantiated for a particular type so that it can be properly type checked. Though the language provides constructs for expressing polymorphism, the resulting compiled program merely contains monomorphic instances of the function overloaded on the identifier min. Research into providing polymorphism in an imperative language is ongoing [Car87].

procedure min(x,y : integer);

begin
    if x < y then
        return(x)
    else
        return(y)
end

Figure 2.2: Pascal min function
generic
type ITEM is private;
  with function "<"(left,right : ITEM)
  return BOOLEAN is <>
package MIN_PKG is
  function min(x,y : ITEM) return ITEM is
  begin
    if x < y then
      return(x)
    else
      return(y)
    end min;
  end MIN_PKG;

Figure 2.3: Ada generic min function

It is clear that parametric polymorphism is a desirable property of practical programming languages. Yet, in practice, situations arise where parametric polymorphism alone cannot provide us with the means to express certain types adequately. Consider a polymorphic type for min in Figure 2.2. Clearly it is meaningful for multiple types. However, if we ascribe the type $\forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$ to min, terms without meaning, such as $\min(true, false)$, become typable. It can be seen that min depends on "<" being defined over its parameters. What is needed is the ability to restrict use of min to input types whose values are partially ordered. In other words, we need to be able to overload "<" so that min is polymorphic yet bounded in the types of arguments to which it can be applied, a form of bounded polymorphism.

C. OVERLOADING

The common view of overloading is stated as follows:

An overloaded symbol is one that has different meanings depending on its context [ASU86] (emphasis added).
This process of determining the meaning of an expression by examining its context is called overloading resolution. This is, in fact, the usual way of treating overloaded symbols in a program; demanding that the local context of an overloaded symbol determine a particular overloading to be used at each occurrence. This kind of treatment is used even in the polymorphic language ML. In fact, any overloading that requires overloading resolution to determine its meaning is termed an incoherent overloading and gives rise to potential semantic ambiguity. For example

(2.1) \[
\begin{cases}
  * : \text{real} \to \text{real} \to \text{real}, \\
  * : \forall \alpha. \text{matrix}(\alpha) \to \text{matrix}(\alpha) \to \text{matrix}(\alpha) 
\end{cases}
\]

is an incoherent overloading of the operator * where * stands for real multiplication and matrix multiplication.

Consider a term \( \lambda x. \lambda y. x * y \). We can infer two different types for it: \( \text{real} \to \text{real} \to \text{real} \) and \( \forall \alpha. \text{matrix}(\alpha) \to \text{matrix}(\alpha) \to \text{matrix}(\alpha) \). We must apply the process of overloading resolution to determine the meaning of the term.

A more desirable form of overloading, called coherent overloading, arises when an overloading is constructed in such a way that its various instances share a common semantics. In this case, overloading resolution is not required to ascribe a unique meaning to terms. It’s meaning is uniquely determined from an inspection of the axioms for the operators occurring in a term. For example, suppose * is commutative. We can readily see that our overloading in (2.1) is incoherent. For, although we can derive from (2.1) that \( \lambda x. \lambda y. x * y \) has type \( \text{real} \to \text{real} \to \text{real} \) and \( \forall \alpha. \text{matrix}(\alpha) \to \text{matrix}(\alpha) \to \text{matrix}(\alpha) \), matrix multiplication is not commutative. If we replace our second assumption on * with * : \( \text{int} \to \text{int} \to \text{int} \) with the meaning of integer multiplication, the overloading now becomes coherent for both integer and real multiplication are commutative. So we know that regardless of the types of \( x \) and \( y \), the function is commutative.
As will be seen, while our implementation of $W_\circ$ does not prohibit the introduction of incoherent overloadings, our assumption is that all overloadings are coherent. If this assumption is invalid with respect to a particular overloading, types will still be correctly inferred for expressions involving that overloading. However, the guarantee that the meaning of such an expression is uniquely determined is lost.

Surprisingly, it is common in current languages to introduce incoherent overloadings regardless of the potential for semantic ambiguities. In Ada, for example, the operator “/” is overloaded with different meanings of integer and floating-point division.

Overloadings allowed in most languages, including: Ada, C++, and standard ML, are restricted to being finite. In the MLo type system this restriction is lifted. For example, we can represent an infinite overloading over lists under equality as: $\forall \alpha$ with $\vDash : \alpha \rightarrow \alpha \rightarrow bool$. list($\alpha$) $\rightarrow$ list($\alpha$) $\rightarrow$ bool. In this case, if $=$ has an instance at $\tau \rightarrow \tau \rightarrow bool$, then it also has an instance at list($\tau$) $\rightarrow$ list($\tau$) $\rightarrow$ bool.
III. THE ML TYPE SYSTEM AND OVERLOADING

In this chapter we consider an extension of a Curry-style typed lambda calculus ($\Lambda_0$) with type schemes called System ML. As mentioned previously, a type scheme represents parametric polymorphism, implying that all quantifiers must be outermost, or shallow. Research aimed at removing this restriction is described in [Lei83, McC84, KT90].

A free identifier may be denoted as having infinitely many types via a type scheme. For instance, the primitive LISP operation $\text{hd}$ may be given the type: $\forall \alpha. \text{seq}(\alpha) \rightarrow \alpha$ which would indicate that for any choice of $\alpha$, say $\tau$, $\text{hd}$ has the type: $\text{seq}(\tau) \rightarrow \tau$.

System ML preserves the property of principal types; every typable term has a principal type, one that is more general than any other type derivable for the term. For instance, the term $\lambda f. \lambda x. f x, f$ and $x$ occurring free, would have as principal type $\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$. This is regarded as the most general typing for this expression. This means that any type whatsoever of $\lambda f. \lambda x. f x$ can be derived from the type $\forall \alpha. \forall \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$ by suitably instantiating $\alpha$ and $\beta$; formally, we say that all the types of $\lambda f. \lambda x. f x$ are instances of the principal type. The existence of principal types means that a type inference algorithm will always compute a unique "best" type for a program.

In order to retain principal types, lambda abstraction in System ML, as in $\Lambda_0$, is monomorphic. This means that lambda-bound identifiers within a $\lambda$-expression cannot be assigned multiple types. Consider the expression $(\lambda x. x(\lambda y. y)) \lambda z. x$. This expression is typable in System ML with principal type $\forall \alpha. \alpha \rightarrow \alpha$. This conforms to
the restriction on lambda abstraction since \( x \), while being able to assume infinitely many values, has polymorphic type \( \forall \alpha. \alpha \rightarrow \alpha \). The restriction is manifest when an attempt at self-application is made within a \( \lambda \)-expression. For example, a term such as \( (\lambda y. yy) \lambda x. x \) is illegal in System ML. Here, \( y \) must be able to assume two different types; \( (\alpha \rightarrow \alpha) \) and \( \alpha \) for some particular \( \alpha \). This results in the term \( \lambda y. yy \) having type \( \forall \alpha. (\forall \beta. \beta \rightarrow \beta) \rightarrow \alpha \), which is not a principal type.

In order to allow free identifiers denoting polymorphic values to be assigned multiple types, one uses the let construct. The above expression can then be represented as let \( y = \lambda x. x \) in \( yy \). This involves no inner quantification, since each instance of \( y \) is replaced with \( \lambda x. x \) in determining the type for \( yy \).

System ML, like \( \Lambda_0 \), has a decidable typability problem. In other words, if a type exists for a program (there may be more than one), the type inference algorithm will be able to infer a correct type for it. Conversely, if a type does not exist, the algorithm is capable of making that determination. System ML is also widely accepted and has been incorporated into mainstream languages like Standard ML [HMM86] and Miranda [Tur86]. Yet, an obvious and practical limitation exists in System ML that prohibits overloading by restricting the number of assumptions per identifier in a type assumption set to at most one. Milner himself makes the comment in his 1978 paper [Mil78] that allowing more than one assumption is desirable.

An extension to the ML type system has been developed called \( ML_4 \) [VoS91]. It retains principal types and allows overloading. Deviations from System ML include the introduction of constrained type schemes and modifications to the type instantiation and generalization rules. Many extensions of System ML have been proposed to incorporate overloading. Among these are the systems of [Kae88, CDO91, Smi91, Kae92, Jon92] and those related to the development of the functional programming language Haskell [WaB89, CHO92, NiP93]. All of these type systems share the no-
tion of a constrained type scheme in various forms. A critique of these type systems is given in [Vol93b].

A. $ML_0$

Given a set of type variables ($\alpha, \beta, \gamma, \ldots$) and a set of type constructors ($\text{int}$, $\text{real}$, $\text{bool}$, $\text{list}$, $\ldots$) of various arities, the set of unquantified types is defined by:

$$\tau ::= \alpha \mid \tau \rightarrow \tau \mid \chi(\tau_1, \ldots, \tau_n)$$

The set of quantified types or type schemes, then, is defined by

$$\sigma ::= \forall(\alpha_1, \ldots, \alpha_n) \text{ with } (x_1 : \tau_1, \ldots, x_m : \tau_m). \tau,$$

where $\alpha_1, \ldots, \alpha_n$ is the set of quantified variables of $\sigma$, $x_1 : \tau_1, \ldots, x_m : \tau_m$ is the set of constraints on $\sigma$, and $\tau$ is the body of $\sigma$. If there are no quantified variables, the "$\forall$" may be omitted. If there are no constraints, the "with" may be omitted. In our terminology, $\sigma$ will always be reserved to represent a type scheme, $\bar{\alpha}$ denotes an abbreviation for $\alpha_1, \ldots, \alpha_n$ and $C$ will be used to represent a list of constraints. The most general form of a type scheme is then:

$$\sigma ::= \forall \bar{\alpha} \text{ with } C. \tau$$

A substitution is a set of replacements for type variables applied simultaneously to all type variables. For example:

$$[\alpha_1, \ldots, \alpha_n := \tau_1, \ldots, \tau_n]$$

is a substitution where all of the $\alpha_i$'s are distinct. The substitution is applied to a type $\tau$ by simultaneously replacing all of the $\alpha_i$'s with the corresponding $\tau_i$'s. The application of substitution $S$ to type $\tau$ will be denoted by $\tau S$.

Two new type assignment rules, ($\forall$-intro) and ($\forall$-elim), are given in Figure 3.1; these represent extensions to System $ML$ developed to accommodate overloading. It should also be noted that if the constraint list $C$ is empty, these two extensions are identical to type generalization and instantiation in system $ML$ [Mil78, DaM82].
Consider a term $M = \lambda x. \lambda y. ((x \times x) = y)$ which contains free identifiers $\ast$ and $\text{=}$. and the following assumption set.

$$A = \left\{ \begin{array}{l}
\ast : \text{real} \rightarrow \text{real} \rightarrow \text{real}, \\
\ast : \text{int} \rightarrow \text{int} \rightarrow \text{int}, \\
\text{=} : \text{int} \rightarrow \text{int} \rightarrow \text{bool}, \\
\text{=} : \forall a \text{ with } \ast : a \rightarrow a \rightarrow a, \text{=} : a \rightarrow a \rightarrow \text{bool}, \text{list}(a) \rightarrow \text{list}(a) \rightarrow \text{bool} \\
\end{array} \right\}$$

Here we show a derivation of

$$A \vdash \lambda x. \lambda y. ((x \times x) = y) : \forall a \text{ with } \ast : a \rightarrow a \rightarrow a, \text{=} : a \rightarrow a \rightarrow \text{bool}, a \rightarrow a \rightarrow \text{bool}$$

in $ML_\omega$.

We are required to introduce assumptions about $\ast$ and $\text{=}$. in our derivation in order to arrive at a type for $M$. However, for our derivation to succeed, we need
to be able to discharge those assumptions via the ∀-intro rule. This ensures that $M$ can be derived from only our initial assumption set $A$. The ∀-intro rule deviates from generalization in system $ML$ in that it requires that the constraint set $C$ be derivable from the initial assumption set $A$. This ensures that $C$ is satisfiable with respect to $A$. In our derivation, we can see from (11) that satisfiability is achieved by substituting $int$ for $\alpha$. In general, there can be more than one finite type which satisfies this requirement. For example, if $= \text{were defined for reals, both int and real could be used in our substitution for } \alpha$. Conversely, it is not always possible to achieve satisfiability. For instance, if we removed the second assumption on $*$ from $A$, our derivation would end at (10). There would be no single substitution for $\alpha$ which could satisfy the overlapping constraint requirements in (11) and we would conclude that $M$ is untypable with respect to $A$.

This requirement for satisfiability of constraint sets ensures that the type system $ML_\alpha$ is sound. It is interesting to compare $ML_\alpha$ to a similar extension to system $ML$ proposed by Kaes [Kae88] based on type kinds, where a type kind is a universe of types over which a type variable may be quantified. It proposed a restricted form of overloading which is generally the same restriction adopted by $ML_\alpha$. However, this type system turns out to be unsound in that it does not enforce satisfiability of constraint sets as outlined above. This results in terms with multiple non-overlapping constraints being deemed typable in some instances. In the last example of the previous paragraph, for instance, the term $M$ would be deemed typable. On the other hand, the similar work of [CDO91], in an effort to relax the restrictions on overloading in Kaes type system, enforces satisfiability and hence remains sound.

We have shown, by example, the process required to determine the typability of a term in $ML_\alpha$. This process can be described as a modification to the concept, used in system $ML$, of strong type inference [Tiu90]. Formally, strong type inference says that a term $M$ is typable with respect to an assumption set $B$ if $A \vdash M : \sigma$ is
derivable for some type $\sigma$ and $B \subseteq A$. This criterion turns out to be less restrictive than required in the presence of overloading. We are free, under strong type inference, to choose any assumption set $A$ which contains $B$. Returning to our derivation, it can be seen that, in step (11), we would have the freedom to introduce any new assumptions we required in order to satisfy typability under strong type inference, resulting in untypable terms being deemed typable. Strong type inference relies on the premise that assumption sets may contain at most one assumption per identifier. This premise, of course, does not hold in $MLo$. We then can view typability in $MLo$ as being that of strong type inference with the requirement that $B = A$. In other words, $B \vdash M : \sigma$ must be derivable for some type $\sigma$. 
IV. TYPE INFERENCE IN SYSTEM $ML_o$

An algorithm, based on $W$ of system $ML$, has been developed for $ML_o$ named $W_o$ [Smi91]. In this chapter, we will discuss type inference utilizing $W_o$, which is given in Figure 4.1.

$W_o$ infers principal types for typable expressions in $ML_o$, failing on untypable expressions. Given assumption set $A$ and expression $e$, $W_o(A, e)$ returns $(S, B, \tau)$. $S$ is a substitution such that $AS \cup B \vdash e : \tau$ is derivable. $B$ represents a set of constraints on $A$, which describe dependencies associated with overloaded identifiers occurring in $e$, needed to arrive at a type for $e$. $W_o$, unlike $W$, utilizes the least common generalization ($LCG$) of an identifier overloaded in $A$. This concept, along with the function $\text{close}(A, B, \tau)$ and $\text{unify}(\tau, \tau')$, we will examine in some detail in this chapter.

The $LCG$ of an overloaded identifier can, perhaps, be best described by beginning with an example. Consider the identifier $\ast$, overloaded in $A$ with the assumptions $\ast : \text{int} \rightarrow \text{int} \rightarrow \text{int}, \ast : \text{real} \rightarrow \text{real} \rightarrow \text{real} \rightarrow \text{real}$ and $\ast : \text{int} \rightarrow \text{real} \rightarrow \text{real}$. We can see that all of these assumptions have in common second and third arguments which are identical. There is no common ground in their structure with respect to their first arguments. We can describe their common structure by the use of two quantified type variables, one for the first argument and another for the remaining two. We would then assign as the $LCG$ of $\ast$, $\forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow \beta$.

More formally, we can say that a common generalization of some set of finite types $\tau_1, \ldots, \tau_n$ is $\tau$ if we can apply some set of substitutions $S_1, \ldots, S_n$ such that $\forall i. \tau S_i = \tau_i$. We further say that $\tau$ is a least common generalization if, for any other generalization $\tau'$ of $\tau$, there exists a substitution $S$ such that $\tau' S = \tau$. We can
\( W_0(A, e) \) is defined by cases:

\[ e \text{ is } x \]

- If \( x \) is overloaded in \( A \) with \( \text{LCG } \forall \alpha. \tau \),
  \[
  \text{return } ([], \{x : \tau S\}, \tau S) \\
  \text{where } S = [\alpha := \beta] \text{ and } \beta \text{ are new}
  \]
- Else if \( (x : \forall \alpha \text{ with } C. \tau) \in A, \)
  \[
  \text{return } ([], CS, \tau S) \\
  \text{where } S = [\alpha := \beta] \text{ and } \beta \text{ are new}
  \]
- Else fail.

\[ e \text{ is } \lambda x.M \]

- Let \( (S, B, \tau) = W_0(A_x \cup \{x : \alpha\}, M) \) where \( \alpha \) is new
  \[
  \text{return } (S, B, \alpha S \rightarrow \tau). 
  \]

\[ e \text{ is } M N \]

- Let \( (S, B, \tau) = W_0(A, M) \)
- Let \( (S', B', \tau') = W_0(A S, N) \)
- Let \( S'' = \text{unify}(\tau S', \tau' \rightarrow \alpha) \) where \( \alpha \) is new
  \[
  \text{return } (SS'S'', BS'S'' \cup B'S'', \alpha S''). 
  \]

\[ e \text{ is } \text{let } x = M \text{ in } N \]

- Let \( (S, B, \tau) = W_0(A, M) \)
- Let \( (B', \sigma) = \text{close}(AS, B, \tau) \)
- Let \( (S', B'', \tau') = W_0(A_xS \cup \{x : \sigma\}, N) \)
  \[
  \text{return } (SS'', B'S' \cup B'', \tau'). 
  \]

Figure 4.1: Algorithm \( W_0 \)
extend this principle to constrained type schemes by applying the concept over the 
bodies of each constrained type scheme. Least common generalizations are discussed 
in [Rey70], which gives an algorithm for computing them.

Function unify of \( W_o \) performs first-order unification of terms in expressions. 
In essence, unify(\( \tau', \tau'' \)) returns a substitution \( S \) such that \( \tau' S = \tau'' S \) and fails if 
no such substitution exists. Formal discussions of unification are given by Knight 
and Robinson in [Rob65, Kni89].

Function close of \( W_o \) takes as input \((A, B, \tau)\) and returns a constrained type 
scheme for \( \tau \). This is accomplished, essentially, by applying the (\( \forall \)-intro) rule of 
ML\( _o \) to \( \tau \). Function satisfy within close checks for satisfiability of \( B \) with respect to 
\( A \). The issue of satisfiability turns out to be one of the more interesting problems in 
the ML\( _o \) type system. We will discuss this problem, therefore, in detail later in this 
chapter. Actually, there is latitude in how one computes the closure of a type in 
\( W_o \). A basic algorithm for close is given by Smith [Smi91] which is sufficient in support- 
ning his soundness and completeness proofs of \( W_o \), but leaves the critical issue 
of satisfiability somewhat unresolved. Our implementation of \( W_o \) uses an algorithm 
developed by Volpano which incrementally determines satisfiability as an expression 
is being constructed [Vol93a]. This approach allows us to detect certain type err- 
ors, with respect to constraints, earlier than the alternative approach of delaying 
satisfiability checks until the complete expression has been type checked.

We reproduce Volpano’s algorithm for close(\( A, B, \tau \)) here for the sake of com- 
pleteness:

1. Let \( V \) be the set of all finite types in \( B \). For any two types \( \tau_1 \) and \( \tau_2 \) in \( V \), 
define an undirected edge \((\tau_1, \tau_2)\) if types \( \tau_1 \) and \( \tau_2 \) share a type variable, and 
let \( E \) be the set of all such edges.

2. Let \( B' \) be the set of all constraints \( x : \tau' \) in \( B \) for which there is no type \( \tau'' \) 
such that \( \tau'' \) contains a variable free in \( A \) and there is a path from \( \tau' \) to \( \tau'' \) in 
(\( V, E \)).
3. If $B'$ is unsatisfiable under $A$ then fail.

4. Let $C$ be the set of all constraints $x : r'$ in $B$ for which there is a type $r''$ such that $r$ and $r''$ share a type variable and there is a path from $r'$ to $r''$ in $(V, E)$.

5. Return $(B - B', \forall \alpha$ with $C . \tau)$, where $\alpha$ are the type variables free in $C$ or $\tau$ but not $A$.

In steps (1) and (2) we define a graph which connects constraints in $B$ which share a type variable, and extract types from $B$ which do not overlap on a type variable. Set $B'$ then contains all of the constraints in $B$ which can be eliminated, provided they are collectively satisfiable with respect to $A$. If we assume as we do in our implementation, that the initial assumption set cannot contain free type variables, then in the final call to close we are guaranteed that all constraints in $B$ will be discharged. This approach allows us to perform satisfiability checks in an incremental manner. We do not eliminate a constraint from $B$ if it requires us to instantiate a type variable to some finite type; a subsequent term in the expression may require instantiation of that type variable, in which case we need to be able to ensure that previous overloading dependencies are satisfied. Consider the example, slightly modified, from [Vol93a]. If we have assumption set:

$$A = \begin{cases} b : bool \\ + : int \rightarrow int \rightarrow int, \\ * : int \rightarrow real \rightarrow real, \\ = : \forall \alpha. \alpha \rightarrow \alpha \rightarrow bool \end{cases}$$

recognizing that $+$ has $LCG \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha$, say we have the partial expression

$$\lambda x . \text{let } y = \lambda z . \text{pair}(z + z, z + x) \text{ in } < \text{exp}> .$$

where $<\text{exp}>$ represents a placeholder. $W_o$, in the process of computing a type for $y$, makes the call,

$$\text{close}(A \cup \{x : \alpha\}, B, \gamma \rightarrow (\gamma \times \alpha)),$$
where $B$ is the constraint set,

$$B = \{+ : \gamma \rightarrow \gamma \rightarrow \gamma, + : \gamma \rightarrow \alpha \rightarrow \alpha\}.$$ 

Function close determines that $B'$ is empty, since all constraints in $B$ share a type variable, $\gamma$. So $B$ is determined satisfiable, and close leaves $B$ intact in returning

$$(B. \forall \gamma \text{ with } B. \gamma \rightarrow (\gamma \times \alpha)).$$ 

Now, suppose we replace $<\text{exp}>$ with the term $x = b$. This determines the type of $x$ to be $\text{bool}$. $W_\gamma$ now makes its final call to close for the entire $\lambda$-expression as

$$\text{close}(A, B, \text{bool} \rightarrow \text{bool})$$

where

$$B = \{+ : \gamma \rightarrow \gamma \rightarrow \gamma, + : \gamma \rightarrow \text{bool} \rightarrow \text{bool}\}.$$ 

In our final call to close, since our initial assumption set contains no free type variables, step (2) of the algorithm discharges all assumptions from $B$. This final call, then, fails since the second constraint on $+$ is unsatisfiable. In the previous call to close, if we had discharged the constraints on $+$ by including them in $B'$, satisfiability would be decided by instantiation of $\gamma$ to $\text{int}$ and $\alpha$ to $\text{real}$. As a result, the final call to close would succeed, causing an untypable expression to be deemed typable.

### A. Parametric Overloading and Satisfiability

Typability in $\text{ML}_0$ is Turing reducible to the problem of deciding whether a set of constraints is satisfiable with respect to a given set of type assumptions. Through the use of constrained type schemes, we can be very expressive in representing overloads. It turns out that, unless we restrict our representations to certain kinds of
overloadings, the problem of constraint-set satisfiability, and therefore typability, in ML is undecidable [Smi91].

Consider the assumption set in Figure 4.2. We can see that the assumptions on `avg` and `+` contain infinite overloadings, e.g., `+` can assume a finite type, say `list(list(...(list(int))))`. Note also the occurrences of recursive overloadings, where the satisfiability of the constraint set depends on the assumption itself. A mutually recursive overloading would result if we added a constraint involving `avg` to the third assumption on `+`.

Constraint-set satisfiability remains undecidable in the presence of mutual recursion and/or straight recursion without restrictions [Vol94a]. We should therefore explore suitable bounds on recursion which make our satisfiability problem decidable. We can see that recursion is a natural occurrence in practice through our example in Figure 4.2. For this reason, while it makes constraint-set satisfiability decidable, forbidding recursion entirely is unacceptable.

Various approaches have been examined. Smith gives a restriction called overloading by constructors which makes constraint-set satisfiability decidable in polynomial time [Smi91]. But it disallows constraints on an overloaded identifier `x` involving `y` where `x ≠ y`. This would prohibit the overloading on `avg` in Figure 4.2. Another restriction, similar to that proposed by Kaes [Kae88] and adopted by Haskell [Has89],
is called \textit{parametric overloading}.

Parametric overloading is a more practical form of overloading which allows naturally recursive overloading like that of Figure 4.2 and makes constraint-set satisfiability decidable. This is the form that we adopt in this thesis.
Parametric overloading makes use of the concept of the least common generalization of finite types discussed earlier. We give a formal definition here from [Vol94b].

**Definition A.1** Parametric assumption sets are defined inductively.

The empty set is parametric.

If $A$ is parametric with no assumption for $x$ and $\sigma$ is a constrained type scheme $\forall \alpha$ with $C \cdot \tau$ such that for each $z : \rho \in C$, $z$ is overloaded in $A$ and $\rho$ is a generic instance of its $LCG$ then $A \cup \{x : \sigma\}$ is parametric.

If $A$ is parametric with no assumption for $x$ and $B$ is the set

\[
\begin{align*}
\{ x : \forall \bar{\gamma}_1 \text{ with } C_1 \cdot \tau[\alpha := \chi_1(\bar{\gamma}_1)] \\
\vdots \\
\{ x : \forall \bar{\gamma}_n \text{ with } C_n \cdot \tau[\alpha := \chi_n(\bar{\gamma}_n)] \}
\end{align*}
\]

such that

- $x$ has $LCG \forall \alpha \cdot \tau$,
- $\chi_i \neq \chi_j$ for $i \neq j$ (where $\chi$'s are type constructors of various arities), and
- $z : \rho \in C_i$ implies that $z$ has $LCG \forall \pi \cdot \rho$, for some $\pi \in \bar{\gamma}_i$, and either $z$ is overloaded in $A$ or $z = x$,

then $A \cup B$ is parametric.

Note that we can only specify constraints which involve an overloaded identifier; constraints involving finite types or even polymorphic types are not allowed under our definition. Though there are instances where this limits the practical use of parametric overloading, this restriction is generally not a limiting factor in practice. Smith has considered approaches to relaxing this particular restriction for type checking a language with subtyping and overloading [Smi91, Smi93]. This thesis, however, considers overloading only. We can also make the observation that an identifier $x$ parametrically overloaded in $A$ can always be characterized by an $LCG$ which has only one quantified variable. This gives us a practical view of the restrictions we are talking about.
We can characterize parametric overloads as a regular forest of trees [Vol94b]. These regular forests can be generated by a class of context-free grammars called regular tree grammars [GeS84]. If A is parametric then every overloaded identifier x in A has an LCG of the form \( \forall \alpha.\tau \) and the set of finite types \( \pi \) to which \( \alpha \) can be instantiated, meaning \( A \vdash x : \tau[\alpha := \pi] \) is derivable, form a regular tree language or forest.

**B. SATISFIABILITY ALGORITHM**

The determination of constraint-set satisfiability, which is computed by the function satisfiable(\( A, C \)), takes the assumption set \( A \) and the constraint set \( C \) as inputs. For any parametric assumption set \( A \), we can construct for every overloaded identifier \( x \) a regular tree grammar \( G_x \) such that if \( x \) has LCG \( \forall \alpha.\tau \) then for any variable-free finite type \( \tau' \), we can derive \( A \vdash x : \tau[\alpha := \tau'] \) if and only if \( \tau' \in L(G_x) \), where \( L(G_x) \) represents the regular tree language generated by \( G_x \). In this context, we need only parse \( \tau' \) with respect to \( L(G_x) \) to determine whether constraint \( x : \tau[\alpha := \tau'] \) is satisfiable with respect to \( A \).

An algorithm for satisfiability has been developed based on the property that regular forests are effectively closed under intersection [Vol94b]. Our implementation of \( W_0 \) uses this algorithm. Consider an example using the parametric assumption set of Figure 4.2 and the type constructors in Figure 4.3, which includes constructors of arity-0,1 and 2.
We can see that /, + and avg are overloaded in A with respective LCG's: \( \forall \alpha. \alpha \rightarrow \alpha \rightarrow \text{real}, \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha \) and \( \forall \alpha. \alpha \rightarrow \text{real} \). Our first task is to construct a dependency graph of assumptions in A; if an assumption on \( x \) contains a constraint on \( y \) we need to produce the grammar of \( y \) before we produce \( x \)'s grammar. We then can proceed to create regular tree grammars for each overloaded identifier in A based on dependencies. We see that avg depends on / and + in A so we must compute the grammar for avg last.

Since identifiers may be overloaded recursively, as in our example, we will represent occurrences of an identifier \( x \) in its own constraint list with the start symbol for \( G_x \). In the case of constrained type schemes with multiple constraints, as occurs in avg, we will represent this as a new non-terminal. This non-terminal will define new productions for the grammar which result from the computed intersection of the constraints. Given a constraint set which contains a constraint on \( x \) and a constraint on \( y \) there intersection is computed as \( L(G_x) \cap L(G_y) \).

We represent the type constructors in \( \Sigma \) as a grammar \( G_\Sigma \). We can then take advantage of the fact that \( L(G_\Sigma) \cap L(G_x) = L(G_x) \) for any overloaded identifier \( x \) as we construct our grammars for A. We therefore obtain the following set of grammars for our example assumption set A:

\[
\begin{align*}
G_\Sigma : & \quad S = \text{int} | \text{real} | \text{bool} | \text{list}(S) | \\
& \quad \text{ref}(S) | S \rightarrow S | \text{pair}(S,S) \\
G_f : & \quad A = \text{int} | \text{real} \\
G_+ : & \quad B = \text{int} | \text{real} | \text{list}(B) \\
G_{avg} : & \quad C = \text{list}(D) | \text{set}(D) \\
& \quad D = \text{int} | \text{real}
\end{align*}
\]

where the non-terminal D represents \( L(G_f) \cap L(G_+) \).

In our batch implementation of \( W_o \), where we do not allow occurrences of free type variables in the initial assumption set, we can create the set of regular tree
grammars once and reuse the representation. This was the approach we adopted in our implementation.

We can now determine satisfiability of a constraint set \( C \) with relation to an assumption set \( B \) by parsing each constraint in \( C \), of the form \( id : r \), with respect to the grammars computed for \( B \) i.e. if \( r \) parses with respect to \( L(G_{id}) \) for each constraint in \( C \) then \( C \) is satisfiable. It is possible, though, that we may encounter overlapping constraints in \( C \). In this case we must first compute the intersections on any overlapping constraints before parsing those that don’t overlap. If the computed intersection is empty then \( C \) is unsatisfiable. An intersection is empty if there exists no common type constructor of arity-0 between constraints. For example, grammar \( G \) below represents an empty intersection.

\[
G = list(G) \mid ref(G)
\]

This algorithm is exponential in the number of forests input, but this is very likely the best we can do for the problem has been shown NP-complete [Vol94b]. The use of our implementation of \( W \) should provide valuable insight into determining whether the NP lower bound for constraint set satisfiability is a practical limitation.
V. IMPLEMENTATION OF $W_o$

As we have shown, algorithm $W_o$ has been developed to infer the most general type of a term given suitable forms of overloading. We envision an interactive programming environment in which incomplete expressions are type checked (may have placeholder terms) and can be subsequently updated, perhaps requiring new types to be inferred.

In this setting, $W_o$ is unsuitable because it is not incremental. If a function, say $f$, is computed on input $x$, then on input change $\Delta$, we say that the computation of $f(x + \Delta)$ is incremental if $f(x + \Delta)$ is computed from only $f(x)$ and $\Delta$. Although our implementation does not type check definitions, it nonetheless exhibits incremental type re-computation at the expression level, as we will show.

In efforts to develop an incremental approach to type inference, we might attempt to re-write $W_o$. We have, however chosen an approach which makes use of a formalism, namely attribute grammars, for achieving incremental type re-computation. Utilizing this formalism we foresee our implementation not only providing a means to validate and explore bounds on the problem of type inference in the presence of overloading, but also as a step towards integrating incremental algorithms for on-line type inference and those for overloading.

A. THE ROLE OF ATTRIBUTE GRAMMARS

Updating expressions affords an opportunity to re-use previous type computation. The attribute grammar formalism provides a framework in which type re-computation is identified with attribute re-evaluation. So if attribute re-evaluation is
done incrementally then type re-computation is incremental as well and furthermore it is implicit in the formalism.

Using an attribute grammar we can specify the syntax of a language via a context-free grammar. Nodes of parse trees are annotated with attributes that are prescribed by a set of attribute equations given as part of the attribute grammar. If a parse tree is edited then attributes of the tree are re-computed using the equations so that a consistent attribution is maintained. Re-computing the attributes is implicit and is done by the attribute evaluator.

The productions of the context-free grammar for type inference in ML, which we have developed for our implementation are given in Figure 5.1. Non-terminals are represented in upper case while terminals are in lower case. Terminals in productions that begin with Null represent placeholder terms which have universal type ∀α.α.

Attributes are distinguished as either synthesized or inherited. Synthesized attributes occur on the left-hand side of attribute equations; inherited attributes occur on the right-hand side. In other words, in one case attributes are propagated up (synthesized) in the parse tree and in the other they are propagated down (inherited) in the parse tree. Figure 5.2 shows the inherited (AI) and synthesized (AS) attributes associated with the productions of Figure 5.1.

To implement W₀, we define attribute equations, which create dependencies between attribute values. As the derivation tree is updated these dependencies determine what part of the tree is affected and where selective re-computation, via the attribute equations, needs to be done in order to re-establish consistent attribute values throughout the tree. The set of attribute equations in Figure 5.3 then defines the dependencies required in each attributed production from Figure 5.1 to implement W₀. Functional support, indicated by italics, is simplified and represented by descriptive function names. The attributes S and B of EXP are precisely those terms returned by W₀ as discussed in Chapter IV.
Figure 5.1: Context-free grammar for $ML_0$ type inference
To illustrate how incremental type recomputation is achieved via incremental attribute evaluation, consider Figure 5.4. Here we have a partial derivation tree annotated with a dependence graph showing the propagation of attributes in the tree. For simplicity, we have chosen one inherited attribute and one synthesized attribute. The inherited attribute $A$ represents an assumption set. The synthesized attribute $T$ is a constrained type scheme representing the type of an expression at each node of the tree. Figure 5.4 represents the partial derivation tree for the expression $pr(x, \lambda y.\lambda z. y z)$, where $pr$ is of type $\forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow \text{pair}(\alpha, \beta)$.

Suppose the expression rooted at node $n_3$ is updated. We can see that $T$ at node $n_2$ now must be recomputed but notice that no change has been made to the expression rooted at node $n_4$, which therefore need not be retypechecked. In practice, this can result in significant savings as the tree whose root is $n_4$ can be arbitrarily large.

1. The Synthesizer Generator Platform

An attribute evaluator generator takes as input a set of attribute equations, such as those in Figure 5.3, for a set of terms and outputs an attribute evaluator that takes a term and annotates it with an attribution as prescribed by the equations. There are attribute evaluator generators available today that not only output an attribute evaluator but output one that evaluates attributes incrementally. One such generator is GrammaTech's Synthesizer Generator (SynGen).
Figure 5.3: Attribute equations for ML* type inference
Figure 5.4: A partial derivation tree and dependence graph for \( pr(x, \lambda y.\lambda z. y z) \)
We have developed our implementation utilizing SynGen for several reasons. We are able to get a comprehensive and visually appealing X-windows interface with relative ease. Utilizing Syngen also fits nicely with our opinion of attribute grammars as being a desirable approach to achieving incremental type inference. Furthermore, since we profit by the incremental algorithms embedded in SynGen, new advances in this area, which may well be incorporated in future versions, will directly enhance our implementation.

The incremental algorithms used in SynGen rely heavily on the concept of ordered attribute grammars which were introduced in [Kas80]. The ordered attribute grammars are a subclass of the noncircular attribute grammars. Though SynGen can accept attribute grammars which are not ordered, it prohibits circular attribute grammars.

The language of SynGen is SSL. Every (useful) SSL specification has three major declaration areas: Abstract syntax which defines a set of grammar rules, Attribution which annotates the grammar with attributes and describes their dependencies, and Unparsing which defines display formats for terms, identifies selectable productions of the grammar and annotates which productions are editable. For our implementation, Figure 5.1 represents the Abstract syntax and Figures 5.2 and 5.3 represent the Attribution.

2. The Implementation

We demonstrate our implementation through an annotated sequence of actual X-windows display screens generated by our type checker. Figure 5.5 shows an initial screen with placeholders for an assumption set entry, where we define extensions to an Initial Environment, and an expression. The currently selected term, corresponding to the ASSUMPTIONSET production of our grammar, is underlined. Note that the type inferred for the placeholder term <exp> is <universal type>.
Figure 5.5: Implementation initial screen.
ASSUMPTIONS:

expon: ∃ α with (mult((α → (α → α))). (α → (int → α)));
exq?: ∃ α with (eq?: ((α → (α → bool))). (α → (α → bool)));
   : ∃ α with () . (list(α) → (list(α) → bool));
   : (int → (int → bool));
mult: ∃ α with (mult((α → (α → α))). (α → (α → α)));
   : (int → (int → int));
   : (real → (real → real));
   : ∃ α with (mult((α → (α → α))). eq?: ((α → (α → bool))). (list(α) → (list(α) → list(α))));
   : <type>

EXPRESSIONS:

<exp>

TYPE: <universal type>

Figure 5.6: An assumption set defined.

We have entered an assumption set in Figure 5.6 with three overloaded identifiers. The first type scheme for each identifier, without the constraints, must represent the LCG of that identifier. The implementation currently does not compute the LCG and so it must be provided by the user. Note the terms enclosed in boxes at the bottom of the screen. These are called transforms. With the placeholder for TYPEEXP selected, we may select a transform with the mouse and replace the selected placeholder term with a term associated with the transform. This provides an alternative means to enter terms without the need for the user to remember the appropriate syntax. Users may also enter terms directly as long as the term being edited is defined in the unparsing rules.
In Figure 5.7 we have entered three expressions whose types have been inferred. The type of our first expression is represented by a constrained type scheme; it is the most general type we can give to it and we can be no more specific without more information. In the second expression, where \( r : \text{real} \) is defined in the initial environment, we see that, since \( \text{mult} \) is defined over reals, applying \( \text{expon} \) to \( r \) satisfies the constraint on \( \text{expon} \) and we are able to infer a finite type for the expression. An unsatisfiable constraint has been encountered in our final expression. This is a result of the multiple constraints on \( \text{mult} \) and \( \text{eq} \) in the third assumption of \( \text{mult} \). We can see that the grammar for \( \text{mult} \) is:

\[
\begin{align*}
G_{\text{mult}} : & \quad S = \text{int} | \text{real} | \text{list}(U) \\
U = & \quad \text{int} | \text{list}(U)
\end{align*}
\]

which clearly does not derive \( \text{list(\text{real})} \).

It is also possible to directly examine the attributes of the parse tree at any point in the execution. This functionality, though mainly useful for debugging, can provide a means to investigate aspects of the implementation from a lower level viewpoint. For example, one might wish to examine a representation of the regular tree grammars produced for overloaded identifiers in a given assumption set. This can be done by examining the attribute \( \text{typeGrammar} \) at any \( \text{EXP} \) node of the parse tree. For instance, Figure 5.8 shows the regular tree grammars computed for the assumption set of Figure 5.7. Note that we have chosen to represent the start symbol of grammar \( G_{\text{id}} \) as \( \text{id} \), for each overloaded identifier \( \text{id} \) in the assumption set. In addition, \( \text{id}_1 \ldots \text{id}_2 \) was chosen to represent \( L(G_{\text{id}_1}) \cap L(G_{\text{id}_2}) \).

We have given a brief overview, through examples, of the X-windows interface and general functionality of our implementation of \( W \) with parametric overloading. By examining instances of type inference in \( ML_\omega \), in the setting of our implementation, we have endeavored to provide the reader with a clearer understanding of concepts discussed more formally in previous chapters.
**Figure 5.7:** Type inference of three expressions.
Figure 5.8: Representation of attribute *typeGrammar*
VI. CONCLUSIONS

We have considered the problem of type inference in an extension to the type system \( ML \) called \( ML_0 \). The type system \( ML_0 \) is a formalism which is more suitable for implementing future languages by virtue of its incorporation of global overloading. Yet this increased functionality introduces new problems in developing algorithms which make typability decidable. Without restrictions on the types of overloading and the structure of constraint sets, typability in \( ML_0 \) is undecidable. Typability in \( ML_0 \) is Turing reducible to the problem of determining if a set of constraints is satisfiable with respect to a given set of assumptions. If assumption sets are restricted to parametric overloading the problem of constraint set satisfiability is NP-complete.

The type inference algorithm \( W_0 \) with parametric overloading has been implemented utilizing the formalism of attribute grammars with GrammaTech’s Synthesizer Generator. It performs type inference on expressions in an interactive environment. Type inference is performed incrementally so that the types of partial expressions can be inferred and efficiency of re-computation in the presence of updates is enhanced. Consequently, immediate feedback is provided to the user as expressions are entered and updated.

Our implementation will be used to examine the practical bounds on the problem of constraint-set satisfiability. It will also represent a significant tool for exploring the limits of bounded polymorphism, or overloading, in programming languages. Can we devise new forms of overloading which are more flexible than parametric overloading yet retain a decidable satisfiability problem?
A. FUTURE WORK

This thesis will serve as a basis for further research aimed at ultimately developing a type discipline for a class of implicitly-typed imperative programming languages with subtypes, overloading and polymorphism. A more immediate goal is to merge our implementation of $W_\circ$ with an SSL implementation that performs on-line type inference utilizing the type inference algorithm $W$ of $ML$.

On-line type inference allows the introduction of new global definitions as a program is produced. This differs from our batch implementation, where we have assumptions about types of free ids available to each expression in the form of an assumption set. The incorporation of overloading in an on-line implementation will be the subject of the next step in this research effort. This will produce an interactive environment where global definitions, perhaps overloaded, may be introduced at any point in the program. Types of all dependent terms are then recomputed as a result of these new definitions.
LIST OF REFERENCES


INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center  
   Cameron Station  
   Alexandria, VA 22304–6145

2. Dudley Knox Library  
   Code 52  
   Naval Postgraduate School  
   Monterey, CA 93943–5101

3. Chairman, Computer Science Department  
   Code CS  
   Naval Postgraduate School  
   Monterey, CA 93943

4. Dr. Dennis M. Volpano  
   Code CS/Vo  
   Naval Postgraduate School  
   Monterey, CA 93943

5. Dr. Craig W. Rasmussen  
   Code MA/Ra  
   Naval Postgraduate School  
   Monterey, CA 93943

6. Bruce J. Bull  
   1106 Spruance Road  
   Monterey, CA 93940