Lambda Calculus Models
of
Typed Programming Languages

by

John Clifford Mitchell

B.S., Stanford University (1978)

S.M., Massachusetts Institute of Technology (1982)

submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements of the Degree of

Doctor of Philosophy in Computer Science

at the

Massachusetts Institute of Technology

August 1984

© John C. Mitchell 1984

The author hereby grants to MIT Permission to reproduce and to distribute copies of this thesis
document in whole or in part.

Signature of Author ____________________________

Department of Electrical Engineering and Computer Science
August 24, 1984

Certified by ____________________________

Albert R. Meyer, Thesis Supervisor

Accepted by ____________________________

Arthur C. Smith, Chairman, Departmental Committee on Graduate Students
Lambda Calculus Models of
Typed Programming Languages

John Clifford Mitchell

Submitted to the Department of Electrical Engineering and Computer Science
on August 24, 1984 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

The first part of this thesis studies the second-order lambda calculus, developed independently
by Girard and Reynolds. In this typed language, featuring polymorphic functions and abstract
data type declarations, types play an important role in defining the set of well-formed terms.
We discuss the features of second-order lambda calculus that correspond to common
programming language constructs, demonstrating some natural extensions to Ada, Aplphard,
CLU, ML and related programming languages. In particular, we describe a simple approach to
passing representations of abstract data types as parameters and returning representations as
results of function calls.

The semantics of second-order lambda calculus is studied using a slightly more general
higher-order lambda calculus that makes it possible to treat type-building operations like the
product-space constructor or the tagged union constructor as optional constants of the language.
We define semantic models for the general language $\mathcal{L}$ and prove a completeness theorem,
extending previous work on the second-order lambda calculus. A formal axiomatization of
models of $\mathcal{L}$ is then given using a higher-order type theory $\mathcal{L}_T$. This axiomatization is similar
in spirit to the first-order combinatory characterization of models of untyped lambda calculus.

The second part of the thesis is concerned with type inference, the problem of finding types for
untyped expressions. We study two type inference systems for untyped lambda calculus. The
first system combines a relatively simple language of types with some simple postulates about
relationships between types. The main results here are a complete axiomatization for all valid
typing statements and a decision procedure for a natural class of typing statements.

The second type inference system includes the more complicated universally quantified type
expressions of second-order lambda calculus. A general definition of the semantics of typing
statements with universally quantified types is proposed, generalizing previous work by
MacQueen, Sethi and Plotkin. These inference models are models of untyped lambda calculus
with extra structure similar to models of second-order lambda calculus. We show that the $GR_{eq}$
axiom system, an extension of the typing rules for second-order lambda calculus, is complete for
all typing statements valid over all inference models. A more specialized set of type inference
rules, the $GRS_{eq}$ rules, characterize the more specialized simple semantics. We also study
containments between types by reformulating the inference rules so that containments play a
central role.

Thesis Supervisor: Albert R. Meyer, Professor of Computer Science

Keywords: types, second-order lambda calculus, polymorphism, abstract data types, lambda
calculus, programming languages, semantics, type inference, coercions.
Acknowledgments

Albert Meyer has been a constant source of enthusiasm, encouragement, and measured criticism during my tenure as a graduate student. Albert read the thesis carefully, contributing many valuable technical and stylistic suggestions. The other members of my thesis committee, John Gutttag and Sy Freidman, also deserve thanks for their efforts. John suggested numerous improvements to an early draft of Chapter 2.

Gordon Plctkin contributed directly to the technical content of the thesis. Gordon's enthusiastic view of the formulas-as-type analogy led to the work described in Chapter 2, which will appear as a joint paper. Chapter 3 draws on previous work by Kim Bruce and Albert Meyer. A number of valuable conversations with Kim and Albert made the work on semantic models possible.

In the early stages of the thesis work, I spent a pleasant and productive summer at AT&T Bell Labs. Ravi Sethi suggested the topic of Chapter 4 and Dave MacQueen provided many helpful insights into type inference. I thank Ravi, Dave and Al Aho for their continuing interest in my research.

I am indebted to Paris Kanellakis, Ron Pinter and Ch:ristos Papadimitriou for their assistance and encouragement during my student years. My family and friends also lent their support at various critical times.

For food and shelter during the last four years, I am grateful to the fellowship committees of NSF and IBM.
Table of Contents

Chapter One: Overview

1.1 Introduction ........................................... 7
1.2 Abstract Types Have Existential Types ................. 9
1.3 Semantic Models of Second-Order Lambda Calculus ... 11
1.4 Type Inference with Simple Coercions ................ 12
1.5 Type Inference with Polymorphic Types ............... 13
1.6 Open Problems and Future Directions ................. 14

Chapter Two: Abstract Types have Existential Types ...

2.1 Introduction ........................................... 15
2.2 The Typed Language SOL ............................. 22
2.2.1 Functions and Let ................................ 24
2.2.2 Products and Sums ................................ 25
2.2.3 Polymorphism ..................................... 26
2.2.4 Data Abstraction and Existential Types .......... 28
2.2.5 Intuitive Semantics of Existential Types ......... 31
2.2.6 Properties of SOL ................................ 33
2.2.7 Alternative Views of Abstype ..................... 35
2.3 Formulas as Types .................................... 37
2.3.1 Extensions to SOL ................................ 39
2.4 More Programming Examples ........................... 40
2.4.1 Universal and Existential Parameterization .... 41
2.4.2 Data Structures using Existential Types ......... 44
2.5 Conclusion and Directions for Further Investigation 46
1 Collected Syntax of SOL ................................. 48

Chapter Three: Semantic Models for Second-Order Lambda Calculus

3.1 Introduction ........................................... 50
3.2 Higher-Order Lambda Calculus ......................... 52
3.2.1 Second-Order Lambda Calculus and Existential Types 55
3.3 Axioms and Inference Rules ............................ 56
3.3.1 Infinite Products, Sums and Existential Types ... 58
3.4 Environment Models .................................... 60
3.4.1 Models for Constructionals ....................... 60
3.4.2 Higher-Order Lambda Frames ...................... 61
3.4.3 Meanings of Terms ................................ 62
3.5 Completeness .......................................... 64
3.6 Higher-Order Type Theory and Combinatory Algebras 68
3.6.1 Higher-Order Type Theory 70
3.6.2 Applicative Frames 70
3.6.3 Combinatory Completeness 71
3.7 Combinatory Models 74
  3.7.1 Combinatory Model Definition 74
  3.7.2 Combinatory Model Theorem 77
3.8 Definability of Existential Types 80
3.9 Directions for Further Investigation 82

Chapter Four: Type Inference with Simple Coercions 84

4.1 Introduction 84
4.2 Lambda Calculus and its Semantics 87
4.3 Type Expressions, Coercion Sets and Type Assignments 88
4.4 Rules for Type Inference 90
4.5 A Type Checking Algorithm 97
  4.5.1 Substitution and Coercion Set Algorithms 100
  4.5.2 Algorithm TYPE 102
4.6 Variations: Conversion Functions and ML Polymorphism 105
  4.6.1 ML Polymorphism 107
4.7 Conclusion and Future Directions 107

Chapter Five: Type Inference with Polymorphic Types 110

5.1 Introduction 110
5.2 Syntactic Type Inference for Second-Order Lambda Calculus 111
  5.2.1 Type Inference Rules 112
5.3 Semantics of Type Inference 114
  5.3.1 Untyped Lambda Calculus 116
  5.3.2 Semantic Models of Second-Order Lambda Calculus 117
  5.3.3 Models of Type Inference 119
5.4 Completeness and Type Containment 120
  5.4.1 Extensions to GR 120
  5.4.2 Soundness and Completeness 121
  5.4.3 Typing Rules Based on Containment 123
5.5 Simple Semantics and Containment 126
  5.5.1 Containments in Simple Inference Models 127
5.6 Discussion 128
  5.6.1 Retyping Functions 129
  5.6.2 Principal Type Schemes? 130
  5.6.3 Examples of Inference Models 131
  5.6.4 Comments on the Function-Semantics 133
  5.6.5 Open Problems 134

Chapter Six: Directions for Further Investigation 136
Chapter One

Overview

1.1 Introduction

Lambda calculus has proven a useful tool in the study of programming languages. Essentially, the value of lambda calculus is that many programming language constructs may be explained and clarified by translating the constructs into lambda calculus expressions. For example, the mathematical semantics of a variety of programming languages may be specified using translations into well-understood forms of lambda calculus [Gordon 79, Stoy 77]. The binding mechanism of lambda calculus also provides a simple paradigm for understanding variable binding in common programming languages [Landin 65, Landin 66, Reynolds 81a]. This thesis is an investigation of some formal systems that combine lambda calculus with types. The goal of this research is to advance our understanding of types in programming languages.

Two essentially different views of typing are studied. Although each point of view has a moderate history, neither is very clearly defined. The proceedings of the recent International Symposium on the Semantics of Data Types contains papers analyzing both views of typing [Semantics of Data Types 84], and some comparisons are given in [Leivant 83a]. Roughly speaking, the first view of types is that every mathematical object belongs to a unique type. In this view the integer 3 is different from the real number 3 and the composition functional for integer functions differs from composition for real functions. It is meaningless to write 3 without specifying whether we mean the integer or real number. Similarly, the typeless double-composition function

\[ \text{double compose} = \lambda f. f \circ f \]

does not have a precise meaning unless we specify the domain of this function.\(^1\) We can only make sense of typed function expressions like the integer double-composition function

---

\(^1\)Intuitively, the lambda expression \( \lambda f. f \circ f \) means "the function defined by treating the expression \( f \circ f \) as a function of the free variable \( f \)." We may regard \( f \circ f \) as an abbreviation for \( \lambda x. f(f(x)) \).
\textit{integer compose} = \lambda f \in \text{int} \to \text{int}. f \circ f.

The domain of this function, \( \text{int} \to \text{int} \), is the type of functions from integers to integers. The programming languages Ada [Ada 80], Algol 68 [TANENBAUM 76], Alphard [SHAW 81, WULF et. al. 76] and CLU [LISKOV et. al. 77, Liskov et. al. 81] are consistent with this view of typing.

The second view of typing is that all the mathematical objects we are concerned with belong to some large universe. We can identify some elements of the universe as being integers, reals, or functionals over integer functions. In addition, we consider some essentially "typeless" expressions to be semantically meaningful. For example, we have the identity function

\[ \text{identity} = \lambda x. x \]

defined on all elements of the universe. In particular, the typeless identity function is an element of its own domain. In this view, types are essentially properties of values from some typeless universe, and many elements will have many types. For example, since the identity function maps integers to integers, the identity function may be regarded as an element of the type \( \text{int} \to \text{int} \) of integer functions. The type checking algorithm for the programming language ML is based on this implicit view of types [Gordon, et. al. 79, Milner 78].

It should be mentioned that there are ways of considering each view as a special case of the other. We will give only a brief, informal hint, beginning with a typed perspective of "typeless" elements like the identity function on the typeless universe. Let us consider the universe \( U \) a type, and assume that we have ways of embedding all other types into \( U \). Then, the identity function from \( U \) to \( U \) corresponds, via the embedding of \( U \to U \) into \( U \), to an element of \( U \). Similarly, elements of all other types may be identified with their images under the appropriate embeddings into \( U \). This technique may be used to interpret untyped lambda expressions in a classical, set-theoretic way [Meyer 82]. Conversely, by embedding all types into some universal domain, we can regard typed elements as elements of some essentially "typeless" universe [Scott 76, Scott 80]. Thus, although the two views of typing differ in spirit, there are important, technical ways in which each view reflects on the other.

The first part of the thesis is concerned with the \textit{second-order lambda calculus}, developed independently by Girard [Girard 71] and Reynolds [Reynolds 74]. In this typed language,
featuring polymorphic functions and abstract data type declarations, every expression has a unique type. Chapter 2, immediately following this overview, describes the features of second-order lambda calculus that correspond to common programming language constructs. A more technical investigation of semantic models for second-order lambda calculus appears in Chapter 3. The second part of the thesis explores some of the consequences of viewing types as descriptive information about untyped programs. Chapters 4 and 5 are concerned with type inference, the problem of finding the types of untyped programming language expressions. In both chapters, the untyped lambda calculus serves as the example programming language. The fourth chapter considers a relatively simple language of type expressions, together with some simple postulates about relationships between types. The fifth chapter studies type inference with type expressions from the second-order lambda calculus.

All of the chapters are written so that each may be read independently of the others. The relatively informal discussion of second-order lambda calculus in Chapter 2 provides some of the computer science motivation for the study of Chapter 3. However, the technical content of Chapter 3 is independent of this motivation. Except for the informal connection between the second-order lambda calculus and the type inference rules of Chapter 5, neither of the first two chapters has much bearing on the later chapters. Since Chapter 4 treats a simpler type inference problem than Chapter 5, it is more accessible to the type inference novice. Each chapter is summarized below.

1.2 Abstract Types Have Existential Types

Based on joint work with Gordon Plotkin, this chapter describes the salient features of second-order lambda calculus using programming language terminology. Although the terse syntax of second-order lambda calculus makes it cumbersome to write large programs, the language seems to capture the essence of the type structures of Ada, Alphard, CLU and related typed programming languages. In addition, the second-order lambda calculus suggests some natural extensions to these languages.

Previous researchers have observed that second-order lambda calculus has a very general
mechanism for defining polymorphic functions [Bruce and Meyer 84, Donahue 79, Fortune, et. al. 83, Haynes 84, McCracken 79, Reynolds 74, Reynolds 83]. For example, one can easily write a function to select the maximum of two elements of any type \( t \), given an order relation on the type \( t \). (This polymorphic maximum function is discussed in Section 2.2.3.) We point out that Girard's original formulation of the language [Girard 71] also suggests ways to allow representations of abstract data types to be passed as parameters or returned as results of function calls.

There are practical applications for the flexible treatment of abstract data types found in second-order lambda calculus. Consider the problem of searching a tree for a node with a certain label. Starting from the root of the tree, there are many ways to search through the tree. Two common strategies are depth-first search and breadth-first search [Aho, Hopcroft and Ullman 83]. Both move a pointer through the tree, keeping track of unsearched subtrees using a data structure. A depth-first search algorithm uses a stack with operations \( \text{push} \) and \( \text{pop} \), repeating the following until the goal node is found:

\[
\text{if \ node is not a leaf \ then \ push(\text{right son}, \ stack), \ push(\text{left son}, \ stack);} \\
\text{node} \leftarrow \text{pop(stack)}
\]

Of course, the second \( \text{push} \) may be considered redundant, since the next statement immediately \( \text{pop} \)'s the left son off the stack. However, this form of the algorithm provides an easy comparison with breadth-first search. A breadth-first search algorithm uses a queue with operations \( \text{add} \) and \( \text{remove} \), repeating the step

\[
\text{if \ node is not a leaf \ then \ add(\text{right son}, \ queue), \ add(\text{left son}, \ queue);} \\
\text{node} \leftarrow \text{remove(queue)}
\]

Since \( \text{remove} \) always returns a node from the highest unsearched level of the tree, nodes are searched breadth-first. Note that breadth-first is identical to depth-first search, except that we use \( \text{queue, add} \) and \( \text{remove} \) instead of \( \text{stack, push} \) and \( \text{pop} \). Therefore, if we are allowed to make an entire abstract data type consisting of a type and its operations a parameter, we could use the same procedure for both search strategies. We would simply pass a representation of \( \text{queue} \) with \( \text{add} \) and \( \text{remove} \) for breadth-first search, or \( \text{stack, push} \) and \( \text{pop} \) for depth-first search. Furthermore, we could adopt a more sophisticated search strategy by passing a priority queue as a parameter. If the strategy used in the \( \text{remove} \) operation of the priority queue depends on the
structure of the tree, then it might be desirable to compute this strategy at run time. The ability to compute abstract data types at run time and pass them as parameters goes beyond most contemporary typed programming languages. The subtle issues of type checking parameters consisting of a type with operations are discussed in the Chapter.

Essentially, the flexible treatment of abstract data types in second-order lambda calculus arises from the fact that, in this language, implementations of abstract data types are values which have types themselves. These types are existential types, a kind of type originally developed in constructive logic. We discuss connections with constructive logic and suggest further research into a programming language whose type expressions are specifications.

1.3 Semantic Models of Second-Order Lambda Calculus

The syntax and operational semantics of second-order lambda calculus have been known for over a decade. However, the model theory (or denotational semantics) of the language is only beginning to be understood (cf. [Bruce and Meyer 84]). It is important to develop semantic models for this language as a basis for proving properties of languages like Ada and CLU which can be viewed as variants of the second-order lambda calculus. Other motives for developing semantics models of programming languages are discussed in textbooks on the subject [Gordon 79, Stoy 77].

Several model definitions for second-order lambda calculus have been proposed, each tailored to a specific set of type constructors and relying on the syntax of terms to characterize an important property of models (roughly, closure under definition of functions by polynomials). The goals of this chapter are (1) to provide a general framework for studying second-order lambda calculus with any set of type constructors, e.g., with or without product types, and (2) to separate the model theory of the language from syntactic considerations.

Since it is natural to think of the second-order universal and existential types $\forall t.\sigma(t)$ and $\exists t.\sigma(t)$ as the results of applying different operators to the "type function" $\lambda t.\sigma(t)$, the semantics of second-order lambda calculus naturally involves functions over the set of types. We study the semantics of second-order lambda calculus using a slightly more general higher-order lambda
calculus $\mathcal{L}$ with, e.g., functionals over type functions. This more general language makes it possible to consider type-building operations like the product-space constructor or the tagged union constructor as optional constants of the language. We define semantic models for the general language $\mathcal{L}$ and prove a completeness theorem, extending the analogous development of [Bruce and Meyer 84] for the basic second-order lambda calculus without products, tagged unions or existential types.

In the later sections of the Chapter, models of $\mathcal{L}$ are characterized without reference to the meanings of terms. This characterization, similar in spirit to the algebraic characterization of models of untyped lambda calculus using combinators [Barendregt 81, Meyer 82], is formalized by introducing a higher-order type theory $\mathcal{L}_H$. Since the binding operators of $\mathcal{L}$ are omitted from $\mathcal{L}_H$, the type theory $\mathcal{L}_H$ is essentially reducible to first-order logic. In essence, we present an axiomatization of $\mathcal{L}$ models in a relatively straightforward logical language.

1.4 Type Inference with Simple Coercions

This chapter studies type inference with type schemes and a simple form of coercion between types. Type schemes, also studied by Hindley [Hindley 69, Hindley 83a, Hindley 83b], Milner [Damas and Milner 82, Milner 78] and others [Barendregt, Coppo and Dezani 83], are type expressions like $t \rightarrow t$ using type variables. Intuitively, the type scheme $t \rightarrow t$ describes a set of elements from some untyped domain which map the type $t$ into itself. The simple coercions express containment properties of the form "every integer is a real number."

We study type inference using typing statements $C, A \vdash M : \sigma$, where $C$ is a set of coercions, $A$ is an assignment of types to variables, $M$ is an untyped lambda expression and $\sigma$ is a type expression. Intuitively, a typing statement means that whenever $C$ and $A$ hold, the expression $M$ has type $\sigma$. We give typing statements precise semantics using models of untyped lambda calculus, following the approaches of [Barendregt, Coppo and Dezani 83, Hindley 83a]. The main results of the Chapter are a complete axiomatization for all valid typing statements and a decision procedure for a natural class of typing statements. (The set of valid typing statements is undecidable.) The completeness theorem is a relatively straightforward generalization of
previous work without coercions [Hindley 83a]. In contrast, the decidability proof with coercions seems to require techniques beyond the proof without coercions [Hindley 69]. In practical terms, the decision procedure is a typing algorithm that may be used to add simple coercions to programming languages like ML [Gordon, et. al. 79, Milner 78].

1.5 Type Inference with Polymorphic Types

This chapter is concerned with type inference using the universally quantified type expressions of second-order lambda calculus. These type expressions have bound type variables, in addition to the free type variables found in type schemes. A general definition of the semantics of typing statements with universally quantified types is proposed, generalizing previous work by MacQueen, Sethi and Plotkin [MacQueen and Sethi 82, Mac:Queen, Plotkin and Sethi 84]. These inference models are models of untyped lambda calculus with extra structure similar to models of second-order lambda calculus. We show that the GR_eq axiom system, an extension of the typing rules for second-order lambda calculus, is complete for all typing statements valid over all inference models. A more specialized set of type inference rules, the GRS_eq rules, characterize the "simple semantics" with the functional type σ→τ interpreted as all elements of the model that map σ to τ and the universal type ∀t.σ(t) interpreted as the intersection of all σ(τ).

The set of types associated with the identity function has an appealing semantic interpretation: the identity λx.x has type σ→τ iff the containment between types σ⊆τ is valid. We study containments by reformulating the inference rules so that containments play a central role. Two containment-based inference system are presented, one for arbitrary inference models and one for the more specialized simple semantics models. The two typing systems differ only in their rules for deducing containments. Both inference systems are complete for deducing typing statements and completeness theorems for the sets of valid containments follow as corollaries. The difference between the two sets of containment rules clearly distinguishes the simple semantics from arbitrary inference models.
1.6 Open Problems and Future Directions

The concluding section of each chapter describes open problems related to the content of the chapter. The most significant open problems are to develop elementary models of second-order lambda calculus (see Chapter 3) and to determine whether the semantically incomplete typing theory based on the typing rules for second-order lambda calculus is decidable (see Chapter 5).

Both problems have a significant history: after attempting to construct an elementary "set-theoretic" model of second-order lambda calculus [Reynolds 83], Reynolds has recently shown that no such construction is possible [Reynolds 84]. Nonetheless, it may be possible to construct models that seem rather "elementary," yet are not "set-theoretic" in the precise sense used by Reynolds. The second-order type inference problem has also attracted some attention [Leivant 83a, McCracken 84], with one incorrect, published proof of decidability.

From a general point of view, there are important questions about types in programming languages that neither the second-order lambda calculus nor the type inference systems are designed to address. For example, a simple-minded view of type checking is that "type checking prevents type errors." None of the formal systems used in the thesis seem to explain adequately, in a precise or semantic sense, what a type error really is. This general question, and others of a similar nature, are discussed in the brief concluding remarks of Chapter 6.
Chapter Two

Abstract Types have Existential Types

2.1 Introduction

Many typed programming languages, such as Ada, Alphard, CLU and ML, have some form of abstract data type declaration. In each of these languages, an abstract data type declaration binds a group of identifiers to some kind of value. This paper examines the values that are bound by abstract data type declarations and suggests that these values have types. In the course of the analysis, we will discover that contemporary typed languages can be extended to allow concrete representations of abstract data types as parameters and results of function calls.

Ada packages [Ada 80, Sherman et. al. 82], Alphard forms [Shaw 81, Wulf et. al. 76], CLU clusters [Liskov et. al. 77, Liskov et. al. 81], and abstype declarations in ML [Gordon, et. al. 79] all bind identifiers to values. In each of these declarations, a group of names is bound to a composite value consisting of a set of values and zero or more operations on the set of values. For example, in the ML declaration

\[
\text{abstype complex} = \text{real} \# \text{real}
\]

\[
\text{with create(x, y) = ...}
\]

\[
\text{and plus(z, w) = ...}
\]

\[
\text{and re(z) = ...}
\]

\[
\text{and im(z) = ... ;;}
\]

the names complex, create, plus, re and im are bound to an implementation of complex numbers. The implementation consists of a set of values, denoted by the ML expression real \# real for the type of pairs of reals, and the functions denoted by the code for create, plus, and so on. We will call a composite value constructed from a set and zero or more operations a

\[\text{\footnote{A single Ada package or ML declaration may bind more than one type identifier. In addition, an Ada package may declare types other than private types. At present, we are only concerned with declarations that introduce a single private type and zero or more operations.}}\]
data algebra. The set will be called the carrier of the data algebra. The relation between data algebras and abstract data types is that a data algebra is an implementation or concrete representation of an abstract data type.

The declaration bodies of Ada packages, Alphard forms, CLU clusters, and ML abstract type (abstype) bindings all define data algebras. An example body is the body of the following CLU complex number cluster. The cluster defines a set of values using the type expression record[r, i: real] and defines four operations by giving implementations.\(^3\)

\[
\text{complex} = \text{cluster is create, plus, re, im}
\]

\[
\text{rep = record[r, i: real];}
\]

\[
\text{create = proc (x, y: real) returns (cvt);}
\]

\[
\text{return (rep$\{r: x, i: y\})}
\]

\[
\text{end create}
\]

\[
\text{plus = proc (z, w: cvt) returns (cvt);}
\]

\[
\text{return (rep$\{r: (z.r + w.r), i: (z.i + w.i)\})}
\]

\[
\text{end plus}
\]

\[
\text{re = proc (z: cvt) returns (real);}
\]

\[
\text{return z.r}
\]

\[
\text{end re}
\]

\[
\text{im = proc (z: cvt) returns (real);}
\]

\[
\text{return z.i}
\]

\[
\text{end im}
\]

\[
\text{end complex}
\]

In essence, the complex cluster binds identifiers to the data algebra defined by the body

---

\(^3\) Some notational conventions that are peculiar to CLU are the use of rep in the body of create and cvt in proc headers. In the expression rep$\{r: x, i: y\}, the reserved word rep is used to clarify the type of the record \(\{r : x, i:y\}\). This record has the same type as the "representation type" of the cluster being defined. The reserved word cvt in headers and return statements means "abstract outside, concrete inside." In determining the type of a routine, cvt is equivalent to the abstract type (in this case complex), but when type-checking the body, cvt is equivalent to the representation type (in this case record\{r, i: real\}).
rep = record[r, i: real];
proc (x, y: real) returns (cvt);
    return (rep$r: x, i: y$)
end
proc (z, w: cvt) returns (cvt);
    return (rep$r: (z.r + w.r), i: (z.i + w.i)$)
end plus
proc (c: cvt) returns (real);
    return c.r
end
proc (c: cvt) returns (real);
    return c.i
end

We will call package bodies, form bodies, cluster bodies and abtype binding bodies data
algebra expressions, even though none of these syntactic constructs is commonly recognized as
an expression that denotes a value. In our terminology, a cluster body is a data algebra
expression that denotes a data algebra.

In the remainder of this section, we discuss the type checking rules associated with abstract data
type declarations. In the process, we define the types of data algebras. These types are
existential types, a kind of type originally developed in constructive logic. Existential types are
closely related to infinite sums in category theory. In the next section, we present a statically-
typed language SOL based on the polymorphic lambda calculus of Reynolds [Reynolds 74] and
Girard [Girard 71]. The design of SOL suggests ways to extend languages like Ada, Alphard,
CLU and ML so that data algebras may be passed as parameters to functions and returned as
results. SOL also seems to be a natural "kernel language" for studying the semantics of
languages with polymorphic functions and abstract data type declarations. An operational
semantics of SOL is presented using reduction rules, but we do not consider practical
implementation issues. We believe that SOL provides greater flexibility in the use of abstract
data types than previous languages, without weakening any of the type checking rules
commonly associated with abstract data type declarations. However, since we do not have a
precise characterization of the "security" of type checking in other languages, we are unable to
show rigorously that SOL is equally secure.

In each of the languages we have mentioned, there are specific type-checking rules associated with abstract data type declarations. We discuss type restrictions by introducing the abstract data type declarations we will use in SOL. Basic data algebra expressions in SOL will have the form

\[ \text{rep } \tau \ M_1 \ldots \ M_n \]

where \( \tau \) is a type expression and \( M_1, \ldots, M_n \) are function expressions. As in CLU cluster bodies, the type expression \( \tau \) denotes a set of values, and expressions \( M_1, \ldots, M_n \) denote operations. However, the function expressions will not involve special reserved words like \textit{cvt} and \textit{rep}. The language SOL will also have more general forms of data algebra expressions.

A declaration of abstract data type \( t \) with operations \( x_1, \ldots, x_n \) has the form

\[ \text{abtype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } N, \]

where \( \sigma_1(t), \ldots, \sigma_n(t) \) are the types of the operations and \( M \) is a data algebra expression. The scope of the declaration is \( N \). For example, a SOL expression with a local declaration of complex numbers might look like

\[ \text{abtype complex with create real } \rightarrow \text{ real } \rightarrow \text{ complex, } \]

\[ \text{plus complex } \rightarrow \text{ complex } \rightarrow \text{ complex, re complex } \rightarrow \text{ real, im complex } \rightarrow \text{ real, } \]

\[ \text{is } \text{rep real } \land \text{ real } \ M_1 \ M_2 \ M_3 \ M_4 \]

\[ \text{in } N, \]

where \( M_1 \) is an expression

\[ M_1 ::= \lambda x \in \text{real. } \lambda y \in \text{real. } \langle x, y \rangle \]

implementing the create operation mapping two real numbers to a complex number\(^4\), \( M_2 \) implements complex addition, \( M_3 \) and \( M_4 \) extract the real and imaginary parts from a complex number, and \( N \) uses \textit{create}, \textit{plus}, \textit{re} and \textit{im}. Note that the types of the operations \textit{create}, \textit{plus}, \textit{re} and \textit{im} are written using the abstract type \textit{complex}, while the operations are implemented by manipulating elements of the representation type.

\(^4\)In SOL, the type \textit{real } \land \textit{real} is the type of pairs of reals and \textit{real } \rightarrow \textit{real} is the type of real functions. When parentheses are omitted, the connective \( \land \) has higher precedence than \( \rightarrow \). We write \( \sigma(t) \) to emphasize that \( t \) may occur free in the type expression \( \sigma \).
There are three typing rules for abstract data type declarations. Let us consider the SOL expression

\[
\text{abstype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } N,
\]

assuming that M is a basic data algebra expression

\[
\text{rep } \tau \ M_1 \ldots M_k.
\]

This expression only makes sense if \( k = m \) and the the types of \( M_1, \ldots, M_k \) match the declared types of the operations \( x_1, \ldots, x_n \) in some appropriate way. The matching rule in SOL is that the type of \( M_i \) must be \( \sigma_i(\tau) \), the result of replacing \( t \) in \( \sigma_i(t) \) by \( \tau \). This simple convention corresponds to implicit insertion of the cvt of CLU [Liskov et. al. 81] in each function expression \( M_i \).

We can recast the matching rule as a typing rule by assigning types to data algebra expressions. An appropriate type for a data algebra expression is one which tells us how the operations may be used, without describing the type used to represent the carrier. If each \( M_i \) has type \( \sigma_i(\tau) \), then the type of

\[
\text{rep } \exists: \sigma_1(t) \wedge \ldots \wedge \sigma_n(t) \quad \tau \ M_1 \ldots M_n
\]

will be \( \exists: \sigma_1(t) \wedge \ldots \wedge \sigma_n(t) \). This type expression may be read "there exists a type \( t \) with operations of types \( \sigma_1(t) \) and \( \ldots \) and \( \sigma_n(t) \)." This type expression provides just enough information to to verify the matching condition stated above, without providing any information about representation of the carrier or the algorithms used to implement the operations. Using existential types, the matching rule for abstype may be stated

\[(\text{AB.1}) \text{ in abstype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } N, \text{ the data algebra expression } M \text{ must have type } \exists: \sigma_1(t) \wedge \ldots \wedge \sigma_n(t).\]

We may occasionally omit the subscript \( \exists: \sigma(t) \) from rep when the type is clear from context.\(^5\)

The operator \( \exists \) binds the type variable \( t \) in \( \exists: \sigma(t) \), so \( \exists: \sigma(t) = \exists: \sigma(s) \).

An important constraint in abstract type declarations is that only the explicitly declared operations may be applied to elements of the type [Morris 73]. This is accomplished by stipulating that

---

\(^5\)Without the subscript \( \exists: \sigma(t) \), the type of \( \text{rep } \tau \ M_1 \ldots M_n \) may be ambiguous. For example, if the type of \( M \) is \( s \rightarrow s \), then \( \text{rep } s \ M \) might have type \( \exists: t \rightarrow t \), or \( \exists: t \rightarrow s \), or several other types.
(AB.2) In abstype \( t \) with \( x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \) is \( M \) in \( N \), if \( y \) is any free identifier in \( N \) different from \( x_1, \ldots, x_n \), then \( t \) must not appear free in the type of \( y \).

For example, if \( f \) is a function with type \( \text{stack} \to \text{int} \), then the expression

\[
\text{abstype stack with empty} \in \text{stack}, \ \text{push} \in \text{int} \times \text{stack} \to \text{stack}, \ \text{pop} \in \text{stack} \to \text{int} \times \text{stack}
\]

is \( M \)

in \( f(\text{empty}) \)

will not be correctly typed. Morris argues that type checking should provide authentication and secrecy [Morris 73]. In the context of the other typing rules of SOL, (AB.2) provides both authentication and secrecy within the scope of the declaration. Note that (AB.2) mentions only free occurrences of \( t \). The language SOL will have types with bound variables, and the names of bound variables are unimportant.

A third restriction is necessary since SOL declarations may be local. As noted in the description of ML, another language with local abstype declarations, values of the abstract type should not be available outside the scope of the declaration (see [Gordon, et. al. 79], page 56). Therefore, we adopt

(AB.3) In abstype \( t \) with \( x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \) is \( M \) in \( N \), \( t \) must not be free in the type of \( N \).

This restriction prevents \( N \) from being an expression like \( x_1 \) when \( t \) appears free in \( \sigma_1(t) \). If \( x_1 \) were allowed to be the body of the declaration, then the value of the entire expression

\[
\text{abstype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } x_1,
\]

would be the implementation of the first operation. This would make the implementation accessible outside the representation of the type. In addition to conflicting with general principles of abstract data types, allowing \( t \) to appear free in the type of the body interferes with type checking if we allow more general data algebra expressions.

In SOL, we will allow a wide variety of data algebra expressions. One useful expression that conventional languages do not provide is the conditional data algebra expression. If both

\[
\text{rep}_{\exists \text{t}\sigma(t)} \tau M_1 \ldots M_n
\]

and

\[
\text{rep}_{\exists \text{t}\sigma(t)} \rho P_1 \ldots P_n
\]

are data algebra expression with type \( \exists \text{t}\sigma(t) \), then

20
if ... then (\text{rep}_{\exists t.t.\sigma(t)} \tau M_1 \ldots M_n) \text{ else } (\text{rep}_{\exists t.t.\sigma(t)} \rho P_1 \ldots P_n)\text{ will be a data algebra expression of SOL with the same type. Conditional algebra expressions are useful for selecting between several alternative implementations of the same abstract type. For example, a program that uses an abstract type matrix may select either sparse or dense matrix representations after analyzing the input data.}

In addition to conditional data algebra expressions, SOL will allow data algebra parameters. One example that illustrates the use of data algebra parameters is the general tree search routine given in Section 3. The usual algorithms for depth-first search and breadth-first search are virtually identical, except that depth-first search uses a stack and breadth-first search uses a queue. The general tree search algorithm in Section 3 uses a formal parameter in place of a stack or queue. If a stack data algebra is supplied as an actual parameter, then the algorithm performs depth-first search. Similarly a queue parameter produces breadth-first search. Additional structures like priority queues may also by passed as actual parameters, resulting in "best-first" search algorithms.

Data algebra parameters are allowed in SOL simply by virtue of the fact that the typing rules do not prevent them. If \( z \) is a variable with type \( \exists t.t.\sigma(t) \land \ldots \land \sigma_n(t) \), then

\[ \text{abtype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } z \text{ in } N \]

is a well-typed expression of SOL. Since SOL allows parameters of all types, there is nothing to prevent the data algebra \( z \) from being a formal parameter. By typing data algebra expressions, and treating all types in SOL in the same way, we allow conditional data algebra expressions, data algebra parameters, and many other useful kinds of computation on data algebras.

The next section presents the language SOL, which is based on the second-order lambda calculus of [Reynolds 74]. Essentially, we have expanded Reynolds' language using ideas from Girard's proof theoretic language [Girard 71]. The connection between proof-theoretic languages and typed programming languages is discussed in Section 3, and some additional programming examples appear in Section 4. The goal of SOL is to demonstrate a flexible treatment of abstract data types and to clarify the semantics of languages with polymorphic functions and abstract data type declarations. The denotational semantics of SOL with existential types is studied in...
[Mitchell 84a], which is based on an earlier treatment of the language without 3-typing [Bruce and Meyer 84]. Some similar languages are Pebble [Burstall and Lampson 84], designed to capture some essential features of Cedar (an extension of Mesa [Mitchell, James G. et. al. 79]), and Kernel Russell, KR, of [Hook 84], based on Russell [Demers and Donahue 80a, Demers and Donahue 80b, Demers et. al. 78]. Both Pebble and KR have flexible data type constructs. Martin-Löf's constructive type theory [Martin-Löf 79] is an even more flexible language with typing rules that seem to require run-time checking. MacQueen's module proposal for ML [MacQueen 84] incorporates some features of SOL into a language that meets practical programming requirements.

2.2 The Typed Language SOL

We demonstrate that representations of abstract data types can be put on an equal footing with functions, pairs and other values by presenting a functional language SOL. The reader may be interested to know that SOL was developed using an analogy with constructive proof theory. This analogy, described later in the paper, gives rise to a large family of typed expression languages which seem to encompass an astounding range of advanced programming language features.

Although SOL is an applicative language, we believe that our treatment of data algebras pertains to imperative languages as well. This belief is based on the general similarity between binding constructs in functional languages and those of imperative languages. Reynolds argues that the ALGOL-like languages, for example, may be derived from simple imperative languages by superimposing the procedure-binding mechanism of fully-typed lambda calculus [Reynolds 81a]. In addition, Burstall and Lampson point out that "programming in the large," building large programs from modules, is generally functional programming, even when imperative languages are used [Burstall and Lampson 84].

There are two classes of expressions in SOL: type expressions and terms. Types may appear in terms, but terms do not appear in type expressions. The type expressions are defined by the

---

6 In contrast, terms may appear in the type expressions of KR [Hook 84], Martin-Löf's constructive type theory [Martin-Löf 79], and the system of Huet and Coquand [Coquand and Huet 84].
following BNF grammar

\[ \sigma ::= t \mid c \mid \sigma \rightarrow \tau \mid \sigma \land \tau \mid \sigma \lor \tau \mid \forall \sigma \mid \exists \sigma. \]

In our "abstract syntax" presentation of SOL, we use two sorts of variables, type variables \( r, s, t, \) ... and ordinary variables \( x, y, z, \) ... In the grammar above, \( t \) is any type variable and \( c \) is any type constant. Some possible type constants are int and bool, which we often use in examples. Intuitively, \( \sigma \rightarrow \tau \) is the type of functions from \( \sigma \) to \( \tau \), an element of the product type \( \sigma \land \tau \) is a pair with one component from \( \sigma \) and the other from \( \tau \), and an element of the disjoint union or tagged sum type \( \sigma \lor \tau \) is an element of \( \sigma \) or \( \tau \). The two remaining type expressions involve the operators \( \forall \) and \( \exists \) which bind type variables. The universal type \( \forall \sigma \) is a type of polymorphic functions and elements of \( \exists \sigma \) are data algebras (implementations of abstract data types). Since \( t \) is bound in \( \forall \sigma \) and \( \exists \sigma \), we consider \( \forall \sigma = \forall s. [s/t] \sigma \) and \( \exists \sigma = \exists s. [s/t] \sigma \), when \( s \) does not occur free in \( \sigma \).

In SOL, as in most typed programming languages, the type of an expression depends on the context in which it occurs. This is because the type of a variable depends on how it is declared, not the way it is used. For each type assignment \( \Lambda \) mapping ordinary variables to type expressions, we define a set SOL\(_{\Lambda} \) of terms that are well-typed in the context of \( \Lambda \). The language SOL\(_{\Lambda} \) is defined using a partial function \( \text{Type}_\Lambda \) from expressions to types. The function \( \text{Type}_\Lambda \) is defined by a set of deduction rules of the form

\[ \text{Type}_\Lambda(M) = \sigma, \ldots \vdash \text{Type}_\Lambda(N) = \tau. \]

meaning that if the antecedents hold, then the value of \( \text{Type}_\Lambda \) at \( N \) is defined to be \( \tau \). The language SOL\(_{\Lambda} \) of terms that are well-typed in context \( \Lambda \) is taken to be the domain of \( \text{Type}_\Lambda \).

A variable of any type is a term. Formally, we have the axiom

\[ \text{Type}_\Lambda(x) = \Lambda(x) \]

without antecedents. We also allow constants, provided that each constant has an associated closed type expression. The type of a constant \( c^\tau \) is the associated type \( \tau \). One particularly useful constant is the conditional \( \text{cond} \) with type \( \forall t. \text{bool} \rightarrow t \rightarrow t \rightarrow t \). This constant will be discussed after \( \forall \)-types are introduced.
2.2.1 Functions and Let

In SOL, we will take functions of a single argument as basic; functions of several arguments can be defined using functions of one argument. A function expression will explicitly declare the type of the formal parameter. Consequently, the type of the function body is determined in a typing context that incorporates the formal parameter type. If $A$ is a type assignment, then $A[\sigma/x]$ is a type assignment with

$$(A[\sigma/x])(y) = \sigma \text{ if } y \text{ is } x, \text{ and } A(y) \text{ otherwise.}$$

The deduction rules for function expressions and function applications are

$$T; \rho \vdash e_{A[\sigma/x]}(M) : \tau \quad \vdash \quad Type_{A}(\lambda x \in \sigma.M) = \sigma \rightarrow \tau$$

and

$$Type_{A}(M) = \sigma \rightarrow \tau, \quad Type_{A}(N) = \sigma \quad \vdash \quad Type_{A}(MN) = \tau.$$ 

Thus a typed lambda expression has a functional type, and may be applied to an argument of the correct type. An example function expression is the lambda expression

$$\lambda x \in \text{int. } x + 1$$

for the successor function on integers.

The semantics of SOL is described by a set of operational reduction rules. The reduction rules using substitution rely on the ability to rename bound variables in terms, i.e.

$$(\alpha_{\lambda}) \quad \lambda x \in \sigma.M = \lambda y \in \sigma.\{y/x\}M, \ y \text{ not free in } \lambda x \in \sigma.M$$

The operational semantics of function definition and application is captured by the following reduction rule (cf. [Barendregt 81]).

$$(\beta_{\lambda}) \quad (\lambda x \in \sigma.M)N \Rightarrow [N/x]M,$$

where we assume that substitution $[N/x]M$ includes renaming of bound variables to avoid capture. This reduction rule is the familiar "copy rule" of ALGOL 60. Intuitively, $(\beta_{\lambda})$ states that the expression $(\lambda x \in \sigma.M)N$ may be evaluated by substituting the argument $N$ for each free occurrence of the variable $x$ in $M$. For example,

$$(\lambda x \in \text{int. } x + 2) 5 \Rightarrow 5 + 2.$$ 

We write $\Rightarrow$ for the transitive closure of $\Rightarrow$.

We define let declarations by the abbreviation

let $x = M \text{ in } N :: = (\lambda x \in \sigma.N) M,$

where $Type_{A}(M) = \sigma$. The typing rules and operational semantics for let are inherited directly
from $\lambda$. Thus

$$\text{let } f = \lambda x : \text{int. } x + 3 \text{ in } f(f(2)) \Rightarrow (2+3)+3.$$ 

A similar declaration is the ML recursive declaration

$$\text{letrec } x \in \sigma = M \text{ in } N.$$ 

Although we will use letrec in programming examples, it is technically useful to define pure SOL as a language without recursion. This pure language will have some interesting theoretical properties that shed some light on the type structure of SOL.

### 2.2.2 Products and Sums

A simple kind of record type is the unlabeled pair. In SOL, we use $\wedge$ for pair, or product, types. Product types have associated pairing and projection functions

$$\text{Type}_\wedge(M) = \sigma, \text{Type}_\wedge(N) = \tau \quad \vdash \quad \text{Type}_\wedge(<M, N>) = \sigma \wedge \tau$$

$$\text{Type}_\wedge(M) = \sigma \wedge \tau \quad \vdash \quad \text{Type}_\wedge(\text{fst } M) = \sigma, \quad \text{Type}_\wedge(\text{snd } M) = \tau.$$ 

The operational semantics of pairing and projection are given by the reduction rules

$$\text{fst}<M, N> \Rightarrow M, \quad \text{snd}<M, N> \Rightarrow N.$$ 

For example

$$\text{let } p = <1, 2> \text{ in } \text{fst}(p) \Rightarrow 1.$$ 

A useful abbreviation using products is

$$\lambda <x_1 : \sigma_1, ..., x_n : \sigma_n>. M ::= \lambda y : \sigma_1 \wedge ... \wedge \sigma_n. M_y,$$

where $y$ is not free in $M$ and $M_y$ is obtained from $M$ by replacing each $x_i$ by the term $\text{fst}^i(y)$ when $i < n$, and $x_n$ by $\text{snd}(\text{fst}^{n-1}(y))$. For example, $\lambda <x : \sigma, y : \tau>. M$ is an abbreviation for $\lambda z : \sigma \wedge \tau. [\text{fst } z, \text{snd } z/x, y]M$. We also use the abbreviation

$$\text{let } (x_1 : \sigma_1, ..., x_n : \sigma_n) = M \text{ in } N$$

for

$$\text{let } f = \lambda <x_1 : \sigma_1, ..., x_n : \sigma_n>. M \text{ in } N.$$ 

Sum types $\vee$ have injection functions and a case expression. The SOL case statement is similar to the tagcase statement of CLU [Liskov et. al. 77].

---

7The term $\text{fst}^i(y)$ is $\text{fst}(...(\text{fst}(y))...)$ with $i$ occurrences of $\text{fst}$. 
\[ Type_A(M) = \sigma \leftarrow Type_A(\text{inleft}_{\sigma, \tau} M) = \sigma \lor \tau, \quad Type_A(\text{inright}_{\tau, \sigma} M) = \tau \lor \sigma \]

\[ Type_A(M) = \sigma \lor \tau, \quad Type_A[\sigma/x](N) = \rho, \quad Type_A[\tau/y](P) = \rho \]

\[ \leftarrow Type_A(\text{case M left } x \in \sigma. N \right \text{ right } y \in \tau. P \text{ end}) = \rho \]

In the expressions above, case binds \( x \) in N and \( y \) in P. As with \( \lambda \)-binding, we equate case expressions that differ only in the names of bound variables. The bindings in case expressions may be replaced by \( \lambda \)-bindings as suggested in [Reynolds 81a], making case a constant of the language instead of a binding operator.

The reduction rules for sums are

\[ \text{case } (\text{inleft}_{\sigma, \tau} M) \text{ left } x \in \sigma. N \right \text{ right } y \in \tau. P \text{ end } \Rightarrow [M/x]N \]

\[ \text{case } (\text{inright}_{\sigma, \tau} M) \text{ left } x \in \sigma. N \right \text{ right } y \in \tau. P \text{ end } \Rightarrow [M/x]P \]

For example,

\[ \text{let } z = \text{inleft}_{\text{int, bool}} 3 \text{ in} \]

\[ \text{case } z \text{ left } x \in \text{int}. x \right \text{ right } y \in \text{bool}. \text{if } y \text{ then } 1 \text{ else } 0 \text{ end} \]

\[ \Rightarrow 3 \]

Note that the type of the case statement is \( \text{int} \), whether \( z \) is \( \text{inleft} \) of an integer or \( \text{inright} \) of a boolean.

### 2.2.3 Polymorphism

Intuitively, \( \Pi t. M \) is a polymorphic expression that can be "instantiated" to values of various types. In an Ada-like syntax, the term \( \Pi t. M \) would be written

```
generic
  type t
  M
```

Polymorphic expressions are instantiated using \text{proj}. The type of \text{proj } \tau N \text{ is } \sigma(\tau) \text{ when } N \text{ has type } \forall t. \sigma(t). \text{ The Ada-like syntax for } \text{proj } \tau N \text{ is}

```
new N(t),
```

The formal definitions are

\[ Type_A(M) = \tau \leftarrow Type_A(\Pi t. M) = \forall t. \tau, \quad \text{t not free in } A(x) \text{ for any } x \text{ free in } M, \]

\[ Type_A(M) = \forall t. \sigma(t) \leftarrow Type_A(\text{proj } \tau M) = \sigma(\tau). \]

26
The restriction on the bound variable \( t \) in \( \Pi t.M \) eliminates nonsensical expressions like \( \lambda x\epsilon t.\Pi t.x \), where it is not clear whether \( t \) is free or bound. (see [Fortune, et. al. 83] for further discussion). Note that unlike Ada and CLU, a SOL polymorphic function may be instantiated using any type expression, regardless of whether a compiler can determine the run-time value of the type expression at compile-time.

One use of \( \forall \)-types is in the polymorphic conditional \( \text{cond} \). Recall that the constant \( \text{cond} \) has type \( \forall t.\text{bool}\rightarrow t\rightarrow t\rightarrow t \). We can now introduce the abbreviation

\[
\text{if } M \text{ then } N \text{ else } P \quad::=\quad \text{(proj } \tau \text{ cond) } M \text{ N P},
\]

where \( \text{Type}_A(N) = \text{Type}_A(P) = \tau \).

Polymorphic type binding may be used to define polymorphic functions such as the polymorphic maximum function. The type of a function \( \text{Max} \) which, given any type \( t \) and order relation \( r\epsilon t\land t\rightarrow \text{bool} \), finds the maximum of a pair of \( t \)'s is

\[
\text{Max}\epsilon \forall t[(t\land t\rightarrow \text{bool})\rightarrow (t\land t)\rightarrow t].
\]

A SOL expression for the polymorphic maximum function is

\[
\text{Max} \quad::=\quad \Pi t. \quad \lambda r\epsilon t\land t\rightarrow \text{bool}. \lambda p\epsilon t\land t. \quad \text{if } (p) \text{ then } \text{fst}(p) \text{ else } \text{snd}(p)
\]

If \( r\epsilon \tau\land \tau\rightarrow \text{bool} \) is an order relation on type \( \tau \), then

\[
(\text{proj } \text{Max } \tau) r <x, y>
\]

will return the maximum element of the pair \( <x, y> \) of elements of type \( \tau \). While \( \text{Max} \) is written with the expectation that the actual parameter \( r \) will be an order relation, the SOL type checking rules cannot ensure this.

**Intuitive Semantics of \( \Pi t.M \)**

The intuitive denotational semantics for \( \Pi t.M \) is an infinite product of the set

\[
\{ M(t) \} = \{ M(\tau) | \tau \text{ a type expression}\},
\]

where \( M(\tau) \) denotes the result of replacing \( t \) by \( \tau \) in \( M \). In the next section, we will see that representations of abstract data types are elements of infinite sums. To see the similarity between \( \forall \)-types and infinite products, we review the notion of infinite product from category theory (cf. [Arbib and Manes 75, Herrlich and Strecker 73, Mac Lane 71]). There are two parts to to the definition of product: product types (corresponding to product objects in categories)
and product elements (corresponding to product arrows). Given a family of types
\[ \{ \sigma(t) \} = \{ \sigma(\tau) \mid \tau \text{ a type expression} \}, \]
the infinite product type \( \forall t. \sigma(t) \) has the property that for each \( \sigma(\tau) \) there is a projection function \( \text{proj}\ \tau \) from \( \forall t. \sigma(t) \) to \( \sigma(\tau) \). Furthermore, given any family of elements \( \{ M(t) \} \) with \( M(\tau) \in \sigma(\tau) \), there is a unique product element
\[ \Pi t. M \in \forall t. \sigma(t) \]
with the property that
\[ \text{proj}\ \tau \ \Pi t. M = M(\tau). \]
In the denotational semantics of SOL, we will not always insist that the product element be unique (cf. [Bruce and Meyer 84, Mitchell 84a]).

The reduction rule for infinite products is based on the product axiom above. Since \( \Pi \) binds type variables, we need the renaming rule
\[ (\alpha_\Pi) \quad \Pi t. M = \Pi s.[s/t]M, \ s \text{ not free in } \Pi t. M. \]

The reduction rule for \( \Pi \) is
\[ (\beta_\Pi) \quad \text{proj}\ \tau \ \Pi t. M \Rightarrow [\tau/t]M, \]
where we assume that bound variables are renamed in \( [\tau/t]M \) to avoid capture of free type variables in \( \tau \).

2.2.4 Data Abstraction and Existential Types

Data algebras, or concrete representations of abstract data types, are elements of existential types.

\[ \text{Type}_A(M) = \sigma(\tau) \vdash \text{Type}_A(\text{rep}_{\exists t. \sigma(t)} \ \tau \ M) = \exists t. \sigma(t). \]

The more general data algebra expression \( \text{rep}_{\exists t. \sigma(t)} \ \tau \ M_1 \ldots M_n \) is defined as an abbreviation
\[ \text{rep}_{\exists t. \sigma(t)} \ \tau \ M_1 \ldots M_n \ ::= \ \text{rep}_{\exists t. \sigma(t)} \ \tau \langle M_1 \ldots M_n \rangle, \]
where \( \sigma = \sigma_1(t) \land \ldots \land \sigma_n(t) \).

Polymorphic data algebras may be written in Ada, Alphard, CIAU and ML. Since SOL has \( \Pi \)-binding of types, we can also write polymorphic representations in SOL. For example, let \( \text{stackrep}(t) \) be a representation of stacks of elements of \( t \), e.g.

\[ \text{stackrep}(t) ::= \text{rep} \ (\text{int} \land \text{array of} \ t) \ \emptyset(t) \ \text{push}(t) \ \text{pop}(t), \]

where \( \emptyset \) represents the empty stack and \( \text{push} \) and \( \text{pop} \) are functions representing the usual
push and pop operations. We assume that the type of \( \text{push}(t) \) is \( t \wedge s \rightarrow s \) and the type of \( \text{pop}(t) \) is \( s \rightarrow t \wedge s \). Then the expression

\[
\text{stack ::= } \Pi_t \text{stackrep}(t)
\]

with type

\[
\text{stack} \in \forall t \exists s (t \wedge (t \wedge s \rightarrow s) \wedge (s \rightarrow t \wedge s))
\]

is a polymorphic representation of stacks. Similarly, we could define a polymorphic representation

\[
\text{queue} \in \forall t \exists q (q \wedge (t \wedge q \rightarrow q) \wedge (q \rightarrow t \wedge q'))
\]

of queues. We write \( \text{stack}[\tau] \) for the expression \( \text{proj } \tau \text{ stack} \), and similarly \( \text{queue}[\tau] \) for \( \text{proj } \tau \text{ queue} \). Note that the type of \( \text{stack}[t] \) is the usual signature for stacks of \( t \)'s, except that we have bound the name of the carrier with an existential quantifier.

Abstract data type declarations are formed according to the rule

\[
\text{Type}_A(M) = \exists t. \sigma(t), \text{Type}_A(N) = \rho \quad \vdash
\]

\[
\text{Type}_A(\text{abstype } t \text{ with } x \in \sigma(t) \text{ is } M \text{ in } N) = \rho,
\]

provided \( t \) is not free in \( \rho \) or the type \( A(y) \) of any \( y \) free in \( N \) different from \( x \).

The more general form is defined by abbreviation

\[
\text{abstype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } N \quad ::= \quad
\]

\[
\text{abstype } t \text{ with } y \in \sigma_1(t) \land \ldots \land \sigma_n(t) \text{ is } M \text{ in } N_y
\]

where \( y \) is not free in \( N \) and \( N_y \) is obtained from \( N \) by replacing each \( x_i \) by the term \( \text{fst}^i(y) \) when \( i \in n \), and \( x_n \) by \( \text{snd}(\text{fst}^{n-1}(y)) \). This definition of \( \text{abstype} \) provides all the type constraints discussed in Section 1. Furthermore, the only restriction on representation expressions is that they have the correct type.

We can use the polymorphic representation of stacks to declare integer stacks. The expression

\[
\text{abstype } \text{int-stk with empty} \in \text{int-stk}, \text{push} \in \text{int} \land \text{int-stk} \rightarrow \text{int-stk},
\]

\[
\text{pop} \in \text{int-stk} \rightarrow \text{int} \land \text{int-stk}
\]

is stack[int]

in N

declarates a type of integer stacks with three operations. Note that the declaration introduces local names for the stack operations.
Programming with Data Algebras

One feature of SOL is that a program can select one of several data type representations at run-time. For example, a parser which uses a symbol table can be parameterized by the representation of the type sym-table. The procedure can be passed different implementations, depending on whether a hash table or binary tree implementation is likely to be more efficient. The unrestricted abstract data type manipulation capabilities of SOL also make a common feature of file systems or linkage editors an explicit part of the programming language. The Cl.U library, for example, is a design for handling multiple implementations of various abstract data types [Liskov et. al. 77]. In SOL, multiple implementations may be manipulated directly by programs.

In addition to allowing programs to choose among representations of a single abstract data type, SOL has a certain flexibility resulting from the fact that only signatures are checked. Since stack[t] and queue[t] have the same type, we can write procedures that will accept representations of either stacks or queues as actual parameters. Thus data type parameters can also be useful for allowing the same procedure to serve several different uses. For example, a general tree search function is written below. The common algorithm for depth-first search uses a stack, while the usual approach to breadth-first search uses a queue. The general algorithm below has a data algebra parameter instead of a stack or queue. If a stack is supplied, then we get a depth-first search function, while a queue parameter produces a breadth-first search function. Passing a priority queue as a parameter produces a best-first search.

The function below searches a tree until it finds a node whose label is target. We assume that the search will eventually succeed. The tree is represented using nodes of type v. Each tree node has a label and two pointers. Thus we assume functions Is-leaf?, label, left and right that test whether a node is a leaf, read the label of a node, or produce the left or right descendant of the node. We also assume that the root, root, of the tree is given. Note that the type of the parameter struct is the same as the type of stack[v]. The local function next chooses a next node to look at, given the current node and the contents of the structure (e.g., stack or queue). The simple recursive function find just calls next repeatedly until the target is found.
Search(stack[Es[\exists s(\nu \Leftarrow \nu \land s) \land (s \rightarrow \nu \land s)]]) =

abstype s with empty\in\in Es, add\in\in \nu \land s \rightarrow \nu \land s is struct

let next(node\in\in \nu, st\in\in Es) =
  if Is_leaf?(node) then remove(st)
  else remove(add(left(node), add(right(node), st)))

let rec find(node\in\in \nu, st\in\in Es) =
  if label(node) = target then \langle node, st \rangle
  else find(next(node, st))

end

end

If the search function is applied to stack[\nu], then the result is a depth-first search algorithm. Similarly, Search(queue[\nu]) is a breadth-first search algorithm. In addition, we could also supply other kinds of search structures. If we have a search heuristic, then we can generate a "best-first" search algorithm by passing a representation whose remove operation returns the "most promising" node searched so far. In an application where a program can develop better search heuristics over time, it will be useful to compute better priority queue representations at run time.

2.2.5 Intuitive Semantics of Existential Types

Intuitively, the meaning of the abstype expression

abstype t with x\in\in \sigma(t) is (rep_{\exists.\sigma(t)} \tau M) in N
is the meaning of N in an environment where the meaning of t is determined by τ and the meaning of x by M. Operationally, we can evaluate abstype expressions using substitution. Since abstype binds variables, we have the renaming equivalence

\[ \text{abstype } t \text{ with } x \in \sigma(t) \text{ is } M \text{ in } N = \text{abstype } s \text{ with } y \in \sigma(s) \text{ is } M \text{ in } [y/x][s/t]N, \]

provided s and y are not free in M and N. The operational reduction rule is

\[ \text{abstype } t \text{ with } x \in \sigma(t) \text{ is } (\text{rep}_\exists \lambda \sigma(t) \tau) M \text{ in } N \Rightarrow [M/x][\tau/t]N, \]

where substitution includes renaming of bound variables as usual.

Existential types are closely related to infinite sums. We can see the relationship by reviewing the categorical definition of infinite sum [Arbib and Manes 75, Herrlich and Strecker 73, Mac Lane 71]. The general definition of sum includes sum types (corresponding to sum objects in categories) and sum functions (corresponding to sum arrows). If Q(t) is a term or type expression, possibly with t free, then we use Q(τ) to denote the result of replacing free occurrences of t by τ. By definition, an infinite sum \( \exists t. \sigma(t) \) of a family of types

\[ \{ \sigma(\tau) : \tau \text{ a type expression} \}, \]

comes equipped with a family of injection functions. For each \( \sigma(\tau) \) there must be an injection function

\[ (\text{rep}_\exists \lambda \sigma(t) \tau) \in \sigma(\tau) \rightarrow \exists t. \sigma(t). \]

Furthermore, for every set of functions \{N(τ)\} with \( N(\tau) \in \sigma(\tau) \rightarrow \rho \), where \( \rho \) is independent of t, there must be a sum function \( \Sigma t. N(t) \) such that

\[ (\Sigma t. N(t)) (\text{rep}_\exists \lambda \sigma(t) \tau) M = N(\tau) M. \]

In SOL, we use abstype expressions to denote sum functions. The expression \( \Sigma t. N \) may be considered an abbreviation

\[ \Sigma t. N ::= \lambda z \in \exists t. \sigma. \text{abstype } t \text{ with } x \in \sigma \text{ is } z \text{ in } N x \]

where x, z are fresh, Type_\Lambda(N) = \( \sigma(t) \rightarrow \rho \) and t is not free in \( \rho \). Note that the abstype expression for \( \Sigma t. N \) has precisely the property required of a sum function. It is interesting to compare abstype with case since \( \forall \)-types with inleft, inright and case correspond to finite categorical sums. Essentially, abstype is an infinitary version of case.

As an aside, we note that the binding construct abstype may be replaced by a constant sum. This treatment of abstype points out that the binding aspects of abstype are essentially \( \Pi \) and \( \lambda \) binding. If N is a term with type \( \sigma(t) \rightarrow \rho \), and t is is not free in \( \rho \), then both \( \Pi t. N \) and \( \Sigma t. N \) are
well-typed terms. Therefore, for each $\rho$, it suffices to have a function $\text{sum}_{\sigma(t)} \rho$ that maps $\Pi t.N \in \forall t.[\sigma(t) \rightarrow \rho]$ to $\Sigma t.N \in \exists t.\sigma(t) \rightarrow \rho$. Essentially, this means $\text{sum}_{\sigma(t)}$ must satisfy the equation

$$(\text{sum}_{\sigma(t)} \rho \times)(\text{rep}_{\exists t.\sigma(t)} \tau y) = (\text{proj} \times \tau) y$$

for any $x, y$ of the appropriate types. Given $\text{sum}$, the declaration $\text{abstyp}$ may be defined as

$$\text{abstyp} s \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } N ::=$$

$$(\text{sum} \prod s. \lambda (x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t)).N) M.$$ 

Like case, the essence of $\text{abstyp}$ is not that it binds variables, but that it provides sum functions.

2.2.6 Properties of SOL

Two important typing properties of SOL can be proved as theorems. The first theorem is a formal statement implying that that all type checking can be done at compile time. We take it for granted that it is a simple matter, given a term $M$ and a type assignment $A$, to calculate $Type_A(M)$. This much should be clear from the way we have defined the language. However, in order to be sure that this is all the type checking that must be done, we must verify that the reduction rules can be applied without regard to the types of terms. The Static Typing Theorem shows that terms can be evaluated without using type information: it is not necessary to check types at run time.

We need a few formal definitions to state the theorem. Given a term $M$, we let $Eraste(M)$ denote the untyped expression produced by erasing all type information from $M$. The function $Eraste$ has the simple inductive definition
$\text{Erase}(x) = x$

$\text{Erase}(c) = c$

$\text{Erase}(\lambda x : \sigma . M) = \lambda x. \text{Erase}(M)$

$\text{Erase}(M N) = \text{Erase}(M) \text{Erase}(N)$

$\text{Erase}(\langle M, N \rangle) = \langle \text{Erase}(M), \text{Erase}(N) \rangle$

$\text{Erase}(\text{fst} M) = \text{fst} \text{Erase}(M)$

$\text{Erase}(\text{snd} M) = \text{snd} \text{Erase}(M)$

$\text{Erase}(\text{inleft}_{\sigma, \tau} M) = \text{inleft} \text{Erase}(M)$

$\text{Erase}(\text{inright}_{\sigma, \tau} M) = \text{inright} \text{Erase}(M)$

$\text{Erase}(\text{case } M \text{ left } x : \sigma . N \text{ right } y : \tau . P \text{ end}) =$

$\text{case} \text{Erase}(M) \text{ left } x . \text{Erase}(N) \text{ right } y . \text{Erase}(P) \text{ end}$

$\text{Erase}(\Pi M) = \text{Erase}(M)$

$\text{Erase}(\text{proj } \tau M) = \text{Erase}(M)$

$\text{Erase}(\text{rep}_{\exists \sigma (t)} \rho M) = \text{Erase}(M)$

$\text{Erase}(\text{abstype } t \text{ with } x : \sigma \text{ is } M \text{ in } N) =$

$= (\lambda x . \text{Erase}(N)) \text{Erase}(M)$

We define $\Rightarrow^E$ by erasing types from terms in each reduction rule, e.g.,

$$(\lambda x . M) N \Rightarrow^E [N/x]M.$$  

Let $\Rightarrow^E$ be the transitive closure of $\Rightarrow^E$. Then we have the following theorem.

**Static Typing Theorem:** Let $M, N$ be two terms of $\text{SOL}_\Lambda$ with the same type. Then $M \Rightarrow N$ iff $\text{Erase}(M) \Rightarrow^E \text{Erase}(N)$.

A related theorem shows that when we evaluate terms using the operational reduction rules, the types of terms are preserved.

**Type Preservation Theorem:** Let $M$ be a term of $\text{SOL}$ with $\text{Type}_\Lambda(M) = \sigma$. If $M \Rightarrow N$, then $\text{Type}_\Lambda(N) = \sigma$.

While both theorems may seem entirely obvious, some of the typed languages discussed in the next subsection do not have one or the other of these simple typing properties. For example, the Type Preservation Theorem fails for the Russell-based language of [Hook 84].

The SOL reduction rules have a number of other interesting theoretical properties. For
example, the reduction rules have the Church-Rosser property [Girard 71, Prawitz 71].

Church-Rosser Theorem: Suppose $M$ is a term of SOL which reduces to $M_1$ and $M_2$. Then there is a term $N$ such that both $M_1$ and $M_2$ reduced to $N$.

In contrast to the untyped lambda calculus, no term of SOL can be reduced infinitely many times.

Strong Normalization Theorem: There are no infinite reduction sequences.

The strong normalization theorem was first proved by Girard [Girard 71]. In light of the strong normalization theorem, the Church-Rosser theorems follows from a simple check of the weak Church-Rosser property (see Proposition 3.1.25 of [Barendregt 81]). A normal form $M$ is a term that cannot be reduced. It follow from the two theorems above that all maximal reduction sequences from a given term all end in the same normal form.\textsuperscript{8} As pointed out by [Fortune, et. al. 83, O'Donnell 79], the proof of the strong normalization theorem cannot be carried out formally in either Peano arithmetic of second-order Peano arithmetic (Analysis). Furthermore, the class of number-theoretic functions that are representable in SOL without base types precisely the functions that may be proved total in second-order Peano arithmetic (see [Fortune, et. al. 83] for a survey of related results).

2.2.7 Alternative Views of Abstype

As noted in the introduction, several language design efforts are similar in spirit to ours. The language SOL is based on the Reynolds' polymorphic lambda calculus of [Reynolds 74] and Girard's proof theoretic language [Girard 71]. Some similar languages are Pebble [Burstall and Lampson 84], Kernel Russell, KR, [Hook 84]. ML with modules as proposed by MacQueen [MacQueen 84], and Martin-Löf's constructive type theory [Martin-Löf 79]. We compare abstype in SOL with an early proposal of Reynolds [Reynolds 74] and, briefly, with the constructs of Pebble and KR.

In defining the polymorphic lambda calculus, Reynolds proposed a kind of abstype declaration based on $\Pi$- and $\lambda$-binding [Reynolds 74]. As Reynolds notes, the expression

$$\text{abstype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } N$$

\textsuperscript{8}Our use of the phrase strong normalization follows [Barendregt 81]. Other authors use strong normalization for the property that all maximal reduction sequences from a given term end in the same normal form.
has the same meaning as
\[(\Pi \lambda x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \cdot N) \cdot \tau \cdot M_1 \ldots M_n\]
if M is of the form \(\text{rep}_\exists \cdot \sigma(t)\cdot \tau \cdot M_1 \ldots M_n\). However, an abtype expression should not be considered an abbreviation for this kind of \(\Pi\) expression for two reasons. First, it is not clear what to do if M is not of the form \(\text{rep}_\exists \cdot \sigma(t)\cdot \tau \cdot M_1 \ldots M_n\). A second drawback of using \(\Pi\) and \(\lambda\) to define abtype in this way is that the expression
\[(\Pi \lambda x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \cdot N) \cdot \tau \cdot M_1 \ldots M_n\]
is well-typed in cases where the corresponding abtype fails to satisfy (AB.3). As noted in the Introduction, rule (AB.3) keeps the "abstract" type from being exported outside the scope of the declaration.\(^9\)

It seems worthwhile to mention an entirely pragmatic justification for (AB.3). With conditional data algebra expressions, improper abtype expressions like
\[
\text{abtype } t \text{ with } x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \text{ is } M \text{ in } x_1.
\]
lead to serious type checking difficulties. If M is a conditional,
\[
\text{if } \ldots \text{ then } (\text{rep}_\exists \cdot \sigma(t) \cdot \tau \cdot M_1 \ldots M_n) \text{ else } (\text{rep}_\exists \cdot \sigma(t) \cdot \rho \cdot P_1 \ldots P_n)
\]
then the type of the entire abtype expression will be either \(\sigma_1(\tau)\) or \(\sigma_1(\rho)\), depending on which arm of the conditional applies. This makes the type of the such abtype expressions generally undecidable.\(^10\) Pebble and KR take view of data algebras that appears to differ from SOL. An intuitively appealing view of \(\text{rep}_\exists \cdot \sigma(t) \cdot \tau \cdot M_1 \ldots M_n\) is to interpret this expression as denoting a record whose first component is a type. This seems to lead one to introduce a "type of types," a path followed by [Burstall and Lampson 84] and [Hook 84]. The type of this \(\text{rep}_\exists \cdot \sigma(t)\) record is something like
\[
\text{Type} \land \sigma_1(t) \land \ldots \land \sigma_n(t).
\]
However, this type expression does not link the value of the first component of the record with the types of the remaining components since the free variable t is not bound anywhere. Pebble

---

\(^9\)While this definition of abtype using \(\lambda\) and \(\Pi\) is flawed, a suitable definition using \(\lambda\) and \(\Pi\) is described in the final section of [Reynolds 83].

\(^10\)The complexity or recursive degree of type checking without (AB.3) depends on the difficulty of deciding equivalence of expressions. In SOL without recursion, equivalence is decidable and so is typing without (AB.3). However, neither decision procedure is even remotely practical.
and KR associate abstract data types with "dependent product" types of the form
\[ t : \text{Type} \land \sigma_1(t) \land \ldots \land \sigma_n(t) \]
in which the variable \( t \) is considered bound.

Since Pebble does not supply projection functions for dependent products, the dependent product of Pebble seems to actually be a sum (in the sense of category theory). KR dependent products do have something that looks like a projection function: if \( A \) is a data algebra, then \( \text{Carrier}(A) \) is a type expression of KR. However, since \( \text{Carrier} \text{rep} \tau \text{ M} \) is not recognized as equal to \( \tau \), it seems that KR dependent products are not truly products. Perhaps it is most appropriate to think of KR dependent products as sums.

The general notion of dependent type seems to have been developed first by DeBruijn or Martin-Löf [DeBruijn 60, Martin-Löf 79]. Both the existential types in SOL and dependent products in Pebble and KR bear a close resemblance to the general sum types of Martin-Löf. All of the constructs have the same general form and set of type restrictions. One notable, technical distinction is that Martin-Löf's types come from stratified universes of types, while the types in SOL do not. While the constructs of SOL look like syntactic restrictions of the constructs in [Martin-Löf 79], there is an important overall difference between the two languages.

### 2.3 Formulas as Types

The language SOL was developed using an analogy between logical formulas and types. The connection between logical formulas and types of expressions has been used in proof theory [Curry and Feys 58, DeBruijn 60, Girard 71, Howard 80, Lambek 80, Lauchli 65, Lauchli 70, Martin-Löf 79, Statman 79a, Stenlund 72]. We illustrate the analogy by considering a simple example. Implicational propositional logic uses formulas that contain only propositional variables and \( \rightarrow \), implication. The formulas of implicational propositional logic are defined by the grammar
\[ \sigma ::= t \mid \sigma \rightarrow \tau, \]
where we understand that \( t \) is a propositional variable. We will be concerned with an intuitionistic interpretation of formulas, so it is best not to think of formulas as simply being true or false whenever we assign truth values to each variable. The intuitionistic semantics of
formulas [Fitting 69, Troelstra 73] are somewhat complicated, and we will not go into this topic. Instead, we will characterize intuitionistic validity by means of a proof system.

Natural deduction is a style of proof system that is intended to mimic the common blackboard-style argument

- Assume \( \sigma \).
- By \( ... \) we conclude \( \tau \).
- Therefore \( \sigma \rightarrow \tau \).

We make an assumption in the first line of this argument. In the second line, this assumption is combined with other reasoning to derive \( \tau \). At this point, we have proved \( \tau \) but the proof depends on the assumption of \( \sigma \). In the third step, we observe that since \( \sigma \) leads to a proof of \( \tau \), the implication \( \sigma \rightarrow \tau \) follows. Since the proof of \( \sigma \rightarrow \tau \) is sound without proviso, we have "discharged" the assumption of \( \sigma \) in proceeding from \( \tau \) to \( \sigma \rightarrow \tau \). In a natural deduction proof, each proposition may depend on one or more assumptions. A proposition is actually demonstrated only if it appears free of assumptions.

The natural deduction proof system for implicational propositional logic consists of three rules, given below. For technical reasons, we use labeled assumptions. Let \( \mathcal{Y} \) be a set, intended to be the set of labels, and let \( A \) be a mapping from labels to formulas. We use the notation \( \text{Conseq}_A(M) = \sigma \) for the assertion that \( M \) is a proof with consequence \( \sigma \), given the association \( A \) of assumptions to labels. Proofs and their consequences are defined as follows.

\[
\text{Conseq}_A(x) = A(x)
\]

\[
\text{Conseq}_A(M) = \sigma \rightarrow \tau, \quad \text{Conseq}_A(N) = \sigma \quad \vdash \quad \text{Conseq}_A(MN) = \tau.
\]

\[
\text{Conseq}_{A[x/\alpha]}(M) = \tau \quad \vdash \quad \text{Conseq}_A(\lambda x \in \sigma.M) = \sigma \rightarrow \tau.
\]

In English, we have

- A label \( x \) is a proof of \( A(x) \) with assumption labeled \( x \).

- If \( M \) is a proof of \( \sigma \rightarrow \tau \) and \( N \) is a proof of \( \sigma \), then \( MN \) is a proof of \( \tau \).

- If \( M \) is a proof of \( \tau \) with assumption \( \sigma \) labeled \( x \), then \( \lambda x \in \sigma.M \) is a proof of \( \sigma \rightarrow \tau \) with the assumption \( x \) discharged.

A formula \( \sigma \) is *intuitionistically provable* if there is a proof \( M \) with \( \text{Conseq}_{\emptyset}(M) = \sigma \). Some classically valid formulas are not intuitionistically provable, e.g., \(((s \rightarrow t) \rightarrow s) \rightarrow s\).
Of course, we have just defined the typed lambda calculus: the terms of typed lambda calculus are precisely the proofs defined above. Furthermore $\text{Conseq}_A$ and $\text{Type}_A$ are precisely the same function. The similarity between natural deduction proofs and terms extends to the connectives $\land$ and $\lor$ and quantification over propositions. The proof rules for $\land$, $\lor$, $\forall$ and $\exists$ are precisely the formation rules for terms of these types. One interesting feature of the proof rule for $\lor$ of [Prawitz 65] is that it is the discriminating case statement of CLU [Liskov et. al. 81] rather than the problematic outleft and outright functions of ML [Gordon, et. al. 79]. The "out" functions of ML are undesirable since they rely on run-time exceptions. (cf. [Liskov et. al. 77], page 569).

The intuitionistic proof constructs for universal and existential types are repeated below for emphasis.

\[
\text{Conseq}_A(M) = \forall t. \sigma(t) \vdash \text{Conseq}_A(\text{proj} \tau M) = \sigma(\tau),
\]

\[
\text{Conseq}_A(M) = \tau \vdash \text{Conseq}_A(\Pi t. M) = \forall \tau. \tau, \text{ t not free in } A(x) \text{ for any } x \text{ free in } M,
\]

\[
\text{Conseq}_A(M) = \sigma(\tau) \vdash \text{Conseq}_A(\text{rep}_{\exists t. \sigma(t)} \tau M) = \exists t. \sigma(t)
\]

\[
\text{Conseq}_A(M) = \exists t. \sigma(t), \text{ Conseq}_A(N) = \rho \vdash \text{Conseq}_A(\text{abtype } s \text{ with } x \in \sigma(t) \text{ is } M \text{ in } N) = \rho,
\]

provided $t$ is not free in $\rho$ or the type $A(y)$ of any $y$ free in $N$ different from $x$.

The rules for $\forall$ are the usual universal instantiation and generalization. The third rule is existential generalization and the fourth a form of existential instantiation. Except for the fact that the explicit notation for proofs is chosen to suggest programming language expressions, these proof rules are exactly those found in [Prawitz 65]. Furthermore, the reduction rules of SOL are precisely the proper reduction rules used to simplify proofs in [Prawitz 71] and elsewhere.

2.3.1 Extensions to SOL

The formulas-as-types analogy can also be applied to other systems of logic. Two particularly interesting systems are the second-order natural deduction systems of [Prawitz 65], Chapter V. The simpler of these two systems amounts to adding first-order terms to the second-order
logic we have already been using. In this language, types are formulas that describe the behavior of terms.

In an ideal programming language, we would like to use specifications to describe abstract data types. The ideal or "intended" type of stack[t] is the specification

\[ \exists s \exists \text{empty} \exists \text{push}_t \forall s \exists \text{pop}_s \rightarrow t \forall s. \]

\[ \forall x \in s \ \forall y \in t \{ \text{pop}(\text{push}(x, y)) = \langle x, y \rangle \}, \]

or perhaps more properly, a similar specification with an induction axiom.

\[ \exists s \exists \text{empty} \exists \text{push}_t \forall s \exists \text{pop}_s \rightarrow t \forall s. \]

\[ \forall x \in s \ \forall y \in t \ \{ \text{pop}(\text{push}(x, y)) = \langle x, y \rangle \ \land \text{induction axiom} \}. \]

Both specifications are, in fact, type expressions in the language based on first and second-order logic. We expect the meaning of each type expression to correspond to a class of algebras [Gratzer 68] satisfying the specification. However, the language based on first- and second-order logic is a cumbersome programming language since constructing an element of one of these existential types involves proving that an implementation meets its specification. Some interesting research into providing environments for programming with specifications as types is [Bates and Constable 81, Constable 80]. Induction rules, used for proofs by "data type induction" [Guttag, et. al. 78], are easily included in specifications since induction is expressible in second-order logic.

The richer "ramified second-order" system in Chapter V of [Prawitz 65] includes \( \lambda \)-abstraction in the language of types. Using formulas-as-types, this leads to the richer languages of [McCracken 79, Mitchell 84a].

2.4 More Programming Examples

Some useful constructions involving abstract data types are to pass representations as parameters, parameterize the data types themselves, and return implementations as results of procedures.
2.4.1 Universal and Existential Parameterization

In SOL, we can distinguish between two kinds of type parameterization. Suppose \( M \) uses operations \( x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) \) on type \( t \), and \( t \) is not free in the type of any other free variable of \( M \). Then the terms

\[
M_1 = \Pi t. \lambda x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) M
\]

\[
M_2 = \Sigma t. \lambda x_1 \in \sigma_1(t), \ldots, x_n \in \sigma_n(t) M
\]

are both parameterized by a type and operations. However, there are significant differences between these two terms. To begin with, \( M_1 \) is well-formed even if \( t \) appears free in the type of \( M \), whereas \( M_2 \) is not. In addition, the two terms have different types. If the type of \( M \) is \( \rho \), then the types of the terms are

\[
M_1 \in \forall t[\sigma_1(t) \land \ldots \land \sigma_n(t) \rightarrow \rho]
\]

and

\[
M_2 \in \exists t[\sigma_1(t) \land \ldots \land \sigma_n(t)] \rightarrow \rho.
\]

We will say that \( M_1 \) is \emph{universally parameterized} and \( M_2 \) is \emph{existentially parameterized}.

Generic packages are universally parameterized data algebras. For example, given any type \( t \) with operations

\[
\text{plus} \in t \land t \rightarrow t
\]

\[
\text{times} \in t \land t \rightarrow t,
\]

we can write a data algebra \emph{matrix}(t) implementing matrix operations over \( t \). Four operations we might choose to include are

\[
\text{create} \in t \land \ldots \land t \rightarrow \text{mat}
\]

\[
\text{mplus} \in \text{mat} \land \text{mat} \rightarrow \text{mat},
\]

\[
\text{mtimes} \in \text{mat} \land \text{mat} \rightarrow \text{mat},
\]

\[
\text{det} \in \text{mat} \rightarrow t.
\]

If \( mbody = \text{rep} \ t \ M_1 \ldots M_4 \) implements \emph{create}, \emph{mplus}, \emph{mtimes} and \emph{det} using \emph{plus} and \emph{times}, then

\[
\text{matrix} ::= \Pi t. \lambda \text{plus} \in t \land t \rightarrow t \lambda \text{times} \in t \land t \rightarrow t. \ mbody
\]

is a universally parameterized data algebra. The type of \emph{matrix} is

\[
\forall t. (t \land t \rightarrow t) \rightarrow (t \land t \rightarrow t) \rightarrow
\]

\[
\exists s[ (t \land \ldots \land t \rightarrow s) \land (s \land s \rightarrow s) \land (s \land s \rightarrow s) \land (s \rightarrow t)].
\]

Note that \( mbody \) cannot be existentially parameterized by \( t \) since \( t \) appears free in the type of

\( mbody \).
Functions from data algebras to data algebras are existentially parameterized. One simple manipulation of data algebras is to remove operations from the signature. For example, a doubly-ended queue, or deque, has two insert and two remove operations. The type of an implementation \( dq \) of dequeues with \( \text{empty}, \text{insert1}, \text{insert2}, \text{remove1}, \text{remove2} \), is

\[
dq\text{-type} \ ::= \ \forall t \exists d [d \land (t \wedge d \rightarrow d) \land (t \wedge d \rightarrow d) \land (d \rightarrow t \wedge d) \land (d \rightarrow t \wedge d)]
\]

A function that converts dequeue implementations to queue implementations is a simple example of an existentially parameterized structure. Given \( dq \), we can implement queues using the form

\[
Q(x, t) \ ::= \ \text{abtype} \ d \ \text{with empty} \in \ldots, \ \text{insert1} \in \ldots, \ \text{insert2} \in \ldots,
\text{remove1} \in \ldots, \text{remove2} \in \ldots,
\text{is} \ \text{(proj} \ t \ \text{x)}
\]
\[
\text{in rep} \ d \ \text{empty insert1 remove2}
\]

with \( dq \) substituted for \( x \). Thus the term

\[
dq\text{-to-q} \ ::= \ \lambda x \in dq\text{-type. } \Pi t. \ Q(x, t)
\]

with type

\[
dq\text{-type} \rightarrow \ \forall t \exists s [s \land (t \wedge s \rightarrow s) \land (s \rightarrow t \wedge s)]
\]

is a function from data algebras to data algebras. Suppose that \( queue \) is the data algebra produced by applying \( dq\text{-to-q} \) to \( dq \). Since the type of \( queue \) is a closed type expression, the fact that \( queue \) uses the same representation type as \( dq \) seems effectively hidden.

Some other useful transformations on data algebras are the transform investigated by Bentley and Shaw (Part VIII of [Shaw 81]) and analogs of the theory building operations \( \text{combine}, \text{enrich} \) and \( \text{derive} \) of CLEAR [Burstall and Goguen 77, Burstall and Goguen 81]. The transform considered by Bentley and Shaw produces a dynamic data structure, one that supports updates and queries in any order, from a static data structure, one that supports queries only after the all insert operations. As a SOL program, this transform is existentially parameterized.

Although a general \( \text{combine} \) operation, for example, cannot be written in SOL, we can write a combine operation on data algebras for each pair of data algebra types we wish to combine. For example, we can write a procedure to combine two data algebras of types \( \exists t \sigma(t) \) and \( \exists t \rho(t) \) into a single data algebra with two carriers. The type of this function

42
Combine_1 = \lambda x \in \exists t. \sigma(t) \ \lambda y \in \exists t. \rho(t).
  abstype s with z \in \sigma(s) is x in
  abstype t with w \in \rho(t) is y in
  rep s (rep t <z, w>)

is
  Combine_1 \in \exists t. \sigma(t) \rightarrow \exists t. \rho(t) \rightarrow \exists s. \exists t. [\sigma(s) \land \rho(t)].

If both data algebras are polymorphic, say of types
  \forall r. \exists t. \sigma(r, t) \quad and \quad \forall r. \exists t. \rho(r, t),
then each data algebra parameter will be parameterized by some type. For example, both \forall r. \exists t. \sigma(r, t) and \forall r. \exists t. \rho(r, t) may be the types of polymorphic stacks, queues, etc. In this case, we can write combine so that in the combined data algebra, the type parameter will be shared. The combine function with sharing

Combine_2 = \lambda x \in \forall r. \exists t. \sigma(r, t) \ \lambda y \in \forall r. \exists t. \rho(r, t).
  \Pi r. \ abstype s \ with \ z \in \sigma(r, s) \ is \ x[r] \ in
  abstype t \ with \ w \in \rho(r, t) \ is \ y[r] \ in
  rep s (rep t <z, w>)

has type
  Combine_2 \in \forall r. \exists t. \sigma(r, t) \rightarrow \forall r. \exists t. \rho(r, t) \rightarrow \forall s. \exists \exists t. [\sigma(r, s) \land \rho(r, t)].

A similar, but slightly more complicated, combine function can be written for the case where the two parameters are both universally parameterized by a type and several operations on the types. For example, a polymorphic matrix algebra could be combined with a polymorphic polynomial algebra to give a combined polymorphic matrix and polynomial package parameterized by a single type t with two binary operations (i.e. plus and times). Furthermore, the function could enrich the combined package by adding a function that finds the characteristic polynomial of a matrix.

It seems that universal parameterization may be used to effect some kind of sharing of types, while existential parameterization obscures sharing of representations. This is an intriguing notion which deserves further investigation.
2.4.2 Data Structures using Existential Types

Throughout this paper, we have viewed data algebras as implementations of abstract data types. An alternative view is that they are records tagged with types. These tagged records can be useful for building data structures that include elements tagged by types. Generally, these structures do not seem directly related to any sort of abstract data type. The following example uses existentially typed data structures to represent streams.

Intuitively, streams are infinite lists. In an applicative language, it is convenient to think of a stream as a kind of process which has a set of possible internal states and a specific value associated with each state. Since the process implements a list, there is a designated initial state and a deterministic state transition function. Therefore, a stream is a type $s$ (of states) with a designated individual (start state) of type $s$, a next-state function of type $s \rightarrow s$, and a value function of type $s \rightarrow t$, for some $t$. An integer stream, for example, will have a value function of type $s \rightarrow \text{int}$.

The Sieve of Eratosthenes can be used to produce an integer stream whose values are prime numbers. This stream is constructed using a sift operation on streams. Given an integer stream $s_1$, $\text{Sift}(s_1)$ is a stream of integers which are not divisible by the first value of $s_1$. If $\text{Num}$ is the stream 1, 2, 3, ..., then the sequence formed by taking the first values from each of the streams $\text{Num}, \text{Sift}(\text{Num}), \text{Sift}(\text{Sift}(\text{Num})), ...$ is the sequence of all primes.

Since streams are represented using existential types, $\text{Sift}$ is a function over existential types.
Sift =
\lambda \text{stream} \in \text{stream}[s \land (s \rightarrow s) \land (s \rightarrow \text{INT})].

\text{abstype } s \text{ with start} \in \text{stream}, \text{ next} \in \text{stream} \rightarrow s, \text{ value} \in \text{stream} \rightarrow \text{INT} \text{ is stream}

\text{in let } n = \text{value(start)}

\text{in letrec } f = \lambda \text{state} \in \text{stream}.

\text{if } n \text{ divides values(state) then } f(\text{next(state)})

\text{else state}

\text{in}

\langle s; \text{value}(f(\text{start})), f \rangle

\text{end}

\text{end}

\text{end}

The stream Sieve uses integer streams as the representation of internal states and Sift as the successor function on states.

Sieve =

\text{abstype } s \text{ with start: } s, \text{ next: } s \rightarrow s, \text{ value: } s \rightarrow \text{INT}

\text{is } \exists t[ t \land t \rightarrow t \land t \rightarrow \text{INT}]

\text{Start} = \langle \text{INT}; 1, \text{Successor}, \lambda x \in \text{INT}. x \rangle

\text{Next} = \text{Sift}

\text{Value} = \lambda \text{state} \in \exists t[ t \land t \rightarrow t \land t \rightarrow \text{INT}].

\text{abstype } r \text{ with r\_start, r\_next, r\_val is state}

\text{in}

r\_val(r\_start)

\text{end}

The infinite list of integer values of Sieve; Value(Start), Value(Next(Start)), ..., Value(Next(Start)), ...; lists all primes.
2.5 Conclusion and Directions for Further Investigation

The language SOL is a natural variant of Reynolds' polymorphic lambda calculus [Girard 71, Reynolds 81b], which has been studied in [Bruce and Meyer 84, Donahue 79, Fortune, et. al. 83, Haynes 84, McCracken 79, Reynolds 83, Reynolds 84]. In addition to the polymorphic functions of Reynolds' calculus, SOL has a very general form of abstract data type declaration. Data algebras, which may be viewed as concrete representations of abstract data types, may be passed as function parameters and returned as results. This makes the language more flexible than many contemporary typed languages. We believe that, although the design of SOL does not address certain practical programming objectives, the language demonstrates useful extensions to current programming languages. SOL also seems very useful for studying the mathematical semantics of data type declarations.

The flexibility of SOL comes about primarily because we treat data algebras as values which have types themselves. The types of data algebras in SOL are existential types, which are closely related to infinite sums. Intuitively, the injection function that maps a set and operations to an element of an existential type is a kind of "information hiding" function. While a product type comes equipped with projection functions that allow the components of a pair or tuple to be recovered, a sum or existential type does not have projection functions. Consequently, it does not seem possible for SOL programs to extract implementation information from data algebras. One direction for further investigation is to try to make this statement about sums and information hiding precise.

Another promising research direction is to use SOL to formalize and prove some natural properties of abstract data types. For example, if M and N implement two data algebras with the same observable behavior (cf. [Kapur 80]), then the meaning of a program using M should correspond appropriately to the meaning of the same program using N. However, the language is sufficiently complicated that it is not clear how to define "observable behavior." Among other complications, data algebras are heterogeneous structures whose operations may be polymorphic or involve existential types. Reynolds, Donahue and Haynes have examined various related "representation independence" properties of SOL without existential types [Donahue 79, Haynes 84, Reynolds 83]. It remains to be seen whether these properties have
important bearing on abstract data types in SOL.

There are a number of technical questions about SOL that merit further study. The semantics of various fragments of SOL are studied in [Bruce and Meyer 84, Donahue 79, Haynes 84, McCracken 79, Mitchell 84a, Reynolds 74, Reynolds 84], but many questions remain. Both [Bruce and Meyer 84] and [Mitchell 84a] list a number of open problems. In addition, there are a number of questions related to automatic insertion of type information into partially-typed expressions of SOL. For example, given a term M of the untyped lambda calculus, is the an algorithm to determine whether type expressions and type binding can be added to M to produce a well-typed term of SOL? Some questions of this nature are discussed in [Leivant 83a, McCracken 84, Mitchell 84b].

A general problem in the study of types is a formal characterization of type security. In the full paper, we state formally and prove two theorems about SOL: expressions may be evaluated without considering type information, and the syntactic type of an expression is not effected by reducing the expression to simpler forms. These theorems imply that types may be ignored when evaluating SOL expressions (using our operational semantics). However, further research seems necessary to show that SOL programs are "type-safe" in a satisfying precise sense.

One interesting aspect of SOL is that it may be derived from quantified propositional (second-order) logic using the formulas-as-types analogy discussed in Section 3. Our analysis of abstype demonstrates that the proof rules for existential formulas in a variety of logical systems all correspond to declaring and using abstract data types. Therefore, the formulas-as-types languages provide a general framework for studying abstract data types. In particular, the language derived from first- and second-order logic seems to incorporate specifications into programs in a very natural way. The semantics and programming properties of this language seem worth investigating.
1 Collected Syntax of SOL

The type expressions are defined by the following BNF grammar

\[ \sigma ::= t | c | \sigma \rightarrow \tau | \sigma \land \tau | \sigma \lor \tau | \forall t. \sigma | \exists t. \sigma. \]

In our "abstract syntax" presentation of SOL, we use two sorts of variables, type variables r, s, t, ... and ordinary variables x, y, z, ... In the grammar above, t is any type variable and c is any type constant.

A type assignment A is a function from variables to type expressions. The set SOL_A of terms typed in context A is defined to be the domain of a partial function Type_A from strings to types. The functions Type_A for all type assignments A are defined by

\[ Type_A(c^\tau) = \tau \text{ for constant } c^\tau \text{ of type } \tau \]

\[ Type_A(x) = A(x) \text{ if } x \text{ is in the domain of } A \]

\[ Type_A(M) = \sigma \rightarrow \tau, \ Type_A(N) = \sigma \vdash Type_A(M \ N) = \tau, \]

\[ Type_A[\sigma/x](M) = \tau \vdash Type_A(\lambda x: \sigma. M) = \sigma \rightarrow \tau, \]

\[ Type_A(M) = \sigma, Type_A(N) = \tau \vdash Type_A(<M, N>) = \sigma \land \tau \]

\[ Type_A(M) = \sigma \land \tau \vdash Type_A(fst M) = \sigma, \ Type_A(snd M) = \tau, \]

\[ Type_A(M) = \sigma \vdash Type_A(inleft_{\sigma, \tau} M) = \sigma \lor \tau, \ Type_A(inright_{\tau, \sigma} M) = \tau \lor \sigma \]

\[ Type_A(M) = \sigma \lor \tau, \ Type_A[\sigma/x](N) = \rho, \ Type_A[\tau/y](P) = \rho \]

\[ \vdash Type_A(\text{case } M \text{ left } x: \sigma. N \text{ right } y: \tau.P \text{ end}) = \rho, \]

\[ Type_A(M) = \forall t. \sigma(t) \vdash Type_A(\text{proj } \tau M) = \sigma(t), \]

\[ Type_A(M) = \tau \vdash Type_A(\Pi t.M) = \forall t. \tau, \ t \text{ not free in } A(y) \text{ for any } y \text{ free in } M \]

\[ Type_A(M) = \sigma(\tau) \vdash Type_A(\text{rep}_{\exists t. \sigma(t)} \tau M) = \exists t. \sigma(t), \]

\[ Type_A(M) = \exists t. \sigma(t), Type_A(N) = \rho \vdash Type_A(\text{ahstype } t \text{ with } x: \sigma(t) \text{ is } M \text{ in } N) = \rho, \]

provided t is not free in \( \rho \) or the type \( A(y) \) of any \( y \) free in \( N \) different from \( x \).
Here $\mathcal{A}[\sigma/x]$ denotes the type assignment $\mathcal{A}_1$ with $\mathcal{A}_1(x) = \mathcal{A}(y)$ for $y$ different from $x$, and $\mathcal{A}_1(x) = \sigma$.

Ignoring type constraints, the syntax of terms is summarized by

$$M ::= c | x | MN | \lambda x : \tau. M | \text{proj } \tau M | \Pi t. M$$

$$\text{rep}_\exists \mathcal{L} \sigma(t) \tau M | \text{abstype } t \text{ with } x \in \sigma(t) \text{ is } M \text{ in } \mathcal{N},$$

where $c$ is an typed constant and $x$ is any variable.
3.1 Introduction

The second-order lambda calculus, discovered independently by Girard [Girard 71] and Reynolds [Reynolds 74], is an extension of the usual simple typed lambda calculus. Like other kinds of lambda calculus, the simple parameter-binding mechanism of this language corresponds closely to parameter binding in many program languages (cf. [Landin 65, Reynolds 81a, Trakhtenbrot, Halpern and Meyer 83]). In contrast to other versions of lambda calculus, the type structure of the second-order system corresponds to the type structures of programming languages with polymorphism and data abstraction (such as Ada and CLU [Ada 80, Liskov et. al. 81]). Further discussion of polymorphism in second-order lambda calculus may be found in [Fortune, et. al. 83, ODonnell 79, Leivant 81]. The correspondence between elements of existential types and concrete representations of abstract data types will be motivated and developed in a forthcoming joint paper with Gordon Plotkin.\footnote{Chapter 2 of this thesis.} Some information about existential types is given in the final section of [Reynolds 83].

The second order in second-order lambda calculus comes about because the language is essentially a notational variant of the natural-deduction proof system for a second-order intuitionistic logic [Fortune, et. al. 83, Howard 80]. Since there are several choices of basic connectives in logic, there are several proof systems for second-order logic, and hence several versions of second-order lambda calculus. Most of the previous semantic work has focused on the second-order lambda calculus based on the intuitionistic logic with implication (→) and universal quantification (∀). The other connectives, conjunction (Λ) and disjunction (∨), as well as existential quantification (∃) are often omitted. In programming language parlance,
function types and polymorphism have been studied, but pairing, tagged sums and abstract data
types have been left out. The language \( \mathcal{L} \) will be defined with the set of type connectives as a
parameter, so that we may study all variants of the language simultaneously.

Several definitions for models of second-order lambda calculus have been proposed [Bruce and
Meyer 84, Leivant 83b, Martin-Löf 75] and a few specific model constructions have been
presented [Haynes 84, McCracken 79]. A companion paper to this one is [Bruce and Meyer 84],
in which a general model definition is proposed and a completeness theorem proved. A major
advantage of [Bruce and Meyer 84] over previous work is that all previously proposed model
constructions (based on Scott's models of untyped lambda calculus) may be viewed as special
cases of their general definition. The present paper broadens the "environment model"
deinition and completeness theorem of [Bruce and Meyer 84] to a more general language.
Since all natural variants of second-order lambda calculus arise by choosing different sets of
constants for \( \mathcal{L} \), completeness theorems for each version of the language follow from the
single theorem in Section 5.

The drawback of the environment model definition is that it can only be stated by referring to
the meanings of terms. We would like to be able to study such notions as homomorphisms and
submodels without constantly returning to arguments by induction on the structure of terms.
To this end, we introduce a logical system \( \mathcal{L}' \) for axiomatizing models of \( \mathcal{L} \) without reference
to definition by abstraction. The axiomatization provides an "algebraic" characterization of
second-order models similar to the characterization of untyped lambda calculus models based
on combinatory algebra [Meyer 82]. It is important to note that the semantics of \( \mathcal{L}' \) is
straightforward. The fact that the model definition can be formalized in \( \mathcal{L}' \) should be taken as
evidence of the simplicity of the definition, not the obscurity of \( \mathcal{L}' \).

By analogy with other forms of lambda calculus, [Barendregt 81, Meyer 82, Statman 84], we do
not expect the class of models of \( \mathcal{L} \) to be closed under homomorphisms. Instead, we expect
the homomorphic images of models of \( \mathcal{L} \) to be similar to untyped lambda algebras
[Barendregt 81, Meyer 82]. The higher-order type theory \( \mathcal{L}' \) provides a general framework for
studying classes of typed structures which are analogous to untyped combinatory algebras or
untyped lambda algebras. An intriguing research direction is to develop a deductive system for \( \lambda \). We expect a completeness theorem for \( \lambda \) along the lines of [Henkin 50] to hold (see also Section 4 of [Scott 80]). The reason for this belief is that validity for \( \lambda \) formulas can be reduced to first-order validity by essentially the method used for ordinary type theory (cf. [Monk 76], Chapter 30).

### 3.2 Higher-Order Lambda Calculus

There are three classes of expressions in the language \( \lambda \Lambda \) of higher-order lambda calculus. We are primarily interested in the terms of \( \lambda \Lambda \) that correspond to programming language expressions. Since each of these terms has a type, we also have type expressions. In building a general framework for studying various type structures, we also want to allow operations on types in the language. As in [MacQueen and Sethi 82, McCracken 79], we associate a *kind* with each symbol that might appear in a type expression, and so we have the language of kinds. We begin by defining kinds.

We use the constant \( T \) for the kind consisting of all types. The set of kinds is given by the grammar

\[
\kappa ::= T | \kappa_1 \rightarrow \kappa_2.
\]

Next we have the set of expressions that have kinds, which, for lack of a better phrase, we call the set of *constructionals*. Let \( c_{\text{kind}} \) be a set of constant symbols \( c^\kappa \), each with a specified kind (which we write as a superscript when necessary) and let \( \tau_{\text{kind}} \) be a set of variables \( v^\kappa \), each with a specified kind. We assume we have infinitely many variables of each kind. The constructionals over \( c_{\text{kind}} \) and \( \tau_{\text{kind}} \) and their kinds, are defined by the following derivation system

\[
\begin{align*}
c^\kappa \in \kappa, & \quad v^\kappa \in \kappa \\
\mu \in \kappa_1 \rightarrow \kappa_2, & \quad v \in \kappa_1 \vdash \mu v \in \kappa_2 \\
\mu \in \kappa_2 \vdash \lambda v^\kappa. & \quad \mu \in \kappa_1 \rightarrow \kappa_2
\end{align*}
\]

For example \((\lambda v^T. v^T) v^T\) is an expression with kind \( T \).

The pure equational theory of constructionals is the familiar equational theory of the simple
typed lambda calculus (cf. [Barendregt 81, Friedman 75, Statman 79b]). In the absence of additional axioms, equality is decidable and every constructional has a $\beta, \eta$-normal form. Although it seems likely that the completeness theorem in Section 5 can be extended to allow arbitrary equations between constructionals as axioms, there are some syntactic difficulties with constructional axioms. In particular, the set of well-typed terms will depend on the set of equations between constructionals. To avoid this complication, we will only consider the pure theory of $\beta, \eta$-equality between constructionals.

A special class of constructionals are the type expressions, the constructionals of kind \(T\). Since we will often be concerned with type expressions rather than arbitrary constructionals, it will be useful to distinguish them by notational conventions. We adopt the conventions that

\(r, s, t, \ldots\) denote type variables

\(\rho, \sigma, \tau, \ldots\) denote type expressions.

As in the definition above, we will generally use \(\mu\) and \(\nu\) for constructionals. Some important constructional constants are

\(\Pi \in T \rightarrow T \rightarrow T\), and \(\Sigma \in [T \rightarrow T] \rightarrow T\).

The following abbreviations will help stress connections with logic and previous work.

\(\sigma \rightarrow \tau ::= F \sigma \tau\)

\(\forall t.\sigma ::= \Pi(\lambda t.\sigma)\)

\(\exists t.\sigma ::= \Sigma(\lambda t.\sigma)\).

We will always assume that \(\mathcal{C}_{\text{kind}}\) includes the constants \(\Pi\) and \(\Pi\) above.

We now define the terms of \(\mathcal{E}\) and their types. The terms will include variables and typed constants. Let \(\mathcal{V}\) be a set of variables, which we will call ordinary variables, and let \(\mathcal{C}_{\text{typed}}\) be a set of constants, each with a fixed, closed type. As in most typed programming languages, the type of an \(\mathcal{E}\) term will depend on the context in which it occurs. For each type assignment \(A\) mapping ordinary variables to type expressions, we define a set \(\mathcal{E}_A\) of terms that are well-typed in the context \(A\). The language \(\mathcal{E}_A\) is defined using a partial function \(Type_A\) from expressions to types. The functions \(Type_A\) are defined by a set of deduction rules of the form

\(Type_A(M) = \sigma, \ldots \vdash Type_A(N) = \tau\).
meaning that if the antecedents hold, then the value of $Type_A$ at $N$ is defined to be $\tau$. The language $\lambda A$ of terms that are well-typed in context $A$ is taken to be the domain of $Type_A$.

The typing functions $Type_A$ are defined by the following rules. If $x$ is a variable, $\sigma$ a type expression and $A$ a type assignment, then $A[\sigma/x]$ is a type assignment with $(A[\sigma/x])(y) = A(y)$ for any variable $y$ different from $x$, and $(A[\sigma/x])(x) = \sigma$.

\[
Type_A(c^\tau) = \tau
\]

\[
Type_A(x) = A(x) \text{ if } x \text{ is in the domain of } A
\]

\[
Type_A(M) = \sigma \rightarrow \tau, \quad Type_A(N) = \sigma \vdash Type_A(app\ M\ N) = \tau,
\]

\[
Type_A[\sigma/x](M) = \tau \vdash Type_A(\lambda x: \sigma.M) = \sigma \rightarrow \tau,
\]

\[
Type_A(M) = \forall t: \sigma(t) \vdash Type_A(proj\ \tau\ M) = \sigma(\tau),
\]

\[
Type_A(M) = \tau \vdash Type_A(\Pi t.M) = \forall t: \tau,
\]
provided $t$ is not free in $A(x)$ for $x$ free in $M$

Free and bound variables are defined as usual, with $\lambda$ binding ordinary variables and $\Pi$ binding type variables in term. For example, $x$ is bound in $\lambda x: \sigma.M$, the type variable $t$ is bound in $\lambda x: \forall t:t.x$, and $t$ is bound in $\Pi t.M$.

Since the set of terms $\lambda A$ depends on $\mathcal{C}_{\text{kind}}$ and $\mathcal{C}_{\text{typed}}$ it is more accurate to write $\lambda A(\mathcal{C}_{\text{kind}}, \mathcal{C}_{\text{typed}})$. However, for simplicity of notation, we will generally leave the sets of constants implicit. It is also convenient to leave the assignment $A$ implicit when no confusion will arise. A term of $\lambda A$ is closed if it has no free ordinary variables. If $M$ is closed, then $Type_A(M)$ does not depend on $A$ and so we may write $Type(M)$ for the type of $M$.

If $M$ is a term and $t$ does not occur free in $A(x)$ for any $x$ free in $M$, then $t$ is bindable in $M$ with respect to $A$. If $t$ is bindable in $M$, then it is often convenient to write $M(t)$ for $M$. In this case, we also write $M(\tau)$ for $[\tau/t]M$. As usual, substitutions $[N/x]M$ and $[\tau/t]M$ of $N$ for $x$ and $\tau$ for $t$ are defined to include renaming of bound variables to avoid capture. The following lemma is easily proved by induction on terms.

**Lemma 3-1:** If $t$ is bindable in $M \in \lambda A$ then $Type_A([\tau/t]M) = [\tau/t](Type_A(M))$. 

54
Since it is cumbersome to write \texttt{app} and \texttt{proj}, we adopt the abbreviations

\[
\text{MN} ::= \text{app} M N
\]

\[
M\tau ::= \text{proj} \tau M
\]

These abbreviations may not be possible in extensions to \( \texttt{\&}\Lambda \). In the presence of constructional axioms, it would seem best to subscript \texttt{app} and \texttt{proj} by types. In this case, adopting the above abbreviations might lead to ambiguity in the types of terms.

### 3.2.1 Second-Order Lambda Calculus and Existential Types

We use \( \texttt{\&}\Lambda(\rightarrow, \forall) \) to denote the basic language with constructional constants \( F, \Pi \) and no typed constants. The pure second order lambda calculus \( \texttt{\&}\Lambda(\rightarrow, \forall) \) defined by Reynolds in [Reynolds 74] and studied in [Bruce and Meyer 84, Donahue 79, Fortune, et. al. 83, Reynolds 84], consists of all terms \( M \) of \( \texttt{\&}\Lambda(\rightarrow, \forall) \) with the property that all free variables of \( M \) are either type variables or ordinary variables. The second order lambda calculus of [Girard 71] with product types and existential types as well is also a sublanguage of some \( \texttt{\&}\Lambda \). We discuss existential types below.

Existential types involve the constructional constant \( \Sigma\epsilon(T \rightarrow T) \rightarrow T \) and families of typed constants \texttt{inj} and \texttt{sum}. The typed constants allow us to construct elements of \( \exists t \sigma \) types and make use of them, just as pairing and projection functions allow us to construct pairs and make use of them. There is a slight complication with the types of \texttt{inj} and \texttt{sum} which illustrates a general problem with constants in \( \texttt{\&}\Lambda \). For each type \( \exists t \sigma \), we would like to add constants \( \texttt{inj}_{\exists t \sigma(t)} \) and \( \texttt{sum}_{\sigma(t)} \) with types

\[
\text{Type}(\text{inj}_{\exists t \sigma(t)}) = \forall t[\sigma \rightarrow \exists t \sigma],
\]

\[
\text{Type}(\text{sum}_{\sigma(t)}) = \forall s[\forall t[\sigma \rightarrow s] \rightarrow \exists t \sigma \rightarrow s]
\]

However, if the type \( \sigma \) has free variables other than \( t \), then the types of \( \texttt{inj}_{\exists t \sigma(t)} \) and \( \texttt{sum}_{\sigma(t)} \) are not closed. Since the interpretation of \( \exists t \sigma \) would depend on the environment, neither \( \texttt{inj}_{\exists t \sigma(t)} \) nor \( \texttt{sum}_{\sigma(t)} \) would really be constant.

Instead of associating open types with constants, we index constants \texttt{inj} and \texttt{sum} by closed constructional of kinds \( T^i \rightarrow T, i > 0 \), where \( T^i \rightarrow T \) abbreviates the expression \( T \rightarrow T \rightarrow \ldots \rightarrow T \) with \( i \)
occurrences of \(-\). Note that for any type expression \(\sigma\) with all free variables of kind \(T\), there is a closed constructional expression \(\lambda t^+\cdot \lambda t_{\sigma}\) with kind \(T^i\rightarrow T\) for some \(i\). We add \(\exists\)-types to the language by adding constants \(\text{inj}_f\) and \(\text{sum}_f\) for each \(f:\subseteq T^i\rightarrow T\) with types

\[
Type(\text{inj}_f) = \forall r^+\{\forall t(f(r^+, t)\rightarrow \exists t.f(r^+, t))\},
\]

\[
Type(\text{sum}_f) = \forall r^+\{\forall s[\forall t(f(r^+, t)\rightarrow s)\rightarrow \exists t.f(r^+, t)\rightarrow s]\}
\]

We use \(\forall \Lambda(\rightarrow, \forall, \exists)\) to denote the language with constructional constant \(\Sigma\) and typed constants \(\text{inj}_f\) and \(\text{sum}_f\). It is often convenient to write \(\text{inj}_{\exists t_{\sigma(t)}}\) as a meta-linguistic abbreviation for the term \(\text{inj}_{\lambda r^+\cdot \exists t_{\sigma}} r^+\), and similarly for \(\text{sum}_{\sigma(t)}\). Note that we only have constants \(\text{inj}_{\exists t_{\sigma(t)}}\) and \(\text{sum}_{\sigma(t)}\) for each \(\exists t_{\sigma}\) without free variables of higher kinds.

The additional constants of \(\forall \Lambda(\rightarrow, \forall, \exists)\) give us the following derived term formation rules.

\[
Type_A(M) = \sigma(\tau) \vdash Type_A(\text{inj}_{\exists t_{\sigma(t)}} \tau M) = \exists t_{\sigma(t)}
\]

\[
Type_A(M) = \forall t_{\sigma(t)}\rightarrow \rho \vdash Type_A(\text{sum}_{\sigma(t)} \rho M) = [\exists t_{\sigma(t)}]\rightarrow \rho, \ t \text{ not free in } \rho
\]

Other second-order types such as \(\land\) (pairing) and \(\lor\) (disjoint sum) may be added to \(\forall \exists \Lambda\) by making suitable choices of \(c_{\text{kind}}\) and \(c_{\text{typed}}\).

Some useful abbreviations in the language with existential types are

\[
\Sigma t.M ::= \text{sum}_{\sigma(t)} \rho \prod t.M,
\]

\[
\text{abstype } t \text{ with } x_{\subseteq \sigma} \text{ is } M \text{ in } N ::= (\Sigma t\lambda x_{\subseteq \sigma} N) M.
\]

The operator \(\Sigma\) binds type variables and is dual to \(\Pi\). The abstype construct is a very general form of abstract data type declaration. A term of the form \(\text{inj } \tau M\) may be considered an implementation or concrete representation of an abstract data type. Thus it is often a helpful mnemonic to write \(\text{rep } \tau M\) for \(\text{inj } \tau M\).

### 3.3 Axioms and Inference Rules

There are two inference systems, one for constructionals and one for terms. The axioms and inference rules for constructionals are the familiar ones for simply-typed lambda calculus. The axioms are

56
\( (\alpha_\kappa) \quad \lambda v^\kappa.\mu = \lambda v^\kappa.[u/v]\mu, \ u \text{ not free in } \mu \)

\( (\beta_\kappa) \quad (\lambda v^\kappa.\mu)v = [v/v]\mu \)

\( (\eta_\kappa) \quad \lambda v^\kappa.(\mu v) = \mu, \ v \text{ not free in } \mu \)

The inference rules are the congruence rules. These rules are written below for terms. Although the axiom \((\eta_\kappa)\) is not necessary, it makes the semantic interpretation of constructionals slightly simpler.

The pure equational theory of terms of \(\forall \Lambda\) is based on \(\alpha\)-conversion, renaming of bound variables, and \(\beta\)-conversion, a rule for evaluating expressions using substitution.

**Axioms**

\((\alpha_\lambda) \quad \lambda x \in \sigma.M = \lambda y \in \sigma.[y/x]M, \ y \text{ not free in } \lambda x \in \sigma.M,\)

\((\sigma_{\Pi}) \quad \Pi.t.M = \Pi.s.[s/t]M, \ s \text{ not free in } \Pi.t.M,\)

\((\beta_\lambda) \quad (\lambda x \in \sigma.M)N = [N/x]M,\)

\((\beta_{\Pi}) \quad (\Pi.t.M)t = [t/t]M,\)

The inference rules will be precisely those required to make equality a congruence relation. It is not necessary to include a reflexivity axiom because \(M = M\) follows from \((\beta_\lambda)\) by symmetry and transitivity.

**Equivalence Rules**

\(M = N \vdash N = M\)

\(M = N, N = P \vdash M = P\)

**Congruence Rules**

\(M = N \vdash \lambda x \in \sigma.M = \lambda x \in \sigma.N,\)

\(\Pi.t.M = \Pi.t.N, \ M\tau = N\tau\)

\(M = N, P = Q \vdash MP = NQ.\)

\(\mu^\kappa = \nu^\kappa \vdash M = N, \) where \(N\) is obtained from \(M\) by substituting \(\nu\) for one or more occurrences of \(\mu\)

An instance \(P = Q\) of an axiom is called an \(A\)-instance if \(Type_A(P) = Type_A(Q)\). An equation \(M = N\) between terms of \(\forall \Lambda_A\) is \(A\)-provable, written \(\vdash_A M = N\), if the equation follows from \(A\)-instances of axioms other than the extensionality axioms, and the above inference rules. We often write \(\vdash\) instead of \(\vdash_A\) when the type assignment \(A\) is clear from context. It follows easily
from the definition that if $\text{Type}_A(M) = \sigma$ and $\Gamma \vdash_A M = N$, then $\text{Type}_A(N) = \sigma$.

A set $\Gamma$ of $\mathcal{E} \Lambda_A$-equations is a set of equations between terms of $\mathcal{E} \Lambda_A$ such that $\text{Type}_A(M) = \text{Type}_A(N)$ for all $M = N \in \Gamma$. A theory $\text{Th for } \mathcal{E} \Lambda_A$ is a set of $\mathcal{E} \Lambda_A$ equations which is closed under $\Gamma \vdash_A$. The theory of the axioms

$\text{Extensionality Axioms}$

$(\eta_\lambda)$ \quad $\lambda x \in \sigma. M x = M$, \quad $x$ not free in $M$

$(\eta_\Pi)$ \quad $\Pi t (\text{proj } t M) = M$, \quad $t$ not free in $M$

is the extensional theory of $\mathcal{E} \Lambda(\rightarrow, \forall)$.

3.3.1 Infinite Products, Sums and Existential Types

Before discussing the theory associated with existential types, we note a similarity between the properties of $\Pi t. M$ and an infinite product. We will then see that existential types are similar to sums, and dual to universal types. We explain the similarity between universal types and products by briefly reviewing the notion of an infinite product from category theory [Arbib and Manes 75, Herrlich and Strecker 73, Mac Lane 71] using the notation of $\mathcal{E} \Lambda$. If $Q(t)$ is a term or type expression, possibly with $t$ free, then we use $Q(\tau)$ to denote the result of replacing free occurrences of $t$ by $\tau$.

Infinite products involve product types (corresponding to product objects in categories) and product elements (corresponding to product arrows). Given a set of types

$\{\sigma(t)\} = \{\sigma(\tau) | \tau \text{ a type expression}\},$

the product type $\forall t. \sigma(t) = \Pi \{\lambda t. \sigma(t)\}$ has the property that for each $\sigma(\tau)$ there is a projection function $\text{proj } \tau$ from $\forall t. \sigma(t)$ to $\sigma(\tau)$. Furthermore, given any set of elements $\{M(t)\}$ with $M(\tau) \in \sigma(\tau)$, there is a unique product element $\Pi t. M(t) \in \forall t. \sigma(t)$ such that

$\text{proj } \tau \quad \Pi t. M(t) = M(\tau).$

Note that equations is essentially $(\beta_\lambda)$.

There are two ways that $\forall t. \sigma(t)$ differs from a true categorical product. One is that only certain syntactically definable product elements need exist. This suggests that $\forall t. \sigma(t)$ may be a limit of a diagram that is not discrete, while products are limits of discrete diagrams. The true product
of \{\sigma(t)\} would contain products of sets like
\{N \in \sigma(\rho)\} \cup \{M(\tau) | \tau \text{ different from } \rho\}
that are not definable by a single term; Reynolds discusses the "parametric" character of the
definable functions of type \(\forall t.\sigma(t)\) in [Reynolds 83, Reynolds 84]. The second difference
between \(\forall t.\sigma(t)\) and a true infinite product is that if we do not assume the extensionality axiom
(\(\eta_\Pi\)), then there may be more than one element of \(\forall t.\sigma(t)\) which behaves like the product of
\{M(t)\}. Thus, in the absence of (\(\eta_\Pi\)), our \(\forall t.\sigma(t)\) is actually somewhat weaker than a limit.
While limits are unique up to isomorphism, there may be nonisomorphic sets that satisfy the
requirements for \(\forall t.\sigma(t)\).

We now discuss the theory \(Th_3\) for the language \(\forall \lambda (\rightarrow, \forall, \exists)\) with existential types. The
axioms for this theory are motivated by the intended properties of existential types. Intuitively,
existential types are similar to infinite sums in category theory [Arbib and Manes 75, Mac Lane
71]. The general notion of sum involves sum types and sum functions. By definition, an infinite
sum \(\exists t.\sigma(t)\) of a family of types
\(\{ \sigma(\tau) | \tau \text{ a type expression} \}\),
comes equipped with a family of injection functions. For each \(\sigma(\tau)\) there must be an injection
function
\((\text{inj}_{\exists t.\sigma(t)} \, \tau) \in \sigma(\tau) \rightarrow \exists t.\sigma(t)\).
Furthermore, for every set of functions \(\{N(\tau)\}\) with \(N(\tau) \in \sigma(\tau) \rightarrow \rho\), where \(\rho\) is independent of
t, there must be a sum function \(\Sigma t. N(t)\) such that
\((\Sigma t. N(t)) \, (\text{inj}_{\exists t.\sigma(t)} \, \tau \, M) = N(\tau) \, M.\)
We associate the set of functions \(\{N(\tau)\}\) with the term \(\Pi t. N\) and use \(\text{sum} \, \Pi t. N\) for \(\Sigma t. N\). In
general, \(\exists t.\sigma(t)\) will differ from a true categorical sum for the same reasons that \(\forall t.\sigma(t)\) differs
from a true categorical product.

The theory \(Th_3\) is the theory of the axiom
\(\beta_\Sigma\) \((\Sigma \sigma(\tau) \, \rho \, M) \, (\text{inj}_{\exists t.\sigma(t)} \, \tau \, N) = (M \, \tau) \, N,\)
based on the sum property above. The extensional theory for \(\forall \lambda (\rightarrow, \forall, \exists)\) is obtained by
adding the axiom
\(\eta_\Sigma\) \((\Sigma t)(\lambda x \in \sigma(\tau) \, M)(\text{inj}_{\exists t.\sigma(t)} \, t \, x) = M,\)
provided \(t\) and \(x\) are not free in \(M\).
to $\Theta_\varnothing$, along with $(\eta_\lambda)$ and $(\eta_\Pi)$.

3.4 Environment Models

Environment models will be defined in this section. We will later show that the environment model definition is equivalent to a combinatory model definition. In each form of model, we need to interpret both constructionals and terms. Since constructionals are essentially terms of the simple typed lambda calculus, their semantics are routine (see, e.g., [Barendregt 81, Friedman 75, Statman 84]). The same straightforward, conventional model for constructionals will be used in both environment models and combinatory models. We discuss the semantics of constructionals briefly before going onto the semantics of terms.

3.4.1 Models for Constructionals

A type structure $\mathcal{T}$ for a set $c_{kind}$ of constructional constants is a tuple $<u, c_1^\mathcal{T}, c_2^\mathcal{T}, \ldots>$, where $u$ is a family of sets $\{U_\kappa\}$ indexed by kinds with $U_{\kappa_1 \rightarrow \kappa_2}$ a set of functions from $U_{\kappa_1}$ to $U_{\kappa_2}$, and the interpretation $c_i^\mathcal{T}$ of a constant $c_i \in c_{kind}$ is an element of the appropriate $U_\kappa$. Since constructionals include all typed lambda expressions, $u$ must be a model of the simple typed lambda calculus. This requires that for all kinds $\kappa_1, \kappa_2$ and $\kappa_3$, there must be elements

$$K_{\kappa_1, \kappa_2} \in [\kappa_1 \rightarrow (\kappa_2 \rightarrow \kappa_1)]$$

$$S_{\kappa_1, \kappa_2, \kappa_3} \in [(\kappa_1 \rightarrow \kappa_2 \rightarrow \kappa_3) \rightarrow (\kappa_1 \rightarrow \kappa_2) \rightarrow \kappa_1 \rightarrow \kappa_3]$$

with the familiar properties

$$K uv = u$$

$$S uvw = (uw)(vw)$$

where $u$, $v$ and $w$ are variables of the appropriate kinds. The meaning of a constructional $\mu$ in environment $\eta$ for type structure $\mathcal{T}$ is defined as in [Barendregt 81, Friedman 75, Henkin 50, Statman 84].

Note that by definition, type structures are extensional. This avoids a bit of confusion when we index constants (like $\text{in}$ and $\text{sum}$) by functions of kinds $T^i \rightarrow T$. It is interesting to note that type structures (models of propositionally-typed lambda calculus) are easily described by the above
axiomatization of families of combinators \( K \) and \( S \). This description is an axiomatization in the usual "simple" type theory of [Friedman 75]; see also [Herkin 50, Statman 84].

If a particular type structure \( \mathcal{A} = \langle \mathcal{U}, \ldots \rangle \) is clear from context, we use closed kind expressions to refer to sets of \( \mathcal{U} \). For example, \( T \) denotes the base set \( U_T \) of \( \mathcal{U} \).

3.4.2 Higher-Order Lambda Frames

The definition of environment model for \( \mathcal{H} \Lambda \) is based on the environment model definitions for untyped lambda calculus and for second-order lambda calculus [Brucc and Meyer 84]. We review the untyped definition briefly following the terminology of [Meyer 82]. Untyped lambda calculus involves untyped applications \( MN \) and function expressions \( \lambda x. M \). Thus we need to be able to treat elements as functions and functions as elements. This is accomplished using a pair of maps, the element-to-function map \( \Phi \) and the function-to-element map \( \Psi \). An environment model for untyped lambda calculus \( \langle D, \Phi, \Psi \rangle \) consists of a set \( D \) together with functions \( \Phi \) and \( \Psi \). For some subset \( [D \to D] \subseteq D^D \), we require that

\[
\Phi : D \to [D \to D] \quad \text{and} \quad \Psi : [D \to D] \to D \quad \text{with}
\]

\[
\Phi \circ \Psi = \text{id}_{[D \to D]}.
\]

Furthermore, in order to be a model, the set \( [D \to D] \) of functions represented by elements of \( D \) must be rich enough to allow all lambda terms to be interpreted. This additional condition, which we refer to as (env), is made rigorous by defining the meanings of terms.

In environment models of \( \mathcal{H} \Lambda \), we will need typed version of \( \Phi \) and \( \Psi \) to interpret typed application and typed lambda abstraction. In addition, we will use "polymorphic" versions of \( \Phi \) and \( \Psi \) for products \( \Pi t. M \) and projection \( \text{proj}_{\tau} M \).

A frame \( \mathcal{F} \) for \( \mathcal{H} \Lambda (\mathcal{C}_{\text{kind}}, \mathcal{C}_{\text{type}}) \) is a tuple

\[
\mathcal{F} = \langle \mathcal{A}, \mathcal{A}, \{ \Phi, \Psi, c_{\mathcal{F}} \}, \mathcal{C}_{\text{typed}} \rangle
\]

such that

61
(i) $\mathcal{G} = \langle \mathcal{U}, c^{\mathcal{G}}, \ldots \rangle$ is a type structure for $c_{\text{kind}}$

(ii) $\mathcal{G}$ is a family of sets $D_a$ indexed by elements of $T$

(iii) For each $a, b \in T$, we have a set $[D_a \to D_b]$ of functions from $D_a$ to $D_b$ with functions

$\Phi_{a,b} : D_{Fab} \to [D_a \to D_b]$ and $\Psi_{a,b} : [D_a \to D_b] \to D_{Fab}$

such that $\Phi_{a,b} \circ \Psi_{a,b}$ is the identity on $[D_a \to D_b]$.

(iv) For every $f \in [T \to T]$, we have a subset

$[\Pi_{a \in T} D_{f(a)}] \subseteq \Pi_{a \in T} D_{f(a)}$ with functions

$\Phi_{f} : D_{\Pi f} \to [\Pi_{a \in T} D_{f(a)}]$ and $\Psi_{a,b} : [\Pi_{a \in T} D_{f(a)}] \to D_{\Pi f}$

such that $\Phi_{f} \circ \Psi_{f}$ is the identity on $[\Pi_{a \in T} D_{f(a)}]$.

(v) For every $c \in c_{\text{typed}}$ there is a specified element $c^{\mathcal{G}}$
of the appropriate type.

An $\mathcal{G} \Lambda$-environment model is an $\mathcal{G} \Lambda$-frame which is sufficiently rich to allow all terms of $\mathcal{G} \Lambda$ to be interpreted. More precisely, an $\mathcal{G} \Lambda$-frame $\mathcal{G}$ satisfies (env) if for all environments $\eta$ satisfying $A$ (as defined below), the meaning function $\llbracket \cdot \rrbracket_\eta$ specified below is a total function from $\mathcal{G} \Lambda A$ to elements of the domain of $\mathcal{G}$. An $\mathcal{G} \Lambda$-environment model is an $\mathcal{G} \Lambda$-frame satisfying (env).

3.4.3 Meanings of Terms

Let $A$ be a type assignment. Let $\eta$ an environment mapping $\mathcal{U}_{\text{kind}}$ to elements of the appropriate kinds, and $\mathcal{U}$ to elements of $\cup \mathcal{G}$. We say that $\eta$ satisfies $A$, written $\eta \models A$, if

$\eta(x) \in \llbracket A(x) \rrbracket_\eta$

for each variable $x \in \text{dom}(A) \cap \mathcal{U}$.

The following lemma gives some straightforward, useful facts.

Lemma 3-2: Suppose $\eta \models A$. Then

(i) If $d \in \llbracket \sigma \rrbracket_\eta$, then $\eta[d/x] \models A[\sigma/x]$
(ii) If \( t \) is bindable in \( M \), then for any \( a \in T \), we have 
\[
\eta[a/t] \models A|_{FV(M)}
\]

If \( \eta \models A \), then the meanings of expressions of \( \lambda \Lambda \) are defined inductively as follows.

\[
\llbracket x \rrbracket \eta = \eta(x)
\]

\[
\llbracket c \rrbracket \eta = c^\eta
\]

\[
\llbracket MN \rrbracket \eta = (\Phi_{a,b} \llbracket M \rrbracket \eta) \llbracket N \rrbracket \eta.
\]

where \( a \rightarrow b \) is the meaning of \( Type_A(M) \) in \( \eta \),

\[
\llbracket \lambda x \in \sigma.M \rrbracket \eta = \Psi_{a,b} g, \text{ where}
\]

\[
g(d) = \llbracket M \rrbracket \eta[d/x] \text{ for all } d \in D_a \text{ and}
\]

\( a, b \) are the meanings of \( \sigma \) and \( Type_A(M) \) in \( \eta \)

\[
\llbracket M \tau \rrbracket \eta = (\Phi_{\tau} \llbracket M \rrbracket \eta) \llbracket \tau \rrbracket \eta,
\]

where \( \Pi \tau \) is the meaning of \( Type_A(M) \) in \( \eta \).

\[
\llbracket \Pi t.M \rrbracket \eta = \Psi_{\tau} g, \text{ where}
\]

\[
g(a) = \llbracket M \rrbracket \eta[a/t] \text{ for all } a \in T \text{ and}
\]

\( f \in [T \rightarrow T] \) is the function \( \llbracket \lambda t. Type_A(M) \rrbracket \eta \)

It is easy to check that the meanings of typed terms have the appropriate semantic types.

**Lemma 3.3:** Let \( \eta \) be an environment for a model \( \langle \mathcal{F}, \mathcal{A}, \ldots \rangle \). If \( M \in \lambda \Lambda \) and \( \eta \models A \), then \( \llbracket M \rrbracket \eta \in D_{Type_A(M)} \llbracket \eta \rrbracket \).

A very useful fact is the following Substitution Lemma.

**Lemma 3.4:** (Substitution) (i) Let \( M \) and \( N \) be terms of \( \lambda \Lambda \) and \( x \) an ordinary variable. Suppose \( \eta \models A \) and \( Type_A(N) = A(x) \). Then
\[
\llbracket N/x \rrbracket M \eta = \llbracket M \rrbracket \eta[[N] \eta/x].
\]

(ii) Furthermore, if \( t \) is bindable in \( M \) and then
\[
\llbracket \sigma / t | M \rrbracket \eta = \llbracket M \rrbracket \eta[[\sigma] \eta/t].
\]

(iii) If \( \mu, \nu^k \) are constructionals, and \( \nu^k \in \mathcal{K}_{\text{kind}} \) then
\[
\llbracket \nu / \nu | \mu \rrbracket \eta = \llbracket \mu \rrbracket \eta[[\nu] \eta/\nu].
\]
Parts (i) and (ii) of the lemma are easily proved by induction on terms. Part (iii) is a well-known property of simply-typed lambda calculus. A special case of the Substitution Lemma is that the meaning \([M]_\eta\) of a term \(M\) in environment \(\eta\) does not depend on \(\eta(x)\) or \(\eta(t)\) for \(x\) or \(t\) not free in \(M\).

### 3.5 Completeness

It is easy to verify that the axioms and inference rules are sound.

**Lemma 3.5:** (Soundness) Let \(\Gamma\) be a set of \(\forall\lambda \in \Lambda_A\) equations and let \(M, N \in \forall\lambda \in \Lambda_A\). If \(\Gamma \vdash_{\Lambda} M = N\), then \(\Gamma \models M = N\).

**Proof:** The proof is entirely straightforward. We will check two axioms, \((\alpha^\lambda_\Sigma)\) and \((\beta^\lambda_\Sigma)\), leaving the remaining axioms and inference rules to the reader. For any term \(\lambda x \in \sigma.M\) and variable \(y\) not free in \(M\),

\[
\begin{align*}
\llbracket \lambda x \in \sigma.M \rrbracket_\eta \\
= \Phi(\lambda d \in D_{[\sigma]} \llbracket M \rrbracket_\eta[d/x]) \\
= \Phi(\lambda d \in D_{[\sigma]} \llbracket y/x M \rrbracket_\eta[d/y]) \\
= \llbracket \lambda y \in \sigma. [y/x M] \rrbracket_\eta
\end{align*}
\]

The second equality follows from Lemma 3.4. The soundness of \((\alpha_\Pi)\) is proved similarly using Lemma 3.4.

For \((\beta^\lambda_\Sigma)\), consider any redex \((\lambda x \in \sigma M)N\). We have

\[
\begin{align*}
\llbracket (\lambda x \in \sigma M)N \rrbracket_\eta \\
= \Psi(\Phi(\lambda d \in D_{[\sigma]} \llbracket M \rrbracket_\eta[d/x])) \llbracket N \rrbracket_\eta \\
= \llbracket M \rrbracket_\eta \llbracket N \rrbracket_\eta/x \\
= \llbracket [N/x M] \rrbracket_\eta
\end{align*}
\]

by Lemma 3.4. The soundness of \((\beta_\Pi)\) and \((\beta_\Sigma)\) are proved similarly. It is clear from the inductive definition of the meanings of terms that semantic equality is a congruence relation. This proves the Lemma.

We will now show

**Theorem 1:** (Completeness) Let \(\Gamma\) be a set of \(\forall\lambda \in \Lambda_A\) equations and let \(M, N \in \forall\lambda \in \Lambda_A\). If \(\Gamma \vdash_{\Lambda} M = N\), then \(\Gamma \models M = N\).
The rest of this section is devoted to proving the completeness theorem. The proof uses a term model construction as in [Bruce and Meyer 84]. Given a theory $Th$ for $\mathcal{E} \Lambda_A$, we construct a model $\langle \mathcal{G}, \mathcal{A}, ... \rangle$ for $Th$ and environment $\eta_0$ satisfying $A$ with the property that $\llbracket M \rrbracket \eta_0 = \llbracket N \rrbracket \eta_0$ iff $M = N \in Th$. Recall that the language $\mathcal{E} \Lambda$ depends implicitly on the sets of constants $c_{\text{kind}}$ and $c_{\text{typed}}$.

We begin by defining a type structure $\mathcal{G} = \langle \mathcal{U}, c_{\mathcal{G}}, ... \rangle$ for $c_{\text{kind}}$. Let $\mathcal{U}$ be the "term model" for $c_{\text{kind}}$ built from equivalence classes of constructionals as in [Friedman 75]. Thus $T$ is the set of equivalence classes of type expressions. We use $\langle \mu \rangle$ to denote the equivalence class of the constructional $\mu$. As usual, the interpretation of a constant $c \in c_{\text{kind}}$ is its equivalence class $\langle c \rangle$.

An inductive argument as in [Friedman 75] shows that $\mathcal{G}$ is a type structure for $c_{\text{kind}}$.

We will define $\mathcal{G}$ using equivalence classes of terms. For any $M \in \mathcal{E} \Lambda_A$, let $\langle M \rangle$ denote the equivalence class

$$\langle M \rangle = \{ N \mid M = N \in Th \}.$$  

For each $\sigma \in T$, let

$$D_\sigma = \{ \langle M \rangle \mid Type_A(M) = \sigma \}.$$  

We now define $\{ \Phi, \Psi, c_{\mathcal{G}} \}_{c \in c_{\text{typed}}}$.

For each $\sigma, \tau \in T$, define $\Phi_{\sigma, \tau}$ and $\Psi_{\sigma, \tau}$ by

$$\Phi_{\sigma} \langle M \rangle \langle N \rangle = \langle MN \rangle,$$

$$\Psi_{\sigma}(\Phi_{\sigma} \langle M \rangle) = \langle \lambda x \in \sigma. M x \rangle,$$

where $x$ is chosen so as not to appear free in $M$. Let $[D_\sigma \rightarrow D_\tau]$ be the range of $\Phi$. We have $\Phi \circ \Psi = id_{[D_\sigma \rightarrow D_\tau]}$ since, for any $\Phi \langle M \rangle \in [D_\sigma \rightarrow D_\tau]$,

$$\Phi(\Psi(\Phi \langle M \rangle)) \langle N \rangle = \Phi \langle \lambda x \in \sigma. M \rangle \langle N \rangle = \langle MN \rangle = (\Phi \langle M \rangle) \langle N \rangle$$

for all $\langle N \rangle \in D_\sigma$.

For any $f = \lambda t. \sigma(t) \in [T \rightarrow T]$, we must define functions $\Phi_f$ and $\Psi_f$. For any $M$, $N$ with $Type_A(M) = \forall t. \sigma(t) = f$ and $Type_A(N) = \sigma(\tau) = f \tau$, let

$$\Phi_f \langle M \rangle \tau = \langle M \tau \rangle.$$

Let $M(t) \in \mathcal{E} \Lambda_A$ be any term with bindable $t$ such that $Type_A(M(t)) = f(t) = \sigma(t)$. Define $\Psi_f$ by

$$\Psi_f(\lambda t. \langle M(t) \rangle) = \langle \Pi s. M(s) \rangle.$$
We take \([\prod_{a \in T} D_{f(a)}] \subseteq \prod_{a \in T} D_{f(a)}\) to be the range of \(\Phi_f\) and note that \(\Phi_f \circ \Psi_f\) is the identity on \([\prod_{a \in T} D_{f(a)}]\) by \(\beta_{\Pi}\). Constants \(c \in \mathbb{C}_{\text{typed}}\) are interpreted as their equivalence classes. The equations verified above show that \(\mathfrak{F} = \langle \mathfrak{G}, \mathfrak{G}, ... \rangle\) is a frame for \(\mathcal{F}\Lambda\).

It remains to show that the frame is a model satisfying \(Th\). Let \(\eta\) be any environment for the frame \(\mathfrak{F} = \langle \mathfrak{G}, \mathfrak{G}, ... \rangle\) with \(\eta \models A\). For any term \(M \in \mathcal{F}\Lambda\) whose free variables are among \(\bar{x}, \bar{v}\), and free kinded variables are among \(\bar{x}', \bar{v}'\), let \(\{\eta\}M\) denote any term

\[
\{\eta\}M = [N/\bar{x}'][\bar{v}'/\bar{v}']M,
\]

where \(N\) and \(\bar{v}'\) are any sequences of terms and constructionals chosen so that \(N_i \in \eta(x_i)\) and \(v_i \in \eta(y_i)\) since \(=\) is a congruence relation, the equivalence class \(\langle \{\eta\}M \rangle\) is independent of the choice of \(N\) and \(\bar{v}'\). Furthermore, if \(\langle M \rangle = \langle N \rangle\), then \(\langle \{\eta\}M \rangle = \langle \{\eta\}N \rangle\). We define \(\langle \{\eta\} \mu \rangle\) similarly for any constructional \(\mu\). We will show that for any constructional \(\mu\), term \(M \in \mathcal{F}\Lambda\) and any environment \(\eta\) satisfying \(A\) we have

\[
[\mu] \eta = \langle \{\eta\} \mu \rangle \text{ and } [M] \eta = \langle \{\eta\} M \rangle.
\]

This shows that \(\mathfrak{F} = \langle \mathfrak{G}, \mathfrak{G}, ... \rangle\) is a model of \(Th\), i.e. every term has a meaning in \(\mathfrak{G}\). It also follows that for \(\eta_0\) with \(\eta_0(x) = \langle x \rangle\) and \(\eta_0(y) = \langle y \rangle\),

\[
[M][\eta_0] = [N][\eta_0] \text{ iff } \langle M \rangle = \langle N \rangle \text{ iff } M = N \in Th.
\]

This will prove the completeness theorem.

We can show by induction on constructionals that \([\mu] \eta = \langle \{\eta\} \mu \rangle\) as in [Meyer 82], for example. In particular, we have \([\sigma] \eta = \langle \{\eta\} \sigma \rangle\). We now consider terms. For any variable \(x\), clearly

\[
[x] \eta = \eta(x) = \langle \{\eta\} x \rangle.
\]

The application case is also straightforward:

\[
[MN][\eta] = (\Phi_{\langle \{\eta\} M \rangle} \langle \{\eta\} N \rangle = \langle \{\eta\} M \rangle \langle \{\eta\} N \rangle = \langle \{\eta\} (MN) \rangle.
\]

For a term that is a \(\lambda\)-abstraction, we have

\[
[\lambda x \in \sigma M] \eta = \Psi_g,
\]

where \(g\) is the function \(\lambda d \in D_{\langle \eta \rangle \sigma} [M][\eta] [d / x]\). For any \(N \in D_{\langle \eta \rangle \sigma}\), we have
\[ g(\textbf{N}) = [M] \eta(\textbf{N}/x) \]
\[ = \{\eta(\textbf{N}/x)\} \]
\[ = \{(\eta \lambda x \in \sigma. M) N\} \]
\[ = (\Phi \{\eta \lambda x \in \sigma. M\}) N \]

Therefore \( g = (\Phi \{\eta \lambda x \in \sigma. M\}) \). It follows that
\[ \|\lambda x \in \sigma. M\| \eta \]
\[ = \Psi(\Phi \{\eta \lambda x \in \sigma. M\}) \]
\[ = \lambda y \in \sigma. \{\eta \lambda x \in \sigma. M\} y \]
\[ = \{\eta \lambda y \in \sigma [y/x] M\} \]
\[ = \{\eta \lambda x \in \sigma. M\}. \]

A few more details may be found in [Meyer 82].

For polymorphic terms, we have
\[ \|\Pi_{\tau}. M\| \eta \]
\[ = \Psi(\lambda \tau \in T. [M] \eta(\tau/\iota)) \]
\[ = \Psi(\lambda \tau \in T. \{\eta(\tau/\iota)\} M) \]
\[ = \{\Pi \{\eta(\tau/\iota)\} M\} \]
\[ = \{\eta \Pi_{\tau}. M\}. \]

The remaining case is type application.
\[ \|M \tau\| \eta \]
\[ = \Phi (\tau) \eta \|M\| \eta \]
\[ = \Phi \{\eta\} \tau \{\eta\} M \]
\[ = \text{proj} \{\eta\} \tau \{\eta\} M \]
\[ = \{\eta\} (\text{proj} \tau M) \]

This concludes the proof of the theorem. \( \Box \)
Some interesting corollaries follow by choosing constants and equations between terms appropriately. Recall that $\forall \Lambda_{\rightarrow, \forall, \exists}$ is the language of existential types with $c_{\text{kind}} = \{ \text{F, } \Pi, \Sigma \}$, and $c_{\text{typed}} = \{ \text{inj}_{\lambda \sigma}, \text{sum}_{\sigma} \}$. The natural "pure" theory for this language is the theory $Th_3$ of $(\beta_\Sigma)$. From the Completeness Theorem, we have the immediate corollary

Corollary: Let $M, N$ be two terms of $\forall \Lambda_{\rightarrow, \forall, \exists}$ of the same type. Then $M = N \in Th_3$ iff $M = N$ holds in every model of $(\beta_\Sigma)$.

3.6 Higher-Order Type Theory and Combinatory Algebras

Much of the complication of the semantics of $\forall \Lambda$ lies in interpreting the binding operators $\lambda$ and $\Pi$ properly. In particular, the meanings of $\lambda x \in \sigma. M$ and $\Pi t. M$ must exist. Furthermore, $\lambda$ and $\Pi$ must be interpreted so that if $M = N$, then $\lambda x \in \sigma. M = \lambda x \in \sigma. N$ and $\Pi t. M = \Pi t. N$. The first requirement is reflected in condition (env) of the environment model definition. The second requirement, called weak extensionality, leads to the $\Psi$ functions of environment models. We develop an alternative to the environment model definition by beginning with a structure, the applicative frame, that only interprets application and projection. Since applicative frames are not designed to interpret $\lambda$ and $\Pi$, these structures are considerably simpler than environment models. We then use equational axioms to describe a special class of applicative frames, the $\forall \Lambda$-combinatory algebras, that satisfy a property similar to (env). Although combinatory algebras bring us a step closer to models, combinatory algebras do not satisfy weak extensionality. In Section 7, we will describe a special class of combinatory algebras, called $\forall \Lambda$-combinatory models, that may be used to interpret $\lambda$ and $\Pi$ properly. The definition of combinatory model is equivalent to the environment model definition is a precise sense.

Typed $\forall \Lambda$-combinatory algebras and $\forall \Lambda$-combinatory models are analogous to untyped combinatory algebras and combinatory models. We review the untyped definitions briefly using terminology similar to [Meyer 82]. An untyped applicative structure $\mathcal{A} = < D, \cdot >$ consists of a set $D$ with a binary operation $\cdot$. As in group theory, for example, it is customary to omit the operation $\cdot$ from expressions. An untyped applicative structure is combinatorially complete if, for every polynomial $p(x_1, \ldots, x_n)$ over $D$ with indeterminates among $x_1, \ldots, x_n$, there is a constant $d \in D$ such that
\[ \mathcal{G} \models p(x_1, \ldots, x_n) = dx_1 \ldots x_n. \]

By convention, the applications \( dx_1 \ldots x_n \) associate to the left. This condition is similar to \((\text{env})\) in that both require the existence of elements representing definable functions. It can be shown that an untyped applicative structure \( \mathcal{G} \) is combinatorially complete iff \( \mathcal{D} \) has elements \( \text{K} \) and \( \text{S} \) with the simple algebraic properties

\[
\text{K} \ x \ y = x \\
\text{S} \ x \ y \ z = (x \ z)(y \ z);
\]

see [Barendregt 81, Meyer 82]. Thus a combinatory algebra is an applicative frame satisfying two equational axioms.

An untyped combinatory algebra is not a model of the untyped lambda calculus since there is no mechanism for choosing \( \lambda x.M \) uniquely. A *combinatory model for untyped lambda calculus* is a structure \( \langle \mathcal{D}, \ast, \varepsilon \rangle \) with \( \langle \mathcal{D}, \ast \rangle \) a combinatory algebra and \( \varepsilon \in \mathcal{D} \) satisfying

\[
(\varepsilon.1) \ \forall \ d, e \ (ed)e = de, \\
(\varepsilon.2) \ \forall e (d_1 e = d_2 e) \ implies \ ed_1 = ed_2, \\
(\varepsilon.3) \ \varepsilon \varepsilon = \varepsilon.
\]

Untyped lambda abstraction can now be interpreted

\[
\llbracket \lambda x.M \rrbracket \eta = ed, \text{ where } \\
de = \llbracket M \rrbracket \eta[e/x] \text{ for all } e \in \mathcal{D}.
\]

Since \( \langle \mathcal{D}, \ast \rangle \) is a combinatory algebra, such an \( e \in \mathcal{D} \) will exist for any \( M \). Furthermore, weak extensionality follows from the properties of \( \varepsilon \). A comprehensive discussion of the equivalence between the environment and combinatory model definitions for untyped lambda calculus is given in [Meyer 82]. Note that since the combinatory algebra axioms are equational, combinatory algebras form an algebraic variety [Gratzer 68]. In contrast, the axioms for \( \varepsilon \) are not equational.

The astute reader will notice that the axioms for combinatory models may be written as first-order sentences. The advantage of the combinatory model definition is that it precisely describes models of untyped lambda calculus in a formal language with straightforward semantics. In the definition of \( \lambda \)-combinatory algebra and \( \lambda \Lambda \)-combinatory model, we will use
a logical language $\mathcal{L}$ to describe properties of applicative frames. We use equational axioms in $\mathcal{L}$ to define $\mathcal{L}$-combinatory algebras. However, as in untyped lambda calculus, additional non-universal axioms are required to satisfy the weak extensionality property of $\mathcal{L}_\Lambda$.

3.6.1 Higher-Order Type Theory

An applicative term is a term without any occurrences of $\lambda$ or $\Pi$. The formulas of $\mathcal{L}_\Lambda$ are defined by

$$G ::= M = N | G \land H | \neg G | \forall v^\kappa.G | \forall x \in \sigma.G,$$

where $M$ and $N$ are applicative terms of $\mathcal{L}_\Lambda$ with $Type_\Lambda(M) = Type_\Lambda(N)$, the formula $\forall v^\kappa.G$ is restricted to $v^\kappa$ not free in $A(x)$ for $x$ free in $G$, and for $\forall x \in \sigma.G$, we have $G \in \mathcal{L}_A[\sigma/x]$. Since only applicative terms appear in formulas of $\mathcal{L}$, we do not need the $\Psi$ functions of $\mathcal{L}_\Lambda$-frames to interpret formulas. Instead, we use the simpler applicative frames described below. Formulas of $\mathcal{L}$ are interpreted by giving the logical connectives $\land$ and $\neg$ their usual meaning, and by interpreting quantifiers as ranging over the appropriate sets of the frame.

3.6.2 Applicative Frames

An applicative frame for $\mathcal{L}_\Lambda(c_{\text{kind}}, c_{\text{typed}})$ is a tuple

$$\mathfrak{F} = \langle \mathfrak{G}, \mathfrak{H}, \{\ast, c^\mathfrak{F}, \ldots\} \rangle$$

with

(i) $\mathfrak{G} = \langle \mathfrak{G}_0, c^\mathfrak{G}, \ldots \rangle$ a type structure for $c_{\text{kind}}$

(ii) $\mathfrak{H}$ a family of sets $D_a$ indexed by elements of $T$

(iii) For each $a, b \in T$, we have a function

$$\ast_{a.b} : D_{a.b} \rightarrow D_a \rightarrow D_b.$$

(iv) For every $f \in [T \rightarrow T]$, we have a function

$$\ast_f : D_{\Pi f} \rightarrow \Pi_{a \in T} D_{fa}.$$

(v) For every $c \in c_{\text{typed}}$, there is a specified element $c^\mathfrak{F}$ of the appropriate type.

Both $\ast_{a,b}$ and $\ast_f$ will be written as infix operations. When the types are clear from context, we will omit the "$\ast$" and write, e.g., $de$ for $d_{a,b}e$. The meaning of an applicative term $M$ of $\mathcal{L}_\Lambda$ in environment $\eta = A$ is defined inductively by
\[ \parallel x \parallel \eta = \eta(x) . \]

\[ \parallel c \parallel \eta = c^\eta . \]

\[ \parallel M N \parallel \eta = \parallel M \parallel \eta \ast_{a,b} \parallel N \parallel \eta . \]

where \( a \rightarrow b \) is the meaning of \( \text{Type}_A(M) \) in \( \eta \).

\[ \parallel M \tau \parallel \eta = \parallel M \parallel \eta \ast_{f} \parallel \tau \parallel \eta . \]

where \( \Pi f \) is the meaning of \( \text{Type}_A(M) \) in \( \eta \).

Every \( \mathcal{A} \Lambda \) frame \( \langle \mathcal{F}, \mathfrak{A}, \{ \Phi, \Psi, c^\mathcal{F}, ... \} \rangle \) has an associated applicative frame \( \langle \mathcal{F}, \mathfrak{A}, \{ \ast, c^\mathcal{F}, ... \} \rangle \) defined by taking

\[ d \ast_{a,b} e = (\Phi_{a,b} d) e \]

\[ d \ast_{f} a = (\Phi_{f} d) a \]

and preserving the interpretations of constants. Consequently, we can interpret \( \mathcal{A} \Lambda \) formulas over \( \mathcal{A} \Lambda \) frames. It is easy to see that the meaning of an applicative term \( M \) in a \( \mathcal{A} \Lambda \) frame is the same as the meaning of \( M \) in the associated applicative frame. Essentially, the only difference is that \( \Phi \) and \( \ast \) have slightly different the semantic types.

If \( \mathcal{F} \) is an applicative frame, then the \( \mathcal{F} \)-terms are the applicative terms of the language \( \mathcal{A} \Lambda(\mathcal{F}) \) with a constant for each element of \( \mathcal{F} \).

### 3.6.3 Combinatory Completeness

Intuitively, an applicative frame is combinatorially complete if it is closed under definition by polynomials over ordinary variables and type variables. A frame \( \mathcal{F} = \langle \mathcal{F}, \mathfrak{A}, ... \rangle \) is \textit{combinatorially complete} if for every \( \mathcal{F} \)-term \( M \), sequence \( \tilde{s}^+ \) of type variables, and sequence \( \tilde{x}^+ \) of ordinary variables such that all free variables of \( M \) are among \( \tilde{s}^+ \) and \( \tilde{x}^+ \), there is a closed \( \mathcal{F} \)-term \( N \) such that

\[ \mathcal{F} \vdash M = N \tilde{s}^+ \tilde{x}^+. \]

This definition is similar to the usual definition of combinatorial completeness for untyped lambda calculus [Barendregt 81, Meyer 82], but with the added consideration of type variables. We do consider variables of higher kinds since terms of \( \mathcal{A} \Lambda \) only include \( \lambda \)-binding of ordinary
variables and \(\Pi\)-binding of type variables, and cannot be abstracted with respect to free variables of higher kinds.

We will see that an applicative frame is combinatorially complete iff it has elements \(I, K, S, A, B, C, D\) satisfying certain equational axioms. These elements are called combinators, following common parlance. The combinators \(I, K\) and \(S\) are similar to the combinators of the same names used in untyped lambda calculus. An applicative frame \(\langle g, g, \ldots \rangle\) has combinators if it contains elements

\[ I \in \forall t.t \rightarrow t \]

\[ K \in \forall s.t.s \rightarrow t \rightarrow s \]

\[ S \in \forall r.s.t.((r \rightarrow s \rightarrow t) \rightarrow (r \rightarrow s) \rightarrow r \rightarrow s) \]

and, for each \(f \in [T^{i+1} \rightarrow T]\), \(g \in [T^{j+1} \rightarrow T]\), and \(h \in [T^{k+2} \rightarrow T]\), with \((i, j, k \geq 0)\), elements

\[ A_f \in \forall r, s^{\downarrow} .((r \rightarrow \forall t.f(s^{\uparrow}, t)) \rightarrow \forall t.(r \rightarrow f(s^{\uparrow}, t)) \]

\[ B \in \forall f[r \rightarrow \forall t.r] \]

\[ C_{f, g} \in \forall f^{\uparrow}, s^{\uparrow} .((\forall t.f(f^{\uparrow}, t) \rightarrow g(s^{\uparrow}, t)) \rightarrow \forall t.f(f^{\uparrow}, t) \rightarrow \forall t.g(s^{\uparrow}, t)) \]

\[ D_{h, f} \in \forall f^{\uparrow}, s^{\uparrow} .((\forall t, u(h(f^{\uparrow}, t, u)) \rightarrow \forall t.h(f^{\uparrow}, t, f(s^{\uparrow}, t))) \]

satisfying the \(\Rightarrow\) axioms below. Note that each combinator has a closed type.

The combinators \(I, S\) and \(K\) must satisfy the typed universal closures of the following equations.

\[ I \, t \, x = x \]

\[ K \, s \, t \, x \, y = x \]

\[ S \, r \, s \, t \, x \, y \, z = x \, z \, (y \, z) \]

The types of variables \(x, y, z\) can easily be determined from the types given for the combinators. For example, the typed universal closure of the axiom for \(I\) is

\[ \forall t \in T . \forall x \in L . I \, t \, x = x \]

Each \(A, B, C, D\) must satisfy the typed universal closure of the appropriate equational axiom below.
\((A_t \, \bar{s}^+)_x \, y = (x \, y)_t\)

\((B_t)_x \, t = x\)

\((C_{h,g} \, \bar{r}^+ \, \bar{s}^+) \, x \, y \, t = x \, t \, (y \, t)\)

\((D_{h,f} \, \bar{r}^+ \, \bar{s}^+) \, x = x \, t \, f(\bar{s}^+, t)\)

An \(\mathcal{J}\Lambda\) frame has combiners if the associated applicative frame has combiners. It is worth noting that this set of combiners has been chosen for ease in proofs and is not intended to be a minimal set of combiners.

An \(\mathcal{J}\)-combinatory algebra is an applicative frame that has combiners. A combinatory algebra falls short of being a \(\mathcal{J}\Lambda\)-model in that it may lack the elements \(e_{a,b}\) and \(e_f\) discussed in the next section. We justify the name "combinatory algebra" by showing that every combinatory algebra is combinatorially complete. Let \(\mathcal{F}\) be any \(\mathcal{J}\)-combinatory algebra. Recall that the language \(\mathcal{J}\Lambda(\mathcal{F})\) has constants for each element of \(\mathcal{F}\). In particular, \(\mathcal{J}\Lambda(\mathcal{F})\) has a constant for each combinator. To show how combiners provide explicit definition of polynomials, we define "pseudo-abstraction" for ordinary variables and type variables. For any applicative term \(M\) of \(\mathcal{J}\Lambda_{A(\mathcal{F})}\), we define the applicative term \(\langle x \in \sigma \rangle M\) of \(\mathcal{J}\Lambda_{A(\mathcal{F})}\) as follows.

\[\langle x \in \sigma \rangle x = I_\sigma\]

\[\langle x \in \sigma \rangle y = K_\tau \sigma y, \text{ where } \tau = Type_A(y)\]

\[\langle x \in \sigma \rangle c = K_\tau \sigma c, \text{ where } \tau \text{ is the type of constant } c\]

\[\langle x \in \sigma \rangle (MN) = S_\tau \sigma \rho (\langle x \in \sigma \rangle M) (\langle x \in \sigma \rangle N)\]

where \(\tau = Type_A(M)\) and \(\rho = Type_A(N)\).

\[\langle x \in \sigma \rangle (M\rho) = (A_\tau \sigma \bar{s}^+) (\langle x \in \sigma \rangle M) \rho\]

where \(\forall \tau \tau = Type_A(M)\), all variables free in \(\forall \tau \tau\) are among \(\bar{s}^+\), and \(f = \lambda \bar{s}^+, \forall \tau \tau\).

If \(t\) is bindable in the applicative term \(M\) of \(\mathcal{J}\Lambda(\mathcal{F})_A\), we define \(\diamond M\) by
\[ \langle \text{D} y = B \tau y, \text{ where } \tau = Type_A(y) \rangle, \]
\[ \langle \text{D} c = B \tau c, \text{ where } \tau \text{ is the type of constant } c \rangle \]
\[ \langle \text{D} (MN) = C_{f,g} \bar{s} \bar{t} \langle \text{D} M \rangle \langle \text{D} N \rangle \]
\[ \text{where } f = \lambda \bar{r} \bar{t}, t. \sigma \text{ and } g = \lambda \bar{s} \bar{t}, t. \tau \text{ are closed and} \]
\[ \sigma \rightarrow \tau = Type_A(M) \text{ and } \sigma = Type_A(N), \]
\[ \langle \text{D} (M\tau) = D_{h,f} \bar{s} \bar{t} \langle \text{D} M \rangle \]
\[ \text{where } h = \lambda \bar{r} \bar{t}, t. u. \sigma \text{ and } f = \lambda \bar{s} \bar{t}, t. \tau \text{ are closed} \]
\[ \text{constructionals and } \forall u. \sigma = Type_A(M). \]

Note that \( \langle x \in \sigma \rangle M \) and \( \langle \text{D} M \rangle \) are applicative terms. The reader may find it instructive to work through a simple example like \( \langle \text{D} \langle x \in \sigma \rangle s \rightarrow t \rangle x \). The essential properties of \( \langle x \in \sigma \rangle M \) and \( \langle \text{D} M \rangle \) are described by the following lemma.

**Lemma 3-6:** Let \( \mathcal{F} \) be an \( \mathcal{H} \)-combinatory algebra. For any applicative terms \( M, N \) of \( \mathcal{H} A(\mathcal{F}) \) and type expression \( \sigma \) with \( Type_A(N) = \sigma \), we have

\[
\langle x \in \sigma \rangle M (N) = [N/x]M
\]
\[
\langle \text{D} M \rangle \sigma = [\sigma/x]M.
\]

The Lemma is easily proved by induction on terms. Using Lemma 3-6 we can prove the following Combinatory Completeness Theorem.

**Theorem 2:** (Combinatory Completeness) An applicative frame \( \mathcal{F} \) is combinatorially complete iff \( \mathcal{F} \) is an \( \mathcal{H} \)-combinatory algebra.

### 3.7 Combinatory Models

Just as untyped combinatory algebras are not suitable models of untyped lambda calculus, \( \mathcal{H} \)-combinatory algebras are not suitable for interpreting terms of \( \mathcal{H} A \). In order to satisfy the weak extensionality property of \( \mathcal{H} A \), we need a family of "choice elements" \( \varepsilon \) analogous to the single untyped \( \varepsilon \) of [Meyer 82]. The typed \( \varepsilon \)'s are similar to the \( \kappa \) and \( \theta \) of [Leivant 83b], but chosen to be uniformly defined as functions of their types.

#### 3.7.1 Combinatory Model Definition

An \( \mathcal{H} A \)-combinatory model is an \( \mathcal{H} \)-combinatory algebra such that
(vi) There is a distinguished element \( \varepsilon_0 \in \forall s \forall t ((s \rightarrow t) \rightarrow (s \rightarrow t)) \) such that for all \( s, t \in T \), all \( x, y \in s \rightarrow t \) and all \( z \in s \)

\[
(e_0 \cdot s \cdot t) \cdot z = xz \\
\forall z \in s (xz = yz) \supset (e_0 \cdot s \cdot t) x = (e_0 \cdot s \cdot t) y.
\]

\[
\varepsilon_0 = e_f \langle \sigma \rangle. e_g s \langle \sigma \rangle. \varepsilon_0 \cdot s \rightarrow t \cdot s \rightarrow t \cdot \langle x \in s \rightarrow t \rangle. e_0 \cdot s \cdot t \cdot \langle y \in s \rangle. xy
\]

where \( f = \lambda s. \forall t. s \rightarrow t \), \( g = \lambda s. \lambda t. s \rightarrow t \) and \( e_f, e_g \) as in (vii) below.

(vii) For every \( f \in [T^i \rightarrow T] \), \( i > 0 \), there is a distinguished element \( e_f \in \forall s^+ [\forall t f(s^+, t) \rightarrow \forall t f(s^+, t)] \) such that for all \( s^+ = s_1 \ldots s_{i-1}, t \in T \), and all \( x, y \in \forall t f(s^+, t) \)

\[
(e_f \cdot s^+) \cdot x \cdot t = xt, \\
\forall t (xt = yt) \supset (e_f \cdot s^+) x = (e_f \cdot s^+) y.
\]

\[
e_f = e_f \langle r \rangle \langle s_1 \rangle \ldots e_f \langle s_i \rangle \langle r \rangle, x r
\]

where \( r = \forall t f(s^+, t) \) and \( f_k = \lambda s_1 \ldots s_k \forall s_{k+1} \ldots s_i. f(s^+, t) \) for \( 1 \leq k \leq i \).

Furthermore, for all \( f \in [T^i \rightarrow T] \) and \( g \in [T^j \rightarrow T] \) with \( i, j > 0 \), we require

\[
\forall f^+, s^+, t[f(f^+, t) = g(s^+, t)] \supset e_f f^+ = e_g s^+.
\]

Note that the \( \varepsilon \) axioms, as well as the axioms for combinators, are formulas of \( \lambda \epsilon \theta \). The complicated third axiom of each \( \varepsilon \) definition is a normalizing condition. In effect, these equations state that

\[
\varepsilon_0 = \Pi s \Pi t \lambda x \in s \rightarrow t \lambda y \in s. xy \\
\varepsilon_f = \Pi s^+ \lambda x \in \forall t f(s^+, t) \Pi t. x t
\]

These conditions, analogous to the condition \( \varepsilon = \varepsilon \varepsilon \) for untyped lambda calculus [Meyer 82], make the mappings defined in Section 7.2 from environment models to combinatory models and from combinatory models to environment models inverses of each other. The final axiom relating \( e_f \) and \( e_g \) seems necessary in order to make the meaning of \( \Pi t. M \) terms uniquely defined. This "coordinated choice axiom" is used in the final part of the proof of Theorem 3-6. The axiom is correctly typed since

\[
\forall f^+, s^+, t[f(f^+, t) = g(s^+, t)]
\]

implies that for all \( f^+ \) and \( s^+ \), \( \lambda t. f(f^+, t) = \lambda t. g(s^+, t) \). Therefore,
\[ Type(\varepsilon_f \bar{r}^*) = \forall t. f(\bar{r}^*, t) = \Pi \{ \lambda t. f(\bar{r}^*, t) \} = Type(\varepsilon_g \bar{s}^*). \]

Since \( \varepsilon_f \bar{r}^* \) and \( \varepsilon_g \bar{s}^* \) are two distinguished choice elements of the same type, we want them to be the same.

If \( \eta \models A \), then the meanings of terms of \( \mathcal{E} \Lambda_A \) in a combinatory model \( \langle \mathcal{G}, \mathcal{F}, \ldots \rangle \) are defined inductively using the clauses for applicative terms (given in Section 6.2) and the following clauses for \( \lambda \) and \( \Pi \) terms.

\[ \llbracket \lambda x \in \sigma. M \rrbracket \eta = (\varepsilon_0 a b) \cdot d, \text{ where } d \text{ satisfies} \]

\[ d \cdot e = \llbracket M \rrbracket \eta[e/x] \text{ for all } e \in \mathcal{D}_a \text{ and} \]

\[ a, b \text{ are the meanings of } \sigma \text{ and } Type_A(M) \text{ in } \eta \]

\[ \llbracket \Pi. M \rrbracket \eta = (\varepsilon_f \bar{s}^*) \cdot d, \text{ where } d \text{ satisfies} \]

\[ d \cdot a = \llbracket M \rrbracket \eta[a/t] \text{ for all } a \in T \text{ and} \]

\( f \in [T \rightarrow T] \) is the meaning of the closed constructional \( \lambda \bar{s}^*. \lambda t. Type_A(M) \)

The meanings of terms of the form \( \lambda x \in \sigma. M \) and \( \Pi. M \) are defined by the above clauses only if certain elements of \( \mathcal{F} \) can be found. We can use Lemma 3-6 to show

**Theorem 3:** If \( \mathcal{G} \) is a combinatory model for \( \mathcal{E} \Lambda \), \( M \) is a term of \( \mathcal{E} \Lambda_A \) and \( \eta \) respects \( A \), then \( \llbracket M \rrbracket \eta \) is a uniquely defined element of \( \mathcal{G} \).

Thus combinatory models are models.

**Proof:** Let \( \mathcal{G} \) be any combinatory model. We must show that for any term \( M \) of \( \mathcal{E} \Lambda_A \) and environment \( \eta \) for \( \mathcal{G} \) with \( \eta \models A \), the meaning \( \llbracket M \rrbracket \eta \) is well-defined. We argue by induction on \( M \).

Essentially, we will show that for every \( M \), there is an applicative \( \mathcal{G} \)-term \( N \) with \( \mathcal{G} \models M = N \). Since the meaning of any applicative term is well-defined in any applicative structure, the Theorem follows. There is a slight complication due to the fact that Lemma 3-6 applies only applicative terms without free variables of higher kinds. (Lemma 3-6 cannot be strengthened since the types of combinators cannot have free variables of any kind.) We first show that for every constructional \( \mu \) and set \( \mathcal{W} \subseteq \mathcal{W}_{\text{kind}} \) of constructional variables, there is a constructional \( \mu_{\mathcal{W}} \) of \( \mathcal{E} \Lambda_A(\mathcal{G}) \) with no free variables from \( \mathcal{W} \) and such that

\[ \llbracket \mu \rrbracket \eta_1 = \llbracket \mu_{\mathcal{W}} \rrbracket \eta_1 \]
whenever \( \eta \) and \( \eta_1 \) agree on all variables of \( \Psi \). The proof is by induction on constructionals. We simply replace any variable in \( \Psi \) by a constant for \( \eta_1(\nu^k) \). The inductive step is straightforward, with \( (\lambda \nu^k. \mu)_\Psi = \lambda \nu^k.(\mu_\Psi \cdot \{\nu^k\}) \).

For any term \( M \), let \( KV(M) \subseteq \mathcal{F}^\text{kind} \) be the set of all free constructional variables in \( M \) other than type variables. We now show by induction on terms that for any term \( M \), there is an applicative term \( M_\mathcal{F} \) with no free variables of higher kinds such that

\[
\llbracket M \rrbracket \eta_1 = \llbracket M_\mathcal{F} \rrbracket \eta_1
\]

whenever \( \eta \) and \( \eta_1 \) agree on \( KV(M) \). If \( M \) is an ordinary variable or typed constant, then we let \( M_\mathcal{F} \) be \( M \). For an application \( MN \), let \( (MN)_\mathcal{F} \) be \( M_\mathcal{F}N_\mathcal{F} \). The most complicated case is a lambda-abstraction \( \lambda x^\sigma.M \). Since \( (\lambda x^\sigma.M)_\mathcal{F} \) must be an applicative term, we will use pseudo-abstraction instead of \( \lambda \). Let \( \Psi = KV(\lambda x^\sigma.M) \) and \( \tau = Type_A(M) \). Let

\[
(\lambda x^\sigma.M)_\mathcal{F} = \{ \sigma_0^\Psi \tau_\Psi \} \langle x^\sigma_\Psi \rangle.M_\mathcal{F}.
\]

It follows from the inductive hypothesis that for any \( \eta_1 \) agreeing with \( \eta \) on \( \Psi \),

\[
\llbracket \langle x^\sigma_\Psi \rangle.M_\mathcal{F} \rrbracket \eta_1 \cdot e = \llbracket M \rrbracket \eta_1[e/x]
\]

for all \( e \in \llbracket \sigma \rrbracket \eta_1 \). Therefore, for all \( \eta_1 \) agreeing with \( \eta \) on \( \Psi \),

\[
\llbracket \lambda x^\sigma.M \rrbracket \eta_1 = \llbracket (\lambda x^\sigma.M)_\mathcal{F} \rrbracket \eta_1
\]

is defined. Type application \( M \tau \) is straightforward. The \( \Pi \)-abstraction case

\[
(\Pi t.M)_\mathcal{F} = \{ \sigma_1 \cdot s^3 \} \langle t \rangle.M_\mathcal{F}
\]

where \( f = \lambda s^3 \lambda t.Type_A(M) \) is closed, is similar to the lambda-abstraction case.

Unicity is straightforward in all cases, except \( \Pi t.M \). If \( f = \lambda s^3 \lambda t.Type_A(M) \) and \( g = \lambda f^3 \lambda t.Type_A(M) \) are both closed, then for any \( f^3, s^3 \in T \), we have

\[
\lambda t.f(s^3, t) = \lambda t.g(f^3, t).
\]

Therefore, by the coordinated choice axiom of the combinatory model definition,

\[
\varepsilon_f s^3 = \varepsilon_g f^3
\]

and so the meaning of \( \Pi t.M \) is uniquely defined. This proves the theorem. \( \blacksquare \)

### 3.7.2 Combinatory Model Theorem

We prove the equivalence of the two model definitions by associating a combinatory model with each environment model, and vice versa. If \( \mathcal{A} = \langle \mathcal{E}, \mathcal{F}, \ldots \rangle \) is an environment model, then the associated combinatory model \( \mathcal{A}_\text{comb} \) will have the same type structure \( \mathcal{E} \), family \( \mathcal{F} \) of sets,
and equational theory as \( \Lambda \). In fact, an environment \( \eta \) for \( \Lambda \) will also be an environment for \( \Lambda_{\text{comb}} \) and \( \Lambda[M] \eta = \Lambda_{\text{comb}}[M] \eta \) for every term \( M \). Similarly, if \( \Lambda \) is a combinatory model, then the associated environment model \( \Lambda_{\text{env}} \) will have the same interpretations of constructionals and terms. In addition, the mappings \( \Lambda \rightarrow \Lambda_{\text{comb}} \) and \( \Lambda \rightarrow \Lambda_{\text{env}} \) will be inverses of each other.

If \( \Lambda = \langle \mathcal{W}, \mathcal{B}, \{ \Phi, \Psi, c^\Lambda, ... \} \rangle \) is an environment model, we define an applicative frame with \( \varepsilon \)'s
\[
\Lambda_{\text{comb}} = \langle \mathcal{W}, \mathcal{B}, \{ \cdot, \varepsilon, c^\Lambda, ... \} \rangle
\]
interpreting the same constants as follows.

For \( a, b \in T \), define \( \cdot_{a,b} \) and \( \varepsilon_0 \) by
\[
d \cdot_{a,b} e = (\Phi_{a,b} d) e
\]
\[
\varepsilon_0 = \Pi s \Pi t \lambda x \varepsilon s \rightarrow t \lambda y \varepsilon s. xy
\]
For \( f \in [T \rightarrow T] \), define \( \cdot_f \) by
\[
d \cdot_f a = (\Phi_f d) a
\]
and for \( f \in [T^1 \rightarrow T] \) define \( \varepsilon_f \) by
\[
\varepsilon_f = \Pi s^+ \lambda x \varepsilon s^+ f(s^+, t) \Pi r. x r
\]
It is easy to verify that \( \cdot \) and \( \varepsilon \) have the required properties. Note that the above \( \varepsilon \) definitions use terms of \( \exists \Lambda(\Lambda) \) to name elements of \( \Lambda \).

Conversely, if \( \Lambda = \langle \mathcal{W}, \mathcal{B}, \{ \cdot, \varepsilon, c^\Lambda, ... \} \rangle \) is a combinatory model, we define an \( \exists \Lambda \)-frame
\[
\Lambda_{\text{env}} = \langle \mathcal{W}, \mathcal{B}, \{ \Phi, \Psi, c^\Lambda, ... \} \rangle
\]
interpreting the same constants as follows.
For each a, b ∈ T, let
\[ \{d \in D_a \rightarrow D_b \mid d \in D_{fab}\}, \]
where \( g_d(e) = d \cdot a \cdot b \) for all \( e \in D_a \)
\( \Phi_{a,b}d = g_d \in [D_a \rightarrow D_b] \) as above, and
\( \Psi_{a,b}d = (e_0 \cdot a \cdot b)d. \)

For each \( f \in [T \rightarrow T] \), let
\[ \{f \in D_{fa} \mid D = \{f \in D_{fa} \mid D \in D_{\Pi}\}\}, \]
where \( h_d(a) = d \cdot f \) for all \( a \in T \)
\( \Phi_fd = h_d \in [D_{fa} \rightarrow D_{fa}] \) as above, and
\( \Psi_fh_d = \epsilon_fd. \)

The properties of \( \ast \) and \( \varepsilon \) may be used to show that \( \Phi \) and \( \Psi \) have the required properties.

We show that \( \mathcal{A}_{\text{comb}} \) has combiners. Consider the following terms of \( \forall \varepsilon A \), written using variables \( f, g \) and \( h \) of higher kinds.

\[ I^\Lambda = \Pi t \lambda x \in t. x \]
\[ K^\Lambda = \Pi s, t \lambda x \in s, y \in t. x \]
\[ S^\Lambda = \Pi r, s, t \lambda x \in r \rightarrow s \rightarrow t, y \in r \rightarrow s, z \in r. x (y z) \]
\[ A_f^\Lambda = \Pi r, s^\ast. \lambda x \in r \rightarrow \forall t. f(s^\ast, t). \Pi t. \lambda y \in r. x y t, \]
\[ B^\Lambda = \Pi r. \lambda x \in r. \Pi t. x \]
\[ C_{f,g}^\Lambda = \Pi f^\ast, s^\ast. \lambda x \in \forall t.f(f(s^\ast, t)) \rightarrow g(s^\ast, t). \lambda y \in \forall t.f(f(s^\ast, t), t). \Pi t.(x t)(y t), \]
\[ D_{h,f}^\Lambda = \Pi f^\ast, s^\ast. \lambda x \in \forall t.u. h(f^\ast, t, u). \Pi t. x t f(s^\ast, t) \]

In every environment, the environment model \( \mathcal{A} \) must interpret each of the terms above. It follows that \( \mathcal{A}_{\text{comb}} \) is a combinatorial model.

To see that \( \mathcal{A}_{\text{env}} \) satisfies condition (env), note that by Theorem 3-6, every term has a meaning in combinatorial model \( \mathcal{A} \). A straightforward induction on terms shows that for any term \( M \in \forall \varepsilon A \) and environment \( \eta = A \), the meaning \( \mathcal{A}_{\text{env}}[M] \eta \) exists and is equal to \( \mathcal{A}[M] \eta \). Thus we have the following combinatory model theorem, analogous to the combinatory model theorem of [Meyer 82].

79
Theorem 4: (Combinatory Model Theorem) If $\mathcal{A}$ is an environment model, then $\mathcal{A}_{\text{comb}}$ is a combinatorial model with the the same $\mathcal{E}$ theory as $\mathcal{A}$ and

$$\mathcal{A}[M]\eta = \mathcal{A}_{\text{comb}}[M]\eta$$

for any term $M$ of $\mathcal{E} \Lambda$ and $\eta = A$. Furthermore, $(\mathcal{A}_{\text{comb}}\phi)_{\text{env}} = \mathcal{A}$. Conversely, if $\mathcal{A}$ is a combinatorial model, then $\mathcal{A}_{\text{env}}$ is an environment model with the same $\mathcal{E}$ theory and

$$\mathcal{A}[M]\eta = \mathcal{A}_{\text{env}}[M]\eta$$

for any term $M$ of $\mathcal{E} \Lambda$ and $\eta = A$. Furthermore, $\mathcal{A}$ and $(\mathcal{A}_{\text{env}})_\text{comb} = \mathcal{A}$.

The identity $(\mathcal{A}_{\text{comb}})_\text{env} = \mathcal{A}$ is straightforward, while $(\mathcal{A}_{\text{env}})_\text{comb} = \mathcal{A}$ requires the $\epsilon$ normalizing conditions discussed in Section 7.1.

3.8 Definability of Existential Types

In formal logic, it is common to adopt a small number of logical constants as basic, introducing the remaining constructs by definition. In intuitionistic second-order logic, for example, we may take $\rightarrow$ and $\forall$ as basic, since Prawitz [Prawitz 65] has shown that these are sufficient to define all others. As noted in the Introduction, there is a close connection between the types of second-order lambda calculus (or $\mathcal{E} \Lambda$) and second-order formulas. This suggests that we may be able to define some type connectives from others, a suggestion implicit in [Reynolds 83].

In this section, we consider syntactic and semantic definitions of product types $\Lambda$, sum types $\vee$ and existential types $\exists$ in the language $\mathcal{E} \Lambda(\rightarrow, \forall)$. The general language $\mathcal{E} \Lambda$ makes it easy to see that models of the second-order lambda calculus with $\Lambda$, $\vee$ or $\exists$ types can be constructed merely by interpreting the additional constants in models of $\mathcal{E} \Lambda(\rightarrow, \forall)$.

The logical constants $\Lambda$, $\vee$ and $\exists$ may be defined in intuitionistic logic by

$$\sigma \wedge \tau ::= \forall s. (\sigma \rightarrow \tau \rightarrow s) \rightarrow s,$$

$$\sigma \vee \tau ::= \forall s. ((\sigma \rightarrow s) \wedge (\tau \rightarrow s)) \rightarrow s,$$

$$\exists t. \sigma ::= \forall s. (\forall t. (\sigma \rightarrow s) \rightarrow s),$$

where $s$ is assumed not to occur free in $\sigma$ or $\tau$. The reader familiar with classical logic may understand the definitions as generalizations of classical definitions of these connectives. Note first that since $\forall t. t$ is an absurd proposition, $\sigma \rightarrow \forall t. t$ is a negation of $\sigma$. Now, by substituting
∀t.t for the bound s in each case, we see that each intuitionistic abbreviation implies a more
familiar classical abbreviation. For example, the ∀ abbreviation ∀s.((σ→s)∧(τ→s))→s implies
((σ→∀t.t)∧(τ→∀t.t))→∀t.t,

a round-about way of expressing ¬(¬σ∧¬τ). Similarly, ∀s.(∀t.(σ→s)→s) is a generalization
of the usual classical definition of ∃ using ∀ and ¬.\textsuperscript{12} We now concentrate on ∃; analogous
developments are easy to work out for the other type constructors.

Recall that the axiom for \textit{inj} and \textit{sum} is

\[(β_β) \ (\text{sum}_{σ(i)} ρ M)(\text{inj}_{∃l.σ(i)} τ N) = M \ τ N.\]

We can introduce constants \textit{sum} and \textit{inj} that satisfy this axiom by nonconstructive definition.
The constants may be defined by the following terms

\[\text{sum}_{σ(i)} ::= Πr. \ λx∈(∀t.(σ(t)→r)). \ λy∈∀s(∀t_1.(σ(t_1)→s)→s). \ y \ r \ x\]

\[\text{inj}_{∃l.σ(i)} ::= Πl. \ λx∈σ(i), \ λs. \ λy∈∀t_1.(σ(t_1)→s). \ y \ r \ x\]

It is easy to simplify

\[(\text{sum}_{σ(i)} ρ M)(\text{inj}_{∃l.σ(i)} τ N)\]

to

\[M \ τ N,\]

using \((β_λ)\) and \((β_Π)\), thus verifying that the axiom for injection and sum holds. As a
consequence, we have

\textbf{Lemma 3-7:} Let M and N be terms of \textit{∃} \textit{λ}(→, ∀, ∃) and let \textit{Trans}(M), \textit{Trans}(N) be
obtained from M, N by replacing each \textit{inj}_{∃l.σ(i)} and \textit{sum}_{σ(i)} by the terms given
above. If \(\vdash M = N\), then \(\vdash \textit{Trans}(M) = \textit{Trans}(N)\).

The Lemma is easily proved by induction on proofs. It seems likely that \(\vdash M = N\) iff
\(\textit{Trans}(M) = \textit{Trans}(N)\), but this remains a conjecture, at present.

Given a model \(\mathcal{A}\) for \textit{∃} \textit{λ}(→, ∀), we let \(\mathcal{A}∃\) denote the model for \textit{∃} \textit{λ}(→, ∀, ∃) with
constructional constant Σ interpreted so that

\[Σf = ∀s.(∀t.(f(t)→s)→s),\]

and constants \textit{sum} and \textit{inj} interpreted as the meanings of the terms above. Then we have the
following relationship between the meanings of terms and their translations.

\textsuperscript{12} As noted in [Prawitz 65], the definition for \(Λ\) is due to Russell [Russell 03].
Lemma 3-8: Let $M$ be a term of $\forall \exists \forall \exists$, let $\text{Trans}(M)$ be as in the preceding lemma, and let $\mathcal{A}$ be a model for $\forall \exists \forall \exists$. Then

$$\mathcal{A} \models [M] \eta = \mathcal{A} \models [\text{Trans}(M)] \eta$$

for any environment $\eta$ respecting $\text{Trans}(\mathcal{A})$.

3.9 Directions for Further Investigation

The main open problem in the semantics of second-order lambda calculus is to determine whether there exist natural mathematics models that are not constructed using universal domains. Recent work by Reynolds [Reynolds 84] suggests that certain unanticipated isomorphisms between types may be forced in all models. Reynolds shows that in every "set-theoretic" model, there is some type $S$ which is isomorphic to $(S \rightarrow B) \rightarrow B$ for some nontrivial $B$. This is a set-theoretic contradiction. However, it is not clear whether some form of Reynolds' argument may be applied to all models. If some form of Reynolds' argument applies generally, then elementary models may not exist.

An important problem related to the problem of constructing models is to characterize the isomorphisms between types (or retracts of types) that must hold in all models. In particular, is there a type $\tau$ such that $(\tau \rightarrow B) \rightarrow B$ must be isomorphic to $\tau$ in all models (where $B$ is some nontrivial type such as $\forall t.(t \rightarrow t) \rightarrow t \rightarrow t$)? Recently, Kim Bruce and G. Longo [Bruce and Longo 84] have characterized a class of "definable" isomorphisms that must hold in every model, but other kinds of isomorphism may also be important. Some other important directions for further research are listed below.

1. Investigate model constructions and relationships between models. It should be easy to show that the product of two models is a model. Can the theory of a product model be related easily to the theories of the constituent models? There are also a number of natural questions about "homomorphic images" and submodels. For example, can every model be embedded in an extensional model, or does every model have an extensional homomorphic image? In order to investigate these questions, the appropriate definition of "homomorphism" must be developed. It seems likely that an extension of logical relation will serve well (cf. [Statman 84]).

2. Is there a nontrivial finite model (i.e. a model in which every $D_a$ is finite)? Are there nontrivial models with finitely or infinitely many $D_a$ finite?
3. Investigate theories involving equations between constructionals. The language is defined to suggest this extension and the completeness proof does not seem to rely on the absence of axioms. The complications here seem largely syntactic. For example, the application symbol \text{app} must be typed in order to make the application \text{app} M N unambiguous. However, this type must be closed, or else binding of type variables in M and N will necessitate binding of type variables in \text{app}. This leads on to consider a single polymorphic \text{app} \in \forall s, t[(s 	o t) 	o s 	o t]. A similar problem arises with \text{proj}, but without as clear a resolution. The natural closed types of \text{proj} are of the form \forall s \forall t \forall r [\forall \sigma (t) \rightarrow \sigma (r)]. But given a \text{proj} of this type, it takes a \text{proj} of a higher type to reduce the type to \forall t [\forall \sigma (t) \rightarrow \sigma (r)]. There does not seem to be a good way to index \text{proj} functions by closed type expressions. Nonetheless, the complications seem largely syntactic and not insurmountable.

4. The language \text{\lambda} only allows binding of ordinary variables and type variables. Investigate the extension of \text{\lambda} which allows any term to be treated as a function of any variable \forall \kappa of any kind. What are the proof-theoretic and semantic properties of this language? What about the language with kind variables?

5. Do models of \text{\lambda} shed any light on models for Martin-Löf’s constructive theory [Martin-Löf 79]? Automath [Barendregt and Reus 84, DeBruijn 80] or the related calculus developed by Huet and Coquand [Coquand and Huet 84]?

6. Develop a deductive system for \text{\lambda}. A general investigation of \text{\lambda} may prove useful in further research into \text{\lambda} and similar systems. It may be relatively straightforward to prove a Henkin-style completeness theorem, but there are some subtle issues about restricting the class of models. If a deductive system is too strong (e.g., Cantor’s theorem can be proved), then it will rule out useful models. The discussion in Section 4 of [Scott 80] applies here.
Chapter Four

Type Inference with Simple Coercions

4.1 Introduction

Type inference is a form of type checking. In programming languages where all identifiers are given types as they are introduced, it is often a simple matter to check whether the types of operators, functions and procedures agree with the types of operands and actual parameters. For some programming applications, it is convenient to be able to omit type declarations from programs. This may make it easier to write or modify experimental programs quickly. More importantly, a single untyped program may represent many explicitly typed programs. This gives rise to an implicit form of polymorphism. When type information is omitted from programs, there is a syntactically-defined type inference problem. If $\mathcal{F}$ is a typed programming language, then type inference for $\mathcal{F}$ is the problem of taking any untyped program $S$ and finding all possible ways of inserting type declarations into $S$ that result in legal typed programs of $\mathcal{F}$. Type inference was first considered in [Curry and Feys 58, Hindley 69] and has been developed further in [Gordon, et. al. 79, Leivant 83a, Milner 78]. A general discussion of the use of type inference in programming languages may be found in [Milner 78].

Automatic coercions, such as coercions from integers to real numbers, are common to many programming languages. However, automatic coercions involve relationships between distinct types and it is not clear, a priori, how coercions effect type inference. One complicating feature of coercions is that a single coercion implies many others: if integers are coercible to reals, then real predicates (boolean-valued functions of real arguments) are coercible to integer predicates. This paper addresses the problem of inferring types in a way which supports automatic coercions. Some of the basic properties of coercions are related to type containment in more complicated type disciplines.

In many cases, with or without coercions, a single untyped function will have infinitely many legal typings. For example, the body $f(x)$ of the function $\text{Apply}(f,x)$ is legally typed if $f$ has
some functional type \( s \rightarrow t \) and \( x \) has type \( s \) (the same \( s \)). When \( f \) has type \( s \rightarrow t \) and \( x \) has type \( s \), the result of \( \text{Apply}(f, x) = f(x) \) has type \( t \). Thus \( \text{Apply} \) has type
\[
((s \rightarrow t) \times s) \rightarrow t
\]
for any \( s \) and \( t \). In particular, \( \text{Apply} \) has every type that is a substitution instance of the above expression, for example
\[
\text{Apply}: ((\text{int} \rightarrow \text{bool}) \times \text{int}) \rightarrow \text{bool}, \text{ and} \\
\text{Apply}: ((\text{real} \rightarrow \text{int}) \times \text{real}) \rightarrow \text{int}. 
\]
In many programming languages, such as Algol and Pascal, we would have to declare a different \( \text{Apply} \) function for each typing that we needed. Since the two functions \( \text{Apply}_1: ((\text{int} \rightarrow \text{bool}) \times \text{int}) \rightarrow \text{bool} \), and \( \text{Apply}_2: ((\text{real} \rightarrow \text{int}) \times \text{real}) \rightarrow \text{int} \) will be the same except for type declarations, it is much more convenient to be able to declare \( \text{Apply} \) only once. If we have a type inference algorithm which infers that \( \text{Apply} : ((s \rightarrow t) \times s) \rightarrow t \), then we should be able to allow a call \( \text{Apply}(g, y) \) whenever the type of the pair \( (g, y) \) is a substitution instance of \( (s \rightarrow t) \times s \). Thus type inference algorithms can be used to support an implicit form of polymorphism. Polymorphism with coercions is considered in Section 6.

The most popular and best known programming language based on type inference is ML [Cordon, et al. 79]. A major feature of ML is that it supports the kind of implicit polymorphism described above. In the specific type system chosen for ML, each expression, function and procedure has a most general type, also called a principal type scheme. This means that for each clause (i.e. expression, function or procedure) \( M \), there is a single type expression \( \sigma \) (generally with type variables) such that all legal typings of \( M \) are substitution instances of \( \sigma \). Most general type expressions have the convenient property that the most general type of a clause involving clauses \( M \) and \( N \) can be determined from the most general types of \( M \) and \( N \). The efficiency of the ML type checker seems to be a direct consequence of the fact that every ML expression has a most general type in every context. When coercions are added to a subset of ML, type schemes alone do not characterize all possible typings of each term. For example,

\[1\] The type operator \( \rightarrow \) means "function space" and \( \times \) means "product space." For example, \( \text{int} \rightarrow \text{bool} \) is the type of functions from integers to boolean and \( \text{int} \times \text{bool}^+ \) is the type of pairs \( \langle a, b \rangle \) where \( a \) is an integer and \( b \) is a boolean.
consider the context "s is coercible to t and y has type t→u." In this context, the expression
\[ \lambda x. \text{proj}_1 \langle x, yx \rangle \]
has types s→s, t→t, and no others. Since neither of these types is coercible to the other, neither can reasonably be regarded as the most general type of the expression. Nonetheless, there is a concise representation for the set of legal typings of any typable term. This representation uses a type scheme and a set of coercions between types.

The simple model of coercion adopted in this paper is based on set containment: if \( \sigma \) is coercible to \( \tau \), then we think of the semantic values associated with type \( \sigma \) as contained in the set of values associated with type \( \tau \). This approach to coercion encompasses common coercions such as treating integers as reals in arithmetic expressions. The set of integers may be viewed as a subset of the set of real numbers. While more general notions of coercion can be developed (e.g., [Reynolds 80]), the simple model used here is likely to be a special case of any more general model. The completeness theorem will carry over to any more general model of coercions, provided that the inference rules remain sound.

Many of the properties of coercion which are discussed in this paper will apply to other type systems that involve some implicit or explicit notion of type containment. In particular, type containment may come up in type systems which include type quantifiers or other binding operations. For example, the type

\[ \forall t. t \to t \]

with a universal quantifier is the type of functions which, for any type \( \sigma \), is a function from \( \sigma \) to \( \sigma \) (cf. [Fortune, et. al. 83, Mitchell 84b, Reynolds 74]). The type \( \forall t. t \to t \) is essentially "contained within" the type \( \forall t. (t \to t) \to (t \to t) \). This statement may be made precise in several ways (cf. [MacQueen and Sethi 82, Mitchell 84b]). Thus we expect that any untyped function which can be given the type \( \forall t. t \to t \) can also be given the type \( \forall t. (t \to t) \to (t \to t) \). In quantified type disciplines (with or without automatic coercion), containment of types is a crucial issue. Automatic coercion motivates a general study of type containment and this study provides insight into type inference for more complicated type systems.
4.2 Lambda Calculus and its Semantics

Lambda calculus, the basis of ML, is used to demonstrate type inference with coercions. The terms of untyped lambda calculus are defined by the grammar

\[ M ::= x | MN | \lambda x.M. \]

A lambda model \(<D, \cdot, \varepsilon>\) is a set \(D\) together with binary operation \(\cdot\), "choice element" \(\varepsilon\), and elements \(K, S \in D\) with certain algebraic properties. This is the combinatory model definition of [Meyer 82]; see also [Barendregt 81].

Given a lambda model \(<D, \cdot, \varepsilon>\) and environment \(\eta\) mapping variables to elements of \(D\), the meaning of a lambda term \(M\) is defined inductively by

\[ \llbracket x \rrbracket \eta = \eta(x) \]
\[ \llbracket MN \rrbracket \eta = \llbracket M \rrbracket \eta \cdot \llbracket N \rrbracket \eta \]
\[ \llbracket \lambda x.M \rrbracket \eta = \varepsilon \cdot d, \text{ where } d \cdot e = \llbracket M \rrbracket \eta[e/x] \]

The existence of \(K\) and \(S\) ensure that there always exists a \(d\) as required in the definition of \(\llbracket \lambda x.M \rrbracket\). The element \(\varepsilon\) makes the meaning of \(\lambda x.M\) independent of the specific choice of \(d\). Again, the reader is referred to [Barendregt 81, Meycr 82] for specifics.

A few facts about the reduction rules of lambda calculus are used. See [Barendregt 81] for a comprehensive presentation. We consider lambda terms modulo \(\alpha\)-conversion

\((\alpha)\)  \(\lambda x.M = \lambda y.[y/x]M\) if \(y\) is not free in \(M\)

so that we can rename bound variables freely. The reduction rules are

\((\beta)\)  \((\lambda x.M)N \rightarrow [N/x]M\),

\((\eta)\)  \(\lambda x.Mx \rightarrow M\) if \(x\) is not free in \(M\),

where substitution is defined with renaming of bound variables to avoid capture. If a term \(M\) is of the form of the left-hand side of rule \((\beta)\) or \((\eta)\), then \(M\) is a \(\beta\)- or \(\eta\)-redex. We say that \(M\) \(\beta\)-reduces to \(N\) in one step if there is a subterm \(P\) of \(M\) which is a \(\beta\)-redex and \(N\) is the result of contracting this redex in \(M\). The term \(M\) \(\beta\)-reduces to \(N\) if there is a sequence of \(\beta\)-reductions leading from \(M\) to \(N\). If we allow \(\eta\)-reduction in addition to \(\beta\)-reduction, then we say \(M\) \(\beta,\eta\)-reduces to \(N\). A term which cannot be reduced is in normal form. Conversion is the least congruence relation containing reducibility; \(\equiv\beta\) denotes \(\beta\)-conversion and \(\equiv\beta,\eta\) denotes
$\beta,\eta$-conversion.

One important model is the term model $\langle D, \cdot, \varepsilon \rangle$ with

$$D = \{ [M] \mid M \text{ an untyped term} \},$$

where $[M]$ denotes the $=_{\beta}$-equivalence class of $M$. Application, $\cdot$, in term models is defined by

$$[M][N] = [MN]$$

and choice element

$$\varepsilon = [\lambda x.y.x y].$$

See [Barendregt 81, Meyer 82] for properties of term models.

4.3 Type Expressions, Coercion Sets and Type Assignments

Although product types, lists, and other kinds of types are useful in programming languages, these types seem to interact with automatic coercion in a relatively straightforward way. The most interesting types for our purposes are the functional types. Intuitively, the functional type $\sigma \rightarrow \tau$ consists of the set of functions which take arguments of type $\sigma$ to results of type $\tau$.

We use $\sigma \subseteq \tau$ to denote the fact (or assumption) that values of type $\sigma$ can be coerced to values of type $\tau$. If we think of the coercion relation between types as an ordering, then $\rightarrow$ is monotonic in its second argument:

if $\sigma \subseteq \rho$ then $\tau \rightarrow \sigma \subseteq \tau \rightarrow \rho$.

However, $\rightarrow$ is antimonotonic in its first argument, i.e.

if $\sigma \subseteq \rho$ then $\rho \rightarrow \tau \subseteq \sigma \rightarrow \tau$

rather than the reverse inclusion. If every value of type $\sigma$ can be treated as a value of type $\rho$, then every function which maps $\rho$ to $\tau$ also maps $\sigma$ to $\tau$. While in some semantics, e.g. [Scott 76], the connective $\rightarrow$ is monotonic in both arguments, antimonotonicity seems natural when we think of containment as the ability to coerce. If $f$ is a function of one real argument, and integers are coercible to reals, then $f$ should be applicable to integer values. It seems inappropriate to apply integer functions to real numbers.

Type expressions are built from type variables and constants using the connective $\rightarrow$. We adopt the notational conventions that
\( r, s, t, \ldots \) denote type variables

\( \rho, \sigma, \tau, \ldots \) denote type expressions.

The set of type expressions is defined by the grammar

\[
\tau ::= t \mid \sigma \to \tau.
\]

While we have omitted type constants, their introduction poses only minor complications. The only difficulties are in the algorithms Match and Coerce discussed in Section 5.

Types will be interpreted as arbitrary sets of elements of lambda models (cf. [Barendregt, Coppo and Dezani 83, Hindley 83a]). A type environment \( \eta \) for a model \( \langle D, \cdot, \varepsilon \rangle \) is a mapping from type variables to subsets of \( D \). The meaning of a type expression \( \sigma \) in a type environment \( \eta \) is defined inductively by

\[
\llbracket t \rrbracket \eta = \eta(t)
\]

\[
\llbracket \sigma \to \tau \rrbracket \eta = \{ d \mid \forall e \in \llbracket \sigma \rrbracket \eta, d \cdot e \in \llbracket \tau \rrbracket \eta \}.
\]

This is the "simple semantics" for \( \to \). Note that membership in \( \sigma \to \tau \) determined only by the applicative behavior of an element \( d \). Other semantics for \( \sigma \to \tau \) are proposed in [Hindley 83a, Hindley 83b, MacQueen and Sethi 82, Scott 76].

A coercion set \( C \) is a set of coercions \( s \subseteq t \) between type variables (or type constants, if they are allowed). A model \( \langle D, \cdot, \varepsilon \rangle \) and type environment \( \eta \) satisfy a coercion set \( C \) if

\[
\llbracket s \rrbracket \eta \subseteq \llbracket t \rrbracket \eta \quad \text{for all} \ s \subseteq t \in C.
\]

Coercions like

\( (r \to s) \subseteq t \)

have complex implications. Some sets of coercions between structured types may only be satisfiable in special lambda models, and some coercions may drastically change the set of typable terms. For example, if we assume the two coercions

\( (t \to t) \subseteq t \) and \( t \subseteq (t \to t) \),

then every term of the untyped lambda calculus will semantically have type \( t \). In contrast, only a subset of the set of terms with normal forms will have types when no coercions are allowed. We make sure that our coercions are not "domain equations" by only considering logical consequences of coercions between atomic types. This also preserves a correspondence between the typable terms and typed lambda calculus. It is possible that the results of this paper can be
extended to more complicated containments.

A type assignment $A$ is a function from a set of variables that may appear in lambda terms to type expressions. Often, type assignments will be functions with finite domains. A type assignment $A$ can be written as a set of statements of the form $x : \sigma$. An environment $\eta$ mapping type variables to subsets of $D$ and ordinary variables to elements of $D$ satisfies a type assignment $A$ if

$$\eta(x) \in \llbracket A(x) \rrbracket_\eta$$

whenever $x \in \text{dom}(A)$.

If $x$ is a variable, $\sigma$ a type expression and $A$ a type assignment, then $A[\sigma/x]$ is a type assignment with $(A[\sigma/x])(y) = A(y)$ for any variable $y$ different from $x$, and $(A[\sigma/x])(x) = \sigma$.

A typing statement describes the type of an expression, given coercions between types and the types of variables. Informally, the statement

$$C, A \supset M : \sigma$$

means that if types may be coerced according to $C$ and free variables have the types assigned by $A$, then the term $M$ has type $\sigma$. More formally, a model and environment $\eta$ satisfy a typing $M : \sigma$ if

$$\llbracket M \rrbracket_\eta \in \llbracket \sigma \rrbracket_\eta.$$ 

A statement $C, A \supset M : \sigma$ is satisfied in a model $<D, \cdot, \epsilon>$ if every environment and type environment for $<D, \cdot, \epsilon>$ which satisfy $C$ and $A$ also satisfy $M : \sigma$. A statement is valid if it is satisfied by every model.

4.4 Rules for Type Inference

There are five simple, semantically complete rules for deducing type statements. The first three rules of the system express well-known facts about the functionality of lambda terms [Curry and Feys 58]. The next rule, (coerce), formalizes the property that if a term $M$ has type $\sigma$, and the type $\sigma$ is coercible to the type $\tau$, then $M$ also has type $\tau$. These four rules are called the Curry rules with coercions, or CC for short. They define a typed language similar to the usual simple typed lambda calculus. The type checking algorithm TYPE in Section 5 will be based on the Curry rules with coercions. As in [Gordon, et. al. 79, Milner 78], rules for other applicative programming language constructs may be added. In particular, the polymorphic let declaration of ML is considered in Section 6.
The Curry rules with coercions are not semantically complete. If we want semantic completeness, then untyped terms that have the same meaning must be given the same types. One natural way to achieve this is to add a fifth rule (equal) based on equality of untyped terms. The resulting system is called *Curry typing with containment and equality*, or $\text{CC}_{eq}$. Since the equations that are valid in all lambda models are recursively enumerable, the antecedents of this rule are recursively enumerable. However, it is not decidable whether the rule is applicable in any given instance. When we adopt rule (equal), the set of types associated with an expression is, in general, undecidable.

The rule (coerce) that models automatic coercions will use two subsidiary rules for deducing consequences of coercion sets. Although coercion sets only contain coercions between atomic types, many other coercions will follow as consequences. The axiom and rules for deriving coercions are

- (ref) $\sigma \subseteq \sigma$,
- (arrow) $\sigma_1 \subseteq \sigma, \tau \subseteq \tau_1 \vdash \sigma \rightarrow \tau \subseteq \sigma_1 \rightarrow \tau_1$

and

- (trans) $\sigma \subseteq \tau, \tau \subseteq \rho \vdash \sigma \subseteq \rho$.

The soundness of these rules follows easily from the definition of $[\sigma]$. A coercion $\sigma \subseteq \rho$ is *provable from C*, written

$C \vdash \sigma \subseteq \rho$,

if $\sigma \subseteq \rho$ can be derived from elements of C using rules (ref), (arrow) and (trans).

The following lemma about the structure of proofs of coercions will be used to prove several facts about derivations of typing statements.

**Lemma 4.1:** Let $\sigma$ and $\tau$ be type expressions with $\sigma = \sigma_1 \rightarrow \sigma_2$ and $\tau = \tau_1 \rightarrow \tau_2$. Then $C \vdash \sigma \subseteq \tau$ if and only if $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$.

**Proof:** One direction is a direct consequence of rule (arrow). If $C \vdash \tau_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \tau_2$, then $C \vdash \sigma \subseteq \tau$. It remains to prove the converse.

We show that if $C \vdash \sigma \subseteq \tau$ for any $\sigma$ and $\tau$ of the form $\sigma = \sigma_1 \rightarrow \sigma_2$ and $\tau = \tau_1 \rightarrow \tau_2$, then there is a proof of $\sigma \subseteq \tau$ from C that ends with an application of rule (arrow). We argue by induction on the length of the proof of $\sigma \subseteq \tau$ from C. If the proof is one step, then this step must be an application of rule (arrow) and so, trivially, there must be a proof ending in an application of (arrow). For the inductive
step, assume that we have a proof whose final step is a use of rule (trans) from antecedents \( \sigma \subseteq \rho \) and \( \rho \subseteq \tau \), where \( \rho = \rho_1 \rightarrow \rho_2 \). Since the proofs of \( \sigma \subseteq \rho \) and \( \rho \subseteq \tau \) are shorter, we may assume we have proofs of these inclusions ending in applications of rule (arrow). Thus

\[
C \vdash \rho_1 \subseteq \sigma_1, \quad \sigma_2 \subseteq \rho_2, \quad \tau_1 \subseteq \rho_1, \quad \rho_2 \subseteq \tau_2.
\]

By rule (trans), we have

\[
C \vdash \tau_1 \subseteq \sigma_1, \quad \sigma_2 \subseteq \tau_2.
\]

We can use these two proofs to construct a proof of \( \sigma \subseteq \tau \) ending in an application of rule (arrow). This proves the lemma. \( \blacksquare \)

Now back to the problem of assigning types to lambda terms. Three well-known rules [Curry and Feys 58, Damas and Milner 82, Hindley 83a, Hindley 83b] are

\[\text{(var) } C, A \supset x: \sigma \quad \text{whenever } A(x) = \sigma,\]

\[\text{(app) } C, A \supset M: \sigma \rightarrow \tau, \quad C, A \supset N: \sigma \vdash C, A \supset MN: \tau,\]

\[\text{(abs) } C, A \supset [\sigma/x] M: \tau \vdash C, A \supset \lambda x. M: \sigma \rightarrow \tau.\]

These three rules are called the Curry typing rules. The coercion rule for typing lambda terms, based on the rules for deducing inclusions, is

\[\text{(coerce) } C, A \supset M: \sigma, \quad C \vdash \sigma \subseteq \tau \vdash C, A \vdash M: \tau.\]

The four rules (var), (app), (abs) and (cont) are called the Curry rules with containment, or CC. It is relatively easy to see that these rules are sound.

An interesting property of the Curry and coercion rules is the following generalization of the Subject Reduction Theorem of [Curry and Feys 58]. The lemma shows that types, as defined by CC, are closed under \( \beta, \eta \)-reduction (n.b. not conversion). The lemma is interesting in itself, and will be also be used in the completeness proof and its corollaries.

**Lemma 4.2:** (Subject Reduction Lemma) If \( CC \vdash C, A \supset M: \sigma \) and \( M \beta, \eta \)-reduces to \( N \), then \( CC \vdash C, A \supset N: \sigma.\)

**Proof:** Let us assume that the lemma holds when \( M \) is a redex and \( N \) is obtained by contracting \( M \). We will show that the lemma follows from this special case. Suppose \( M \) is a term with a term \( P \) which is a \( \beta \)- or \( \eta \)-redex. We can write \( M = Q[P/x] \) for some term \( Q \). Let \( N \) be the result of contracting the redex \( P \) in \( M \). It is easy to show by induction on the structure of \( Q \) that if \( C, A \supset M: \sigma \) is provable, then so is \( C, A \supset N: \sigma \). Thus whenever \( M \) reduces to \( N \) by a single reduction step, the lemma holds. In general, \( M \) may reduce to \( N \) by more than one reduction step. By induction on the length of the reduction path, we can prove the lemma. We must now prove the
lemma for the special case that $M$ is a redex.

We consider $\eta$-reduction first. Assume that the statement $C,A \vdash \lambda x.Mx : \sigma$ is provable for $x$ not free in $M$. We wish to show that $C,A \vdash M : \sigma$ is provable. For some type $\tau$, there is a proof of $C,A \vdash \lambda x.Mx : \tau$ which ends in a use of rule (abs) and such that $C \vdash \tau \subseteq \sigma$. Since the proof of $C,A \vdash \lambda x.Mx : \tau$ ends in a use of rule (abs), the type $\tau$ must be of the form $\tau_1 \rightarrow \tau_2$. Furthermore, we must have a proof of the antecedent of (abs),

$$C, A[\tau_1/x] \vdash Mx : \tau_2.$$ 

It follows that for some type $\rho$ with $C \vdash \rho \subseteq \tau_2$, there is a proof of

$$C, A[\tau_1/x] \vdash Mx : \rho$$

that ends in a use of rule (app). Hence, for some type $\nu$, the statements

$$C, A[\tau_1/x] \vdash M : \nu \rightarrow \rho$$

and

$$C, A[\tau_1/x] \vdash x : \nu$$

are both provable.

Since we can prove $C, A[\tau_1/x] \vdash x : \nu$, we know that $C \vdash \tau_1 \subseteq \nu$. Thus, by (arrow),

$$C \vdash \nu \rightarrow \rho \subseteq \tau_1 \rightarrow \tau_2$$

and so by rule (trans)

$$C \vdash \nu \rightarrow \rho \subseteq \sigma.$$ 

From this we conclude that the there is a proof of $C, A \vdash M : \sigma$.

The remaining case is $\beta$-reduction. Assume that $C,A \vdash (\lambda x.M) N : \sigma$ is provable. We wish to conclude that $C,A \vdash M[N/x] : \sigma$ is provable. There is some type $\tau$ with $C \vdash \tau \subseteq \rho$ such that $C,A \vdash (\lambda x.M) N : \tau$ has a proof that ends in a use of rule (app). Thus for some type $\rho$, we have proofs of

$$C,A \vdash \lambda x.M : \rho \rightarrow \tau$$

and

$$C,A \vdash N : \rho.$$ 

A straightforward induction on the structure of $M$ shows that if $C,A[\rho/x] \vdash M : \nu$ and $C,A \vdash N : \rho$ are provable then so is $C,A \vdash M[N/x] : \nu$. This proves the lemma. $\blacksquare$

The $\text{CC}$ rules are not semantically complete. This follows from the fact that every term which is typable using these rules looks just like a term of the simple typed lambda calculus, and every subterm of every term of that calculus has a normal form. However, the expression

$$(\lambda x. \lambda y.y) (\lambda x.xx \lambda x.xx).$$

93
has a subterm without a normal form but the term is semantically equal to the typable term \( \lambda y.y \). Therefore, this term has a type semantically, but it cannot be typed according to the rules above.

We obtain a semantically complete set of rules if we add

\[
(\text{equal}) \quad C, A \triangleright M : \sigma, \quad M =_\beta N \vdash C, A \triangleright N : \sigma.
\]

Rule (equal) and the CC rules comprise the system CC\(_{eq}\). The equality rule gives us a complete system and accounts for the undecidability of the consequences of the typing rules.\(^{14}\) Note that the theory of \( \beta \)-conversion, as well as the theory of \( \beta, \eta \)-conversion, has the property

\((*)\) if \( M, \beta, \eta \)-reduces to \( M \) and \( e = e_1 \), then there exists \( e_2 \) such that \( c_1 \beta, \eta \)-reduces to \( e_2 \) and \( e_2 = f \)

(cf. [Hindley 83a]).

The following lemmas generalize results found in [Hindley 83a].

**Lemma 4-3:** (Equality Postponement) If \( CC_{eq} \vdash C, A \triangleright M : \sigma \), then there is a proof of \( C, A \triangleright M : \sigma \) in which all CC rules appear before any uses of (equal). Equivalently, if \( CC_{eq} \vdash C, A \triangleright M : \sigma \), there is some \( M \) such that \( CC \vdash C, A \triangleright N : \sigma \) and \( M = N \).

Lemma 4-3 is proved by an easy induction on the length of proofs.

**Lemma 4-4:** If \( C, A \triangleright M : \sigma \) is provable, possibly using (equal), and \( M \beta, \eta \)-reduces to \( N \), then there is a proof of \( C, A \triangleright N : \sigma \).

**Proof:** Suppose \( CC_{eq} \vdash C, A \triangleright M : \sigma \). Then there is some \( P \) with \( M = P \) and \( CC \vdash C, A \triangleright P : \sigma \). By the subject reduction theorem, if \( P \beta, \eta \)-reduces to \( Q \), then \( C, A \triangleright Q : \sigma \) is provable. It remains to show that there exists such a \( Q \) with \( Q = N \). But this is just property \((*)\) above.

We can now prove the completeness of \( CC_{eq} \). The proof is a direct generalization of the completeness proof of [Hindley 83a].

**Theorem 1:** The \( CC_{eq} \) inference rules are sound and complete for deducing valid typing statements \( C, A \triangleright M : \sigma \).

**Proof:** Assume that \( C, A \triangleright M : \sigma \) holds in all models. Without loss of generality, we

\(^{14}\) This type inference rule can be tailored to specific lambda theories. In addition to \( =_\beta \), equality can be interpreted as equivalence under \( \beta, \eta \)-conversion. With \( \beta \)-conversion, the system is complete for all lambda models. If we choose \( \beta, \eta \)-conversion instead, then we have a proof system which is complete for all models of \( \eta \)-reduction. In fact, Theorem 1 holds whenever \( =_\eta \) is interpreted as equivalence in any lambda theory that extends the theory of \( \beta \)-conversion (cf. [Barendregt 81]), provided that property \((*)\) holds (cf. [Hindley 83a]).
may assume that the only variables typed by $A$ are those that occur in $M$ (cf. [Hindley 83a]). This assumption ensures that there are infinitely many variables which are not typed by $A$. Extend $A$ to a type assignment $B$ such that every type expression is assigned to infinitely many variables. This is so that for every term $M$ and type expression $\tau$, there will be a variable $x$ not appearing in $M$ such that $C, B \supset x : \tau$ is provable. We will construct a model $\langle D, \ast, \varepsilon \rangle$ with environment $\eta$ satisfying $C, B$ with the property that

$$\eta \models M : \sigma \iff CC_{eq} \vdash C, B \supset M : \sigma.$$ 

Since $B$ extends $A$, it will follow from the fact that $C, A \supset M : \sigma$ is valid that $C, A \supset M : \sigma$ holds in this model, and hence must be provable.

Let $\langle D, \ast, \varepsilon \rangle$ be a term model. Recall that $[M]$ denotes the equivalence class of the term $M$. Let $\eta$ be the environment which maps the variable $x$ to $[x]$, and

$$\eta(a) = \{ [M] \mid CC_{eq} \vdash C, B \supset M \}.$$ 

It follows from the usual term model construction (cf. [Barendregt 81, Meyer 82]) that for every term $M$,

$$\llbracket M \rrbracket_\eta = [M].$$

To see that $\eta$ satisfies $C$, note that if $s \subseteq t \in C$, then for every $M$, if $[M] \in \llbracket s \rrbracket_\eta$, then $C, B \supset M : s$ is provable and hence $C, B \supset M : t$ may be proved using rule (coerce). We will see that $\eta$ satisfies $B$, and that the rules are complete, by showing that

$$[M] \in \llbracket \sigma \rrbracket_\eta \iff C, B \supset M : \sigma$$

is provable.

The argument will proceed by induction on the structure of type expressions.

For a type variable $t$, the equivalence is a trivial consequence of the definition. Consider a functional type $\sigma \to \tau$. Suppose that the statement

$$C, B \supset M : \sigma \to \tau$$

is provable. For any term $N$, if there is a proof of $C, B \supset N : \sigma$, we can prove $C, B \supset MN : \tau$ using rule (app). So, for every $N$,

$$\text{if } [N] \in \llbracket \sigma \rrbracket_\eta, \text{ then } [MN] \in \llbracket \tau \rrbracket_\eta.$$ 

Hence, by definition of $\llbracket \sigma \to \tau \rrbracket_\eta$, we have $[M] \in \llbracket \sigma \to \tau \rrbracket_\eta$.

For the converse, assume that $[M] \in \llbracket \sigma \to \tau \rrbracket_\eta$. Let $N$ be any term. If there is a proof $C, B \supset N : \sigma$, then $[N] \in \llbracket \sigma \rrbracket_\eta$ by the inductive hypothesis. Furthermore, if $[N] \in \llbracket \sigma \rrbracket_\eta$, then

$$[M] \cdot [N] = [MN] \in \llbracket \sigma \rrbracket_\eta$$

and hence $C, B \supset MN : \tau$ must be provable. In particular, suppose that $N$ is a variable $x$ with $x : \sigma \in B$. By construction of $B$, we know there exists such a variable $x$ which does not occur in $M$. Then there is a proof of

$$C, B \supset Mx : \tau$$

95
and so, by rule (abs), we can prove
\[ C, B \vdash \lambda x. M : \sigma \rightarrow \tau. \]
Therefore, by Lemma 4, \( CC_{eq} \vdash C, B \supset M : \sigma \rightarrow \tau. \]

The completeness theorem has two interesting corollaries: CC is semantically complete for typing terms in normal form and the three containment rules are complete for deducing consequences of coercion sets.

**Corollary 1.1:** If \( M \) is in \( \beta \)-normal form and \( C, A \supset M : \sigma \) is valid in all lambda models, then \( CC \vdash C, A \supset M : \sigma \).

**Proof:** Suppose \( C, A \supset M : \sigma \) is valid. Then \( CC_{eq} \vdash C, A \supset M : \sigma \). By the equality postponement lemma, there is some \( N \) with \( N = M \) and \( CC_{eq} \vdash C, A \supset N : \sigma \). But since \( M \) is in normal form, we conclude that \( N \) reduces to \( M \). So by the subject reduction lemma, it follows that \( C, A \supset M : \sigma \) is provable without rule (equal).

**Corollary 1.2:** If \( \sigma \subseteq \tau \) holds in every model and type environment satisfying a coercion set \( C \), then \( C \vdash \sigma \subseteq \tau \).

**Proof:** Note that if \( C \) semantically implies \( \sigma \subseteq \tau \), then \( C, \{ x : \sigma \} \mid x : \tau \) must be valid for any variable \( x \). Since \( x \) is in normal form, this typing statement is provable using rule (var), (app), (abs) and (coerce). But then it is easy to see that the only applicable rules are (var) and (coerce). Thus \( C \vdash \sigma \subseteq \tau \).

The proofs of both corollaries rely on equality postponement and the subject reduction lemma. Although rule (arrow) is not used in the proof of Theorem 1, it is used critically in the proof of the subject reduction lemma. Since Theorem 1 depends on rule (arrow) only in that (arrow) ensures that types are closed under \( \eta \)-reduction, the rule

(\text{eta}) \quad \quad \quad C, A \supset \lambda x. M : \sigma \vdash C, A \supset M : \sigma, \ x \text{ not free in } M

events the need for rule (arrow). Note that rule (eta) is a sound, derived rule (by the subject reduction lemma), even for nonextensional models.

It is interesting to note that Corollary 1.2 also holds for the much more specialized "ideal" model of types [MacQueen] for lambda models which are complete partial orders (cf. [MacQueen and Sethi 82, MacQueen, Plotkin and Sethi 84]). In the ideal model, the only sets that are considered to be types are sets with specific order properties.
4.5 A Type Checking Algorithm

An important feature of the Curry typing with coercions is that whenever we can prove a typing statement \( C, A \supset M : \sigma \), we know that all subterms of \( M \) are typable. An intuitively appealing principle is that an expression should be typable only if all its subterms are typable. This principle, along with the fact that typing using (equal) is algorithmically intractable, suggests that we should base a type checking algorithm on the proof system without rule (equal). In the remaining sections of the paper, we consider only Curry typing with coercions, not semantic typing characterized by the system \( CC_{eq} \) with (equal).

While a statement \( C, A \supset M : \sigma \) tells the type of \( M \) subject to some assumptions about coercions and types of variables, it does not tell us the types of subterms of \( M \). As a consequence, it is difficult to tell by inspection whether a statement is provable from the CC rules. Furthermore, a provable statement \( C, A \supset M : \sigma \) does not give any information about which types might have been given to bound variables in the proof of the statement. We will be able to analyze the type checking algorithm TYPE more easily, and associate types with binding occurrences of bound variables, if we use more detailed formulas that include the types of all subterms.

An explicitly typed term \( M \) is a term \( M \) together with a mapping from subterms of \( M \) to type expressions. We will be informal about the mapping and write types as subscripts whenever we need to mention them explicitly, i.e. \( M_\sigma \) denotes an explicitly typed term \( M \) whose type is \( \sigma \).

An explicit typing statement is a statement \( A \supset M \). We can think of an explicit typing statement as a conjunction of typing statements about subterms of \( M \). Thus the CC rules could be considered rules for explicit typing statements. Following the terminology of [Milner 78], we call the CC-provable explicit typing statements well-typings. A special class of well-typings are the normal well-typings. Normal well-typings are defined as follows.
(var) \( C,A \ni x_o \) is a normal well-typing if \( C \vdash A(x) \subseteq \sigma \).

(app)_{\text{normal}} C,A \ni M_{\sigma} N_{\tau} is a normal well-typing if
\[ C,A \ni M_{\sigma} \text{ and } C,A \ni N_{\tau} \text{ are normal well-typings and } \sigma = \tau \rightarrow \rho \text{ for some } \rho \]

(abs) C,A \ni (\lambda x_{\rho} N_{\tau})_{\sigma} is a normal well-typing if
\[ C, A[\rho/x] \ni N_{\tau} \text{ is a normal well-typing and } \sigma = \rho \rightarrow \tau \]

Well-typings may also be defined using an inductive definition as above, but with

(app) C,A \ni M_{\sigma} N_{\tau} is a well-typing if
\[ C,A \ni M_{\sigma} \text{ and } C,A \ni N_{\tau} \text{ are well-typings and } \]
\[ \sigma = \mu \rightarrow \rho \text{ for some } \mu, \rho \text{ such that } C \vdash \tau \subseteq \mu. \]

instead of (app)_{\text{normal}}. Note that every normal well-typing is a well-typing. Since there is an efficient algorithm for checking whether \( C \vdash \tau \subseteq \rho \) (see Lemma 4-5 below) we can easily decide whether a candidate \( C,A \ni M_{\sigma} \) is in fact a well-typing.

Lemma 4-5: The predicate \( C \vdash \tau \subseteq \sigma \) is decidable in linear time, given a subroutine for the transitive closure of \( C \).

The proof is straightforward using Lemma 4-1.

We have the following lemma relating well-typings to provable statements. The lemma may also be interpreted as a normal form lemma for proofs in CC. Lemma 5 implies that it is only necessary to use rule (coerce) immediately following a use of rule (var).

Lemma 4-6: A statement \( C, A \ni M: \sigma \) is provable (without the equality rule (equal)) iff there is an explicit typing \( M_{\sigma} \) of \( M \) such that \( C,A \ni M_{\sigma} \) is a normal well-typing.

Proof: It is easy to show by induction on formulas that if \( C,A \ni M_{\sigma} \) is a well-typing, then \( C,A \ni M: \sigma \) is provable. The converse is essentially a normal form theorem for CC proofs. Note that (var) is the only axiom scheme and so every proof is essentially a tree with an instance of (var) at each leaf. We think of each node as labeled by both the statement proved at that node and the final rule used in that proof. Given a proof of a statement \( C,A \ni M: \sigma \), define the degree of the proof to be the number of pairs of internal tree nodes \( \langle \alpha, \beta \rangle \) such that there is a path from a leaf through \( \alpha \) to \( \beta \), node \( \alpha \) is labeled with a rule different from (coerce), and node \( \beta \) is labeled with rule (coerce). Intuitively, the degree gives us a measure of how far the occurrences of
(coerce) are from the leaves. We show by induction on the degree of a proof that every provable statement has a proof of degree zero.

We need a preliminary fact about proofs for the case in which a node labeled (coerce) follows a node labeled (abs). Suppose we are given a proof of $C, A[\sigma/x] \vdash M: \tau$ and that $C \vdash \rho \subseteq \sigma$. Then we can produce a proof of $C, A[\rho/x] \vdash M: \tau$ by replacing every leaf labeled with the statement $C, A[\sigma/x] \vdash x: \sigma$ by a short proof of this statement beginning with $C, A[\rho/x] \vdash x: \rho$ and then using (coerce). Note that the proof we produce has the same degree as the proof we start with. It is now a simple matter to prove by induction on the degree of proofs that every provable statement has a proof of degree zero.

Suppose we have a node labeled (coerce) following another node labeled (coerce). Then we can collapse these two proof steps into one using rule (trans) for inclusions. We now consider (app). Suppose $C, A \supseteq MN: \tau$ follows from $C, A \supseteq M: \sigma \rightarrow \tau$ and $C, A \supseteq N: \sigma$ by rule (app) and then $C, A \supseteq MN: \rho$ follows by (coerce). We have $C \vdash \tau \subseteq \rho$. Therefore, by rule (arrow),

$$C \vdash (\sigma \rightarrow \tau) \subseteq (\sigma \rightarrow \rho).$$

So we can derive $C, A \supseteq M: \sigma \rightarrow \rho$ from $C, A \supseteq M: \sigma \rightarrow \tau$ by rule (coerce) and then proceed to use rule (app) to derive $C, A \supseteq MN: \rho$. This reduces the degree of the proof by one.

The final case is a node labeled (coerce) following a node labeled (abs). Suppose the proof has a path with nodes labeled

$$C, A[\sigma_1/x] \supseteq M: \sigma_2$$

$$C, A \supseteq \lambda x. M: \sigma_1 \rightarrow \sigma_2 \quad \text{(by rule abs)}$$

$$C, A \supseteq \lambda x. M: \rho_1 \rightarrow \rho_2 \quad \text{(by rule coerce).}$$

We want to move the use of rule (coerce) above the use of rule (abs). Note that since $C \vdash \sigma_1 \rightarrow \sigma_2 \subseteq \rho_1 \rightarrow \rho_2$, we have $C \vdash \rho_1 \subseteq \sigma_1$ and $C \vdash \sigma_2 \subseteq \rho_1$ by Lemma 1. By the preliminary fact noted above, there is a proof of $C, A[\rho_1/x] \supseteq M: \sigma_2$ with the same degree as the proof of $C, A[\sigma_1/x] \supseteq M: \sigma_2$. Now, applying rule coercé, we can prove $C, A[\rho_1/x] \supseteq M: \rho_2$ and so, by rule (abs), $C, A \supseteq \lambda x. M: \rho_1 \rightarrow \rho_2$. This reduces the degree of the proof by one and proves the lemma.

Given a term $M$, Algorithm TYPE will deduce a well-typing $C, A \supseteq M_\sigma$ if any well-typing for $M$ exists. The coercion set $C$ will contain coercions. It is tempting, at first glance, to try to generalize the algorithms of [Hindley 69] or [Milner 78] to find typings relative to some given,
fixed set of coercions between type constants. For example, given a coercion set \( C \) and a closed term \( M \), we might try to compute an explicitly typed term \( M_\emptyset \) such that for every well-typing \( C \), \( A \supseteq M_\tau \) the term \( M_\tau \) would be a substitution instance of \( M_\tau \). Without coercion sets, this is possible [Damas and Milner 82, Hindley 69]. However, this cannot be done for all coercion sets, as the example given in the introduction demonstrates.

4.5.1 Substitution and Coercion Set Algorithms

Algorithm TYPE will use three subsidiary algorithms. One computes coercions sets and the other two produce substitutions. A substitution is a function from type variables to type expressions. If \( \sigma \) is a type expression and \( S \) is a substitution, then \( S\sigma \) is the type expression obtained by replacing each variable \( t \) in \( \sigma \) by \( S(t) \). If \( M \) is an explicitly typed term, then \( SM \) is the result of applying \( S \) to every type in \( M \). A substitution \( S \) applied to a type assignment \( A \) is the assignment \( SA \) with \( (SA)(x) = S(A(x)) \). We define an equivalence relation on types before defining the effect of a substitution on a coercion set. After these definitions, we consider the substitution algorithms.

Intuitively, two type expressions \( \text{match} \) if they have the same form. More precisely,

(i) if \( \sigma \) is a type variable, then \( \sigma \) matches \( \tau \) iff \( \tau \) is a type variable

(ii) if \( \sigma = \sigma_1 \rightarrow \sigma_2 \), then \( \sigma \) matches \( \tau \) iff

\[ \tau = \tau_1 \rightarrow \tau_2 \text{ and } \sigma_1 \text{ matches } \tau_1 \text{ and } \sigma_2 \text{ matches } \tau_2. \]

It is easy to verify that matching is an equivalence relation on types. Furthermore, for any type \( \sigma \), there is a type \( \rho \), the most general type that matches \( \sigma \), with the property that if \( \sigma \) matches \( \tau \), then \( \tau \) is a substitution instance of \( \rho \). The most general type that matches \( \sigma \) may be produced from \( \sigma \) just by renaming occurrences of type variables so that no type variable appears twice in the expression.

Recall that a coercion set \( C \) may only contain coercions between type variables, not type expressions. For example, the coercion set \( \{ t \rightarrow t \subseteq t \} \) is not well-formed. In order to use substitution as a mechanism for generating new well-typings, we have to define the action of a substitution \( S \) on a coercion set \( C \). Given any coercion \( \sigma \subseteq \tau \) between matching type expressions
σ and τ, there is a unique minimal coercion set that implies σ ⊆ τ.

Lemma 4-7: Let σ and τ be matching type expressions. There is a coercion set C with C ⊢ σ ⊆ τ and such that if C₁ ⊢ σ ⊆ τ, then C₁ ⊢ C.

The lemma is proved by observing that a coercion set C implies

σ₁ ⊢ σ₂ ⊆ τ₁ ⊢ τ₂

only if C implies τ₁ ⊆ σ₁ and C implies σ₂ ⊆ τ₂. The minimal coercion set that implies σ ⊆ τ will be denoted COERCΣ(σ, τ). The naive algorithm for COERCΣ(σ, τ) runs in linear time. A substitution S respects coercion set C if, for every σ ⊆ τ in C, we have Sσ matches Sτ. If S respects C, then we define the action of S on C by

SC = \bigcup_{σ ⊆ τ ∈ C} COERCΣ(σ, τ).

Suppose A₁ and A₂ are type assignments with the same finite domain. Robinson's unification algorithm, UNIFY(A₁, A₂), can be used to find a most general substitution S that gives SA₁ = SA₂. If no such substitution exists, then UNIFY will fail.

Lemma 4-8: There is an algorithm UNIFY(A₁, A₂) with which computes a substitution S with SA₁ = SA₂ if any such substitution exists. Furthermore, if R is any substitution with RA₁ = RA₂, then there is a substitution T with R = T ⋆ S.

Proof: See [Robinson 65].

There are efficient, even linear, implementations of unification (cf. [Paterson and Wegman 78]).

We will also use a modified version of UNIFY to compute most general matchings. Two assignments A₁ and A₂ match if they have the same domain and, for every x in their domain, A₁(x) matches A₂(x). The algorithm TYPE uses an algorithm MATCH to compute matching substitutions that respect certain coercion sets. The algorithm MATCH(C, A₁, A₂) computes a most general substitution S that causes A₁ to match A₂ and that respects C. The algorithm, a modification of unification, fails if there is no substitution S such that SA₁ matches SA₂ and S respects C. More formally,

Lemma 4-9: Let C be a coercion set and A₁ and A₂ be type assignments. If S = MATCH(C, A₁, A₂), then S respects C and SA₁ matches SA₂. Furthermore, if R is a substitution that respects C such that RA₁ matches RA₂, then S = MATCH(C, A₁, A₂) succeeds and R = T ⋆ S for some substitution T.
The straightforward details of algorithm MATCH are left to the reader.\textsuperscript{15} Given any finite set of type variables $\forall$, we may assume that the range of $S = MATCH(C, A_1, A_2)$ does not use any elements of $\forall$, since matching is invariant under renaming of variables. This assumption will be used in the proof of Theorem 3.

If $A$ is a type assignment, $A \setminus x$ denotes the assignment which is identical to $A$, but not defined on $x$. If $A_1$ and $A_2$ are type assignments, then $A_1 \setminus A_2$ is the assignment $A_3$ with $A_3(x) = \sigma$ iff $A_1(x) = \sigma$ and $x$ is not in the domain of $A_2$.

**Instances of Well-Typings**

Well-typings are preserved by substitutions that respect coercion sets. Let $C, A \supset M_\sigma$ be a well-typing. A well-typing $C', A' \models M_\tau$ an instance of $C, A \supset M_\sigma$ if there exists a substitution $S$ which respects $C$ such that

$$C' \leftarrow SC, \quad A' |_{FV(e)} = SA |_{FV(e)}', \quad \text{and} \quad M_\tau = SM_\sigma.'$$

Here $FV(M)$ denotes the set of variables that appear free in $M$. Note that $SC, SA \supset SM_\sigma'$ is an instance of $C, A \supset M_\sigma$. We have

**Lemma 4-10:** Let $C, A \supset M_\sigma$ be a well-typing. Every instance of $C, A \supset M_\sigma$ is a well-typing.

The lemma is proved by induction on the structure of $M$.

**4.5.2 Algorithm TYPE**

The algorithm is written below in an applicative style. Given an untyped expression $M$, the algorithm either returns a well-typing $C, A \supset M$ or \textit{fails}. The "overbars" on explicitly typed terms are omitted for typographical reasons.

\textsuperscript{15}This variant of unification is \textit{not} algorithm Match of [Dwork, Kanellakis and Mitchell 84].
\[ \text{TYPE}(M) = \]

\text{cases}

\text{M is a variable } x:

\text{let } s \text{ and } t \text{ be new type variables}
\text{return } \{ s \subseteq t \}, \{ x : s \} \supset x_t

\text{M is an application } NP:

\text{let } C_1, A_1 \supset N_\sigma = \text{TYPE}(N)
\text{C}_2, A_2 \supset P_\tau = \text{TYPE}(P)
A_3 = A_1 \cup (A_2 \setminus A_1)
A_4 = A_2 \cup (A_1 \setminus A_2)
R = \text{MATCH}(C_1 \cup C_2, A_3[\sigma/z], A_4[\tau \mapsto t/z])
\text{where } z \text{ and } t \text{ are new variables}
S = \text{UNIFY}(R A_3, R A_4) \circ R
C = \text{COERCES}(S \tau, \text{Left}(S \sigma))
\text{return } \] C \cup S C_1 \cup S C_2, S A_3 \supset S N_\sigma P_\tau

\text{M is an abstraction } \lambda x.N

\text{let } C, A, M_\tau = \text{TYPE}(N)
\text{return } C, A \setminus x \supset (\lambda x A(x) \cdot N_{\tau}) A(x) \mapsto \tau
\text{end cases}

The function \text{Left}(\sigma) used in the application case returns \sigma_1 if \sigma is of the form \sigma_1 \rightarrow \sigma_2 and is undefined otherwise. It is easy to see that the value of \text{Left} is defined in the call above. The algorithm may \text{fail} in the application case if either the call to MATCH or to UNIFY \text{fails}. If \text{TYPE}(M) does not \text{fail}, then it produces a well-typing.

\textbf{Theorem 2:} Let } C, A \supset M_\rho = \text{TYPE}(M). \text{ Then every instance of } C, A \supset M_\rho \text{ is a well-typing.}

Conversely, if there exists a well-typing for } M, \text{ then } \text{TYPE}(M) \text{ will return a well-typing.

\textbf{Theorem 3:} Suppose } C, A \supset M_\rho \text{ is a well-typing. Then } \text{TYPE}(M) = C_1, A_1 \supset M_\sigma \text{ succeeds. Furthermore, } C, A \supset M_\rho \text{ is an instance of } C_1, A_1 \supset M_\sigma.

\text{Theorem 2 is proved by induction on the structure of terms.}

\textbf{Proof of Theorem 2:} By Lemma 4.10, it suffices to show that if Algorithm TYPE
does not fail, it produces a well-typing. The proof is by induction on the structure of terms. It is easy to see that TYPE(x) is always a well-typing.

Consider TYPE(MN). By the inductive assumption, both

\[ C_1, A_1 \vdash M_\sigma = \text{TYPE}(M), \quad \text{and} \]

\[ C_2, A_2 \vdash N_\tau = \text{TYPE}(N) \]

are well-typings. Note that since RA_1 and RA_2 are renamed the substitution UNIFY(RA_1, RA_2) is only a renaming of variables and therefore respects all coercion sets. In addition, S_\sigma matches S(\tau \rightarrow \iota). Furthermore, since S respects C_1 \cup C_2,

\[ SC_1, SA_3 \supset SM_\sigma \quad \text{and} \quad SC_2, SA_4 \supset SN_\tau \]

are well-typings. It follows from the properties of MATCH and UNIFY that SA_1 = SA_2 and S_\tau matches Left(S_\sigma). Furthermore, since

\[ C \cup SC_1 \cup SC_2 \vdash S_\tau \subseteq \text{Left}(S_\sigma), \]

C \cup SC_1 \cup SC_2, SA_1 \supset SM_\sigma N_\tau \text{ is a well-typing.} \]

Finally, consider an abstraction \( \lambda x.M \). By the inductive assumption,

\[ C, A, M_\tau = \text{TYPE}(M) \]

is a well-typing. Thus C, (A \& x) \( \supset x_\alpha \), \( M_\alpha \) must be a well-typing. This finishes the proof of Theorem 2. \( \blacksquare \)

**Proof of Theorem 3:** The theorem is proved by induction on the structure of terms. Suppose C, A \( \supset x_\sigma \) is a well-typing. Then TYPE(x) returns

\[ \{s \subseteq t\}, \{x:s\} \supset x_t. \]

Let \( \tau = A(x) \) and let T be the substitution \( [\tau, \sigma/s, t] \). Then certainly

\[ A|_{FV(e)} = (T|x:s)|_{FV(e)} \quad \text{and} \quad Tx_t = x_\sigma. \]

Furthermore, since C, A \( \supset x_\sigma \) is a well-typing, we must have C \( \vdash T\{s \subseteq t\} \). Thus C, A \( \supset x_\sigma \) is an instance of TYPE(x).

Suppose C, A \( \supset \mu \rightarrow \nu M_k \) is a well-typing. Consider the definition of TYPE(MN) from TYPE(M) and TYPE(N). By the inductive hypothesis,

\[ C, A \supset \mu \rightarrow \nu M_k \text{ is an instance of } C_1, A_1 \mid N_\sigma \text{ and} \]

\[ C, A \mid M_k \text{ is an instance of } C_2, A_2 \mid M_\tau, \]

where C_1, C_2, etc. are as defined in the application case of Algorithm TYPE. Since A_1 assigns types to all free variables in M we have A_1 \( |_{FV(M)} = A_3 \mid_{FV(M)} \). Similarly, A_2 \( |_{FV(N)} = A_4 \mid_{FV(N)} \). Therefore,
\( C', A^\prime \mid \mathcal{N}_{\mu \rightarrow \nu} \) is an instance of \( C_1, A_3 \mid \mathcal{N}_\sigma \) and

\( C', A^\prime \mid \mathcal{M}_{k} \) is an instance of \( C_2, A_4 \mid \mathcal{M}_{\tau} \).

We assume that UNIFY and MATCH always use new variables, so no type variables in \( C_2, A_4 \supset \mathcal{M}_{\tau} \) appear in \( C_1, A_3 \supset \mathcal{N}_\sigma \). Therefore, both instances above are by a single substitution \( T \) that preserves \( C_1 \cup C_2 \). By the properties of MATCH, we know that \( T = T_R \circ R \) for some \( T_R \), where \( R \) is as in Algorithm TYPE. Furthermore, by the properties of UNIFY, we can see that there must be some substitution \( T_S \) such that \( T = T_S \circ S \), where again \( S \) is as in Algorithm TYPE.

We have now shown that

\[
C \vdash T_S(SC_1 \cup SC_2), \text{ and}
\]

\[
A \mid_{FV(e')} = T_SSA_1 = T_SSA_2.
\]

Since \( C, A \supset \mathcal{M}_\rho \) is a well-typing we must have

\[
C \vdash T_R COERCER(S_T, Left(S_\sigma)).
\]

It follows that \( C, A \supset \mathcal{M}_\rho \) is an instance of TYPE(MN).

Finally, consider an abstraction \( \lambda x.M \). Suppose \( C, A \supset \lambda x_{\mu}.M_{\nu} \) is a well-typing. Then, by definition of well-typing,

\[
C, A[\mu/x] \mid \mathcal{M}_{\nu}
\]

is a well-typing. By the inductive hypothesis, this well-typing is an instance of

\[
\text{TYPE}(M) = C', A' \supset \mathcal{M}_{\tau}.
\]

This means that there must be a substitution \( S \) such that

\[
C \vdash SC', A[\mu/x] \mid_{FV(e')} = SA \supset \mathcal{N}_{FV(e')} \text{ and } \mathcal{M}_{\nu} = SM_{\tau}.
\]

So \( A \mid_{FV(e')} = S(A' \setminus x) \mid_{FV(e')} \) and \( \lambda x_{\mu}.M_{\nu} = S(\lambda x_{A'(x)}.M_{\tau}) \). This concludes the proof of Theorem 3.

A simple modification to Algorithm TYPE inserts calls to conversion functions. The modification, discussed in Section 6, does not affect the running time of the algorithm.

4.6 Variations: Conversion Functions and ML. Polymorphism

Algorithm TYPE calculates a well-typing of a term, but does not insert any conversion functions. This is consistent with the view that whenever a type \( \sigma \) is coercible to another type \( \tau \),
the values of type $\sigma$ are also of type $\tau$. However, in practice, elements of type $\sigma$ and elements of $\tau$ may have different concrete representations. When this is the case, it is necessary to apply a conversion function to a value of type $\sigma$ in order for it to behave properly as a value of type $\tau$. A simple variation of algorithm TYPE will insert calls to conversion functions.

We assume that for every coercion $\sigma \subseteq \tau$ there is some conversion function $h_{\sigma, \tau}$ mapping values of type $\sigma$ into type $\tau$. The function $h_{\sigma, \tau}$ will be added to a term only when $\sigma \subseteq \tau$ is implied by the coercion set at hand. The types $\sigma$ and $\tau$ are considered parts of the expression $h_{\sigma, \tau}$ so that when a substitution is applied to a term containing $h_{\sigma, \tau}$, type variables in $\sigma$ and $\tau$ may be changed.

If $\sigma$ and $\tau$ are not atomic types, then we can construct $h_{\sigma, \tau}$ from conversion functions between atomic types. Given conversion functions for each coercion in $C$, we can define conversion functions for each $\sigma \subseteq \tau$ implied by $C$. If $C \vdash \sigma \subseteq \tau$, then we define the typed lambda term $h_{\sigma, \tau}$ by induction on the proof of $\sigma \subseteq \tau$ from $C$. Given any proof of $\sigma \subseteq \tau$, we define $h_{\sigma, \tau}$ as follows:

(i) If $\sigma \subseteq \tau$ follows from $\sigma \subseteq \rho$ and $\rho \subseteq \tau$, then define

$$h_{\sigma, \tau} = h_{\rho, \tau} \circ h_{\sigma, \rho}.$$

(ii) If $\sigma = \sigma_1 \rightarrow \sigma_2$, $\tau = \tau_1 \rightarrow \tau_2$ and $\sigma \subseteq \tau$ follows from $\tau_1 \subseteq \sigma_1$ and $\sigma_2 \subseteq \tau_2$, then define

$$h_{\sigma, \tau} = \lambda f_{\sigma_1} h_{\sigma_2, \tau_2} \circ M \circ h_{\tau_1, \sigma_1}.$$

Thus for every $\sigma \subseteq \tau$ provable from $C$, we have a conversion function mapping $\sigma$ into $\tau$.

The only modification to TYPE is in the application case. The return statement in the second case reads

```
return ... S M_{\sigma} N_{\tau}.
```

Substitute

```
return ... SM_{\sigma}(h_{\tau, Left(\sigma)} N_{\tau}).
```

The inductive proof of Theorem 2 may be modified to show that if the modified version of TYPE succeeds, it produces an explicit typing statement $C$, $SA \supset N_{\sigma}$ that can be proved from rules (var), (app) and (abs) only. Furthermore, if each $h_{\sigma, \tau}$ in $N$ is replaced by the identity
function, then $N$ reduced to $M$. The proof of Theorem 3 is may also be modified to show that the new algorithm is also guaranteed to find a typing whenever one exists.

4.6.1 ML Polymorphism

The algorithm $\text{TYPE}_{\text{c..n}}$ also be extended to lambda calculus with a polymorphic let construct as in ML [Gordon, et. al. 79, Milner 78]. From a theoretical point of view, the simplest way to do this is to introduce let as an abbreviation for a $\lambda$-term. We define $\text{let}$ by

$$\text{let } x = M \text{ in } N ::= [M/x]N.$$ 

It follows immediately that algorithm $\text{TYPE}$ finds most general typings for terms containing $\text{let}$. Moreover, we can adapt $\text{TYPE}$ to $\text{let}$ expressions directly so that it is not necessary to perform substitutions on terms.

Since Algorithm $\text{TYPE}$ deduces a typing for each subterm independently, the algorithm will type every occurrence of $M$ in $[M/x]N$ by precisely the same process. If we wish to type

$$\text{let } x = M \text{ in } N,$$

we compute $\text{TYPE}(M) = C, A \vdash M : \sigma$ once and proceed to type $N$ as usual. However, as each occurrence of $x$ in $N$ is encountered, we simply rename all type variables in $C, A \vdash M : \sigma$ and treat this as the typing for $x$. An easy induction on the structure of $N$ shows that this version of $\text{TYPE}$ finds precisely the same typing for $\text{let } x = M \text{ in } N$ as for $[M/x]N$.

4.7 Conclusion and Future Directions

As with Curry typing without coercions, a relatively simple set of inference rules is sufficient to deduce all semantically valid typing statements. However, semantic completeness is achieved at the cost of making the set of types of a term undecidable. A type inference algorithm for the decidable set of inference rules is presented. Algorithm $\text{TYPE}$ allows automatic coercions to be added to programming languages like ML. It may also be used to insert calls to conversion functions at compile time. One appealing property of algorithm $\text{TYPE}$ is that a term has a type only if all subterms have types.

The simple language of type schemes without binding operators does not allow some useful
expressions to be typed. More complicated type disciplines, e.g. [Fortune, et. al. 83, Leivant 83a, MacQueen and Sethi 82 Reynolds 74], let more complicated expressions be given types, without sacrificing the principle that an expression is typable only if every subexpression is typable. Containment of types is a crucial issue in type inference for these type disciplines. It is hoped that the study of type containment presented here will be of use in developing more flexible type inference procedures.

There are a number of ways that the work described in this paper might be extended. For example, we have limited our attention to simple coercions between atomic types. It seems like a relatively minor variation to allow coercions of the form $\sigma \subseteq \tau$ as long a $\sigma$ matches $\tau$, but we have not worked out the details. A more significant generalization is to allow coercions between arbitrary types, e.g. $s \rightarrow t \subseteq s$. Coercions of this form will certainly change the set of typable terms drastically, as remarked in Section 3. Do the inference rules remain complete for this kind of containment, and can the algorithm be modified to find most-general type deductions with this form of coercion?

Another topic is to consider different semantics for the type connective $\rightarrow$. Two possibilities are the quotient-set semantics [Hindley 83a] and the F-semantics [Hindley 83b]. It seems likely that the techniques of [Hindley 83a, Hindley 83b] will suffice to prove completeness theorems for typing with coercions for both of these semantics.

In [Reynolds 80], a much more general notion of coercion is considered. What bearing do the present results have on more general coercions that do not seem related to containments?

In this paper, we have shown that the CC rules are decidable, and that with the addition of an equality rule, the inference rules are semantically complete. This shows that typing with the simple coercions we have considered has essentially the same properties as typing without without coercions. However, we have not found any reason not to extend the decidable system by adding more sound inference rules. For example, we could adopt an axiom saying that the term

$$(\lambda x \lambda y. y) (\lambda x. x),$$

which reduces to the identity, has type $t \rightarrow t$. This extended typing system would still be sound
according to our semantics, and certainly decidable. Is there any natural property that CC has that this new, decidable notion of typing does not have? More generally, is there some semantic property of the CC rules which in some way distinguishes them from sound extensions to these rules?
Chapter Five

Type Inference with Polymorphic Types

5.1 Introduction

The second-order lambda calculus, developed independently by Girard [Girard 71] and Reynolds [Reynolds 81b] is a typed language. Type expressions appear in the syntax of the ordinary expressions (terms) of the language and figure prominently in defining the set of well-formed terms. The semantics of second-order lambda calculus [Bruce and Meyer 84, Mitchell 84a] naturally involves giving meanings to type expressions and using the meanings of type expressions to define the meanings of terms. This paper studies systems for assigning types from second-order lambda calculus to untyped lambda terms. While we begin with typing rules inspired by the second-order lambda calculus, we are led to a very different semantic interpretation of types and, consequently, to additional typing rules. A semantic interpretation of second-order type assignment over arbitrary models of untyped lambda calculus is presented, generalizing previous work by MacQueen, Sethi and Plotkin [MacQueen and Sethi 82, MacQueen, Plotkin and Sethi 84]. We prove completeness theorems for the valid type assignments over all models, and over a more specialized class of "simple semantics" models. In addition, complete inference rules are presented for deducing the valid containments between types in arbitrary models or "simple semantics" models.

The rules for inferring types for untyped terms are based on the notion of type erasures. In the second-order lambda calculus, there is a clear distinction between type expressions and the other constituents of terms. Consequently, it is clear what it means to erase types from terms; we use $\text{erase}(M)$ to denote the result of removing all type expression from second-order term $M$. Since the erasure of any second-order term is a term of untyped lambda calculus, we can associates types with untyped terms as follows: if $M$ is a second-order term and the type of $M$ is $\sigma$, then the untyped term $\text{erase}(M)$ has type $\sigma$. The GR inference rules presented in Section 2 capture this notion of "having a type" exactly. In general, an untyped term may have many
types, or none at all.

In a semantic view of type inference, we associate each type with a set of meanings of terms. If M has type \( \sigma \), then the meaning of M is an element of the set associated with \( \sigma \). A general definition of models of type inference is given in Section 3. In order to achieve semantic completeness, we add a rule (equal) that gives equal terms the same type. The extended system, \( \text{GR}_{eq} \), is sound and complete for all inference models.

The set of types associated with the identity function has an appealing semantic interpretation: the identity \( \lambda x.x \) has type \( \sigma \rightarrow \tau \) iff the containment between types \( \sigma \subseteq \tau \) is valid. We study containments by reformulating the inference rules so that containments play a central role. The GRC inference rules, presented in Section 4, are equivalent to the GR rules, but use a single containment rule (cont) instead of the two GR rules for manipulating type quantifiers.

A specific class of inference models, the simple inference models, are studied in Section 5. In simple inference models, the type connective \( \rightarrow \) is interpreted according to the simple semantics of, e.g., [Barendregt, Coppo and Dezani 83, Hindley 83a], i.e. the function-space type \( \sigma \rightarrow \tau \) is interpreted as the set of all elements of the untyped model that map \( \sigma \) into \( \tau \). The GRS rules, an extension of the GR rules, are sound and complete for all simple inference models. In addition, the GRS rules can be reformulated using the containment rule instead of two quantifier rules and a rule based on \( \eta \)-reduction of lambda terms. The containments that are valid in all simple models distinguish the simple semantics for \( \rightarrow \) from other interpretations.

5.2 Syntactic Type Inference for Second-Order Lambda Calculus

We begin by reviewing second-order lambda calculus \( \Lambda \) developed by Girard and Reynolds in [Girard 71, Reynolds 74]. Type expressions are built from type variables and constants using the connective \( \rightarrow \) and the binding operator \( \forall \). We adopt the notational conventions that

\[ r, s, t, \ldots \text{ denote type variables} \]

\[ \rho, \sigma, \tau, \ldots \text{ denote type expressions.} \]

The set of type expressions is defined by the grammar

\[ \tau ::= t \mid \sigma \rightarrow \tau \mid \forall \sigma \]

111
We identify type expressions that differ only in the names of bound variables. Thus $\forall t. \sigma(t) = \forall s. \sigma(s)$. We use type assignments rather than typed variables in the definition of $SA$. A type assignment $A$ is a function from a set of variables that may appear in lambda terms to type expressions. There is no loss of generality in assuming that the domain of each type assignment is finite. A type assignment $A$ with $\text{dom}(A)$ finite can be written as a finite set of statements of the form $x: \sigma$. If $x$ is a variable, $\sigma$ a type expression and $A$ a type assignment, then $A[\sigma/x]$ is a type assignment with $(A[\sigma/x])(y) = A(y)$ for any variable $y$ different from $x$, and $(A[\sigma/x])(x) = \sigma$.

For each type assignment $A$, we define a partial function $Type_A$ from strings of symbols to type expressions. We let the set $SA_A$ of terms that are well-typed in context $A$ be the domain of $Type_A$. The function $Type_A$ is defined to be the least function satisfying the following conditions.

(i) $Type_A(x) = A(x)$ if $x$ is in the domain of $A$,

(ii) If $Type_A(M) = \sigma \rightarrow \tau$ and $Type_A(N) = \sigma$, then $Type_A(MN) = \tau$,

(iii) If $Type_A[\sigma/x](M) = \tau$, then $Type_A(\lambda x \in \sigma. M) = \sigma \rightarrow \tau$,

(iv) If $Type_A(M) = \forall t. \sigma(t)$ then $Type_A(M\tau) = \sigma(\tau)$,

(v) If $Type_A(M) = \tau$ and $t$ does not occur free in the type of $A(x)$ for any $x$ free in $M$, then $Type_A(\Pi t. M) = \forall t. \tau$.

We will often leave the assignment $A$ implicit and speak of the language $SA$ of second-order lambda calculus. A term of $SA$ is closed if it has no free ordinary variables. If $M$ is closed, then $Type_A(M)$ does not depend on $A$ and so we may just as well write $Type(M)$ for the type of $M$.

5.2.1 Type Inference Rules

We use typing statements to describe the type of an untyped expression, given the types of free variables. Informally, the statement

$A \vdash M: \sigma$

means that if variables have the types assigned by $A$, then the expression $M$ has type $\sigma$. The semantics of type statements will be given in the next section. If $M$ is a term and $A$ a type
assignment, the type variable \( t \) is bindable in \( M \) with respect to \( A \) if \( t \) does not occur free in the type of \( A(x) \) for any \( x \) free in \( M \). The following axiom and inference rules will be referred to as the GR-inference rules.

(var) \( A \supset x: \sigma \) if \( A(x) = \sigma \)

(app) \( A \supset M: \sigma \rightarrow \tau, A \supset N: \sigma \vdash A \supset MN: \tau \)

(abs) \( [\sigma/x]A \supset M: \tau \vdash A \supset \lambda x.M: \sigma \rightarrow \tau \)

(inst) \( A \supset M: \forall t.\sigma(t) \vdash A \supset M: \sigma(\tau) \),

(gen) \( A \supset M: \sigma \vdash A \supset M: \forall t.\sigma \),

whenever \( t \) is bindable in \( M \) with respect to \( A \).

We say that the statement \( A \supset M: \sigma \) is GR-provable, written

\[ \text{GR} \vdash A \supset M: \sigma \]

if the statement can be proved using the GR-inference rules. For any term of \( \Sigma \Lambda \), we define \( \text{erase}(M) \), the erase of \( M \), to be the result of removing all type symbols from \( M \), i.e.,

\[ \text{erase}(x) = x, \]

\[ \text{erase}(MN) = \text{erase}(M) \text{ erase}(N), \]

\[ \text{erase}(\lambda x.\epsilon \tau. M) = \lambda x.\text{erase}(M), \]

\[ \text{erase}(M \tau) = \text{erase}(M), \] and

\[ \text{erase}(\Pi t.M) = \text{erase}(M). \]

Note that \( \text{erase}(M) \) has precisely the same free ordinary variables as \( M \).

The explicitly typed language \( \Sigma \Lambda \) is related to the GR-inference rules by the following lemma.

**Lemma 5.1:** If \( M \) is a typed term of \( \Sigma \Lambda \) with \( \text{Type}_A(M) = \sigma \), then \( \text{GR} \vdash A \supset \text{erase}(M): \sigma \). Conversely, if \( M \) is an untyped term and \( A \) is a type assignment with \( \text{GR} \vdash A \supset M: \sigma \), then there is a typed term \( N \) of \( \Sigma \Lambda \) with \( \text{Type}_A(N) = \sigma \) and with \( \text{erase}(N) = M \).

The proof of the Lemma, which is essentially Theorem 4.1 of [Leivant 83a], is completely straightforward. Beta-reduction, \( \rightarrow_\beta \), and \( \beta \)-convertibility, \( =_\beta \), are standard notions in lambda
calculus [Barendregt 81]. Both are discussed briefly in Section 3. It is not hard to see that 
$\beta$-redexes in a typed term $M$ correspond exactly to $\beta$-redexes in $\text{erase}(M)$. Therefore, using
Lemma 5-1, we have

**Lemma 5-2:** If $GR \vdash A \supset M : \sigma$ and $M$ $\beta$-reduces to $N$, then $GR \vdash A \supset N : \sigma$.

A converse of this lemma fails: if $GR \vdash A \supset N : \sigma$, and $M$ $\beta$-reduces to $N$, then $A \supset M : \sigma$ may
not be provable. Since $GR \vdash A \supset M : \sigma$ implies that $M$ is strongly normalizable [Fortune, et. al.
83, Girard 71], there will be many terms equal to $\lambda x.x$, for example, which cannot be typed
using the GR rules. One term that cannot be typed is

$$(\lambda w \lambda x.x)(\lambda y.yy \lambda y.yy),$$

even though this term reduces to $\lambda x.x$ in one step. The term is not strongly normalizing since
the subterm $(\lambda y.yy \lambda y.yy)$ may be reduced infinitely many times.

5.3 Semantics of Type Inference

If we are to interpret typing statements $A \supset M : \sigma$ semantically, then we need interpretations for
untyped lambda expressions, type expressions, and the relation $:$ between lambda terms and
type expressions. A model $<D, \cdot, \varepsilon>$ of the untyped lambda calculus consists of a set $D$,
together with a binary operation $\cdot$, for application, and a "choice element" $\varepsilon$. (Models will be
discussed in more detail below.) Given an untyped model $<D, \cdot, \varepsilon>$, it seems natural to
interpret types as subsets of $D$ and $:$ as set membership. For technical reasons, we do not
interpret type expressions as subsets of $D$ directly. Instead, type expression will denote
elements of some set $T$ of types, and each $a \in T$ will be associated with a subset $D_a \subseteq D$. This
approach to interpreting type expressions leads to much simpler completeness proofs than the
more direct interpretation. We will also need to assume that the subsets $\{D_a\}_{a \in T}$ have certain
properties. These are motivated in the following discussion.

In studies of type inference without quantifiers, e.g., [Barendregt, Coppo and Dezani
83, Hindley 83a, Hindley 83b, Milner 78, Mitchell 84c], we simply need an environment
mapping type variables to arbitrary subsets of $D$. We then define the meanings of expressions
$\sigma \rightarrow \tau$ using some interpretation of $\rightarrow$ as an operation on sets. Two possibilities (see [Hindley
83a] for discussion) are

114
(simple) $A \rightarrow B = \{ d \in D \mid \forall e \in A, d \cdot e \in B \}$

and

(function) $A \rightarrow B = \{ d \in D \mid \forall e \in A, d \cdot e \in B \} \cap \mathcal{S}$,

where $\mathcal{S}$ is the range of $\varepsilon$. The significance of $\mathcal{S}$ is that every unary function from $D$ to $D$ that is represented by some element of $D$ (by left application) has a unique representative in $\mathcal{S}$. Intuitively, $\mathcal{S}$ is the set of functions in the model.\(^{16}\)

While it might be appealing to allow any set of elements to be a type, there are difficulties when quantifiers are introduced. It is natural to take the meaning of $\forall t. \sigma$ as an intersection. If all sets are considered types, then we are led to the naive definition

(naive) $\llbracket \forall t. \sigma \rrbracket \eta = \bigcap_{S \subseteq D} \llbracket \sigma \rrbracket \eta[S/t]$.

However, there are some significant problems with this. For example, consider the type $\forall t. t$. By (naive), the meaning of $\forall t. t$ is the empty set in every model. This makes any typing statement of the form $A[\forall t. t/x] \subseteq \mathcal{M}: \sigma$ vacuously valid, since the assignment $A[\forall t. t/x]$ is unsatisfiable. No environment $\eta$ can satisfy $\eta(x) \in \llbracket \forall t. t \rrbracket_{(naive)}$ since $\llbracket \forall t. t \rrbracket_{(naive)}$ is empty. It seems somewhat pathological to have empty types, since $A \subseteq \mathcal{M}: \sigma$ should be independent of $A(x)$ for $x$ not free in $\mathcal{M}$.

A better approach to the semantics of $\forall$ is developed in [MacQueen and Sethi 82, MacQueen, Plotkin and Sethi 84]. If we restrict our attention to models with order structure, then we can single out ideals, sets which are nonempty, "downward closed" and "closed under limits of chains", as the meanings of types. We can then define

(ideal) $\llbracket \forall t. \sigma \rrbracket \eta = \bigcap_{I \subseteq D \text{ an ideal}} \llbracket \sigma \rrbracket \eta[I/t]$.

All type expressions denote nonempty ideals and typing does not degenerate as in the naive semantics of $\forall$. However, since the ideal model seems to be a commitment to a number of

\(^{16}\)In [Hindley 83a, Hindley 83b], the interpretation (function) is called the $F$-semantics. We use (function), and Function semantics, to avoid conflict with the operator $F$ defined in the next Section.
complicated relationships between types, we will not adopt this model directly. Instead, we develop a more general definition which allows the ideal model of [MacQueen and Sethi 82, MacQueen, Plotkin and Sethi 84] as an important special case. We review models of untyped lambda calculus and discuss models of \( \mathcal{SA} \) briefly before defining inference models.

5.3.1 Untyped Lambda Calculus

Recall that the terms of untyped lambda calculus are given by

\[
M ::= x \mid MN \mid \lambda x.M.
\]

A lambda model \( \langle D, \cdot, e \rangle \) consists of a set \( D \) together with binary operation \( \cdot \) and element \( e \) of \( D \) such that

\[
\forall d, e \ (e \cdot d) \cdot e = d \cdot e, \text{ and}
\]

\[
\forall e \ (d_1 \cdot e = d_2 \cdot e) \text{ implies } e \cdot d_1 = e \cdot d_2.
\]

Furthermore, \( D \) must contain elements \( K \) and \( S \) with simple algebraic properties. This is the combinatory model definition of [Meyer 82]; see also [Barendregt 81].

Given a lambda model \( \langle D, \cdot, e \rangle \) and environment \( \eta \) mapping variables to elements of \( D \), the meaning of a lambda term \( M \) is defined inductively by

\[
\llbracket x \rrbracket \eta = \eta(x)
\]

\[
\llbracket MN \rrbracket \eta = \llbracket M \rrbracket \eta \cdot \llbracket N \rrbracket \eta
\]

\[
\llbracket \lambda x.M \rrbracket \eta = e \cdot d, \text{ where } d \cdot e = \llbracket M \rrbracket \eta[e/x] \text{ all } e \in D
\]

The existence of \( K \) and \( S \) ensure that there always exists a \( d \) as required in the definition of \( \llbracket \lambda x.M \rrbracket \). The function \( e \) makes the meaning of \( \lambda x.M \) independent of the specific choice of \( d \). Again, the reader is referred to [Barendregt 81, Meyer 82] for specifics.

A few facts about the reduction rules of lambda calculus are used. See [Barendregt 81] for a

---

17 One advantage of the ideal model is that it provides interpretations for recursively-defined types [MacQueen, Plotkin and Sethi 84], which the models discussed below generally do not.

18 It is also convenient, in some contexts, to require that \( e \) be idempotent. This is not entirely necessary since \( e^2 \) is idempotent whenever \( e \) has the properties described above; see [Meyer 82] for details.
comprehensive presentation. We consider lambda terms modulo $\alpha$-conversion

$$(\alpha) \quad \lambda x.M = \lambda y.[y/x]M \text{ if } y \text{ is not free in } M$$

so that we can rename bound variables freely. The reduction rules are

$$(\beta) \quad (\lambda x.M)N \rightarrow [N/x]M,$$

$$(\eta) \quad \lambda x.Mx \rightarrow M \text{ if } x \text{ is not free in } M,$$

where substitution is defined with renaming of bound variables to avoid capture. If a term $M$ is of the form of the left-hand side of rule $(\beta)$ or $(\eta)$, then $M$ is a $\beta$- or $\eta$-redex, respectively. We say that $M$ $\beta$-reduces to $N$ in one step if there is a subterm $P$ of $M$ which is a $\beta$-redex and $N$ is the result of contracting this redex in $M$. The term $M$ $\beta$-reduces to $N$ if there is a sequence of $\beta$-reductions leading from $M$ to $N$. If we allow $\eta$-reduction in addition to $\beta$-reduction, then we say $M$ $\beta,\eta$-reduces to $N$. A term which cannot be reduced is in normal form. Conversion is the least congruence relation containing reducibility; $=_{\beta}$ denotes $\beta$-conversion and $=_{\beta,\eta}$ denotes $\beta,\eta$-conversion.

One important model is the term model $<D, \cdot, \epsilon>$ with

$$D = \{ [M] \mid M \text{ an untyped term} \},$$

where $[M]$ denotes the $=_{\beta}$-equivalence class of $M$. Application, $\cdot$, in term models is defined by

$$[M] \cdot [N] = [MN]$$

and choice element $\epsilon$ by

$$\epsilon = [\lambda x. y. xy].$$

See [Barendregt 81, Meyer 82] for properties of term models.

5.3.2 Semantic Models of Second-Order Lambda Calculus

General definitions of models of second-order lambda calculus, and specific model constructions, have been put forth in [Bruce and Meyer 84, Haynes 84, Leivant 83b, McCracken 79, Mitchell 84a]. We use a general definition based on [Bruce and Meyer 84, Mitchell 84a]. We begin by defining the kind of algebraic structure that is needed to interpret type expressions. These will play an important role in the definition of inference model. A type structure is a tuple $<T, F, \Pi>$, where

(i) $T$ is any set (called the set of types),
(ii) \( F \) is a function \( F : T \times T \to T \), and

(iii) \( \Pi : [T \to T] \to T \) for some set \([T \to T]\) of functions from \( T \) to \( T \)

The function \( F \) is used to interpret the connective \( \to \) and the function \( \Pi \) for the quantifier \( \forall \).

Given an environment \( \eta \) mapping type variables to elements of \( T \), the meaning of a type expression is defined inductively as follows.

\[
\llparenthesis t \rrparenthesis_\eta = \eta(\alpha)
\]

\[
\llparenthesis \sigma \to \tau \rrparenthesis_\eta = F(\llparenthesis \sigma \rrparenthesis_\eta)(\llparenthesis \tau \rrparenthesis_\eta)
\]

\[
\llparenthesis \forall \cdot \sigma \rrparenthesis_\eta = \Pi f, \text{ where } f \in [T \to T] \text{ has the property that for all } a \in T, \ f a = \llparenthesis \sigma \rrparenthesis_\eta[a/t].
\]

We define a type algebra to be any type structure with the domain \([T \to T]\) of \( \Pi \) sufficiently large to allow all type expression to be interpreted.

A second-order applicative frame is a tuple \( \langle \mathcal{S}, \{ D_a \}, \{ \ast \} \rangle \) with

(i) \( \mathcal{S} = \langle T, F, \Pi \rangle \) a type algebra

(ii) \( \{ D_a \}_{a \in T} \) a family of sets \( D_a \) indexed by elements of \( T \)

(iii) For each \( a, b \in T \), we have a function \( \ast_{a,b} : D_{Fab} \to D_a \to D_b \).

(iv) For every \( f \in [T \to T] \), we have a function \( \ast_f : D_{\Pi f} \to \Pi_{a \in T} D_{fa} \).

We refer to \( \ast_{a,b} \) as application on \( D_{Fab} \) and \( \ast_f \) as type application on \( D_{\Pi f} \). Both \( \ast_{a,b} \) and \( \ast_f \) will be written as infix binary operations. In addition, we will omit the subscripts when it is unlikely to cause confusion.

A second-order combinatory algebra is an applicative frame closed under definition of polynomials. More precisely, a frame \( \mathcal{S} = \langle \mathcal{S}, \{ D_a \}, \ldots \rangle \) is combinatorially complete if for every applicative term \( M \) (possibly containing constants for elements of \( \mathcal{S} \)), sequence \( s^+ \) of type variables, and sequence \( x^+ \) of ordinary variables such that all free variables of \( M \) are among \( s^+ \) and \( x^+ \), there is a constant \( d \) of \( \mathcal{S} \) such that

\[
\mathcal{S} \vdash M = d \ast s^+ \ast x^+.
\]

This definition, taken from [Mitchell 84a], is similar to the usual definition of combinatory completeness for untyped lambda calculus [Barendregt 81, Meyer 82], but with the added consideration of type variables. It can be shown that an applicative frame is combinatorially
complete iff it contains certain combinators I, K, S, A, B, C and D. A combinatory algebra falls short of being a second-order model since it lacks the choice elements described in [Mitchell 84a]. While second-order combinatory algebras may be used to interpret terms of $SA$ without the binding operators $\lambda$ or $\Pi$, the lack of choice elements precludes them from satisfying the weak extensionality property of $SA$ with $\lambda$ and $\Pi$.

5.3.3 Models of Type Inference

A model of type inference will be a triple $\langle \mathcal{G}, \mathcal{A}, \text{subset} \rangle$ with $\mathcal{G} = \langle T, F, \Pi \rangle$ a type algebra, $\mathcal{A} = \langle D, \cdot, \varepsilon \rangle$ a model of untyped lambda calculus, and subset a function from $T$ to nonempty subsets of $D$. We write $D_a$ for $\text{subset}(a)$, $a \in T$. The model definition will also involve some conditions on the sets $\{D_a\}_{a \in T}$. Conditions (F) and (P) will ensure that $D_{Fab}$ is a reasonable (but not extensional) representation of the set of functions from $D_a$ to $D_b$ and that $D_{\Pi f}$ is a set of polymorphic elements.

A model of type inference, or inference model for short, is a triple $\langle \mathcal{G}, \mathcal{A}, \text{subset} \rangle$ with $\mathcal{G} = \langle T, F, \Pi \rangle$ a type algebra, $\mathcal{A} = \langle D, \cdot, \varepsilon \rangle$ a model of untyped lambda calculus, subset a function from $T$ to nonempty subsets of $D$.

such that for all $a, b \in T$ and $f \in [T \rightarrow T]$,

(F) If $d \in D_{Fab}$, then $d \cdot D_a \subseteq D_b$. Furthermore, if $d \cdot D_a \subseteq D_b$, then $\varepsilon d \in D_{Fab}$.

(P) $D_{\Pi f} = \bigcap_{a \in T} D_{fa}$.

Inference models are related to second-order combinatory algebras by the following lemma.

Lemma 5-3: Let $\langle \mathcal{G}, \mathcal{A} \rangle$ be an inference model and consider the following definitions.

(i) Define $\cdot_{a,b} : D_{Fab} \rightarrow D_a \rightarrow D_b$ by $d \cdot_{a,b} e = d \cdot e$.

(ii) Define $\cdot_f : D_{\Pi f} \rightarrow \Pi_{a \in T} D_{fa}$ by $d \cdot_f a = d$.

Then $\langle \mathcal{G}, \{D_a\}_{a \in T}, \{\cdot\} \rangle$ is a second-order combinatory algebra.

An inference model is a second-order model only if it contains the choice elements $\{\varepsilon\}$ described in [Mitchell 84a]. Given an inference model, there is a natural associated combinatory algebra, but not necessarily a natural way of finding an associated second-order
model.

We interpret typing statements in inference models using environments that map type variables to elements of \( T \) and ordinary variables to elements of \( D \). An environment \( \eta \) for inference model \( \mathfrak{S} \) satisfies a type assignment \( A \), written \( \eta \vdash A \), if

\[
\ll x \rr \eta \in D \ll \Lambda(x) \rr \eta \quad \text{all} \ x \ \text{in} \ \text{dom}(A),
\]
and satisfies a typing \( M: \sigma \), written \( \eta \vdash M: \sigma \), if

\[
\ll M \rr \eta \in D \ll \sigma \rr \eta
\].

An environment \( \eta \) satisfies a typing statement, written \( \eta \vdash A \supset M: \sigma \), if

\[
\eta \vdash A \implies \eta \vdash M: \sigma
\].

A typing statement is valid if it is satisfied by every model and environment.

5.4 Completeness and Type Containment

5.4.1 Extensions to GR

The GR inference rules cannot be semantically complete because they depend on the form of a term, while the validity of a typing statement \( A \supset M: \sigma \) depends only on the meaning of \( M \). Since\( GR \vdash A \supset M: \sigma \) implies that \( M \) is strongly normalizable [Fortune, et. al. 83, Girard 71], there will be many terms equal to \( \lambda x.x \), for example, which cannot be typed using the GR rules. We obtain a complete system by adding the inference rule (equal) that gives equal terms the same types. The equality inference rule is

\[
(\text{equal}) \quad A \supset M: \sigma, M = N \vdash A \supset N: \sigma.
\]

We use \( GR_{eq} \) to denote the system consisting of the GR rules and the inference rule (equal).

A useful fact about proofs in GR or \( GR_{eq} \) is that no proof of \( A \supset M: \sigma \) depends on \( A(x) \) for \( x \) not free in \( M \).

**Lemma 5-4:** Suppose \( GR \vdash A \supset M: \sigma \). If \( GR \vdash A_1 \supset x:A(x) \) for all \( x \) free in \( M \), then \( GR \vdash A_1 \supset M: \sigma \). Similarly for \( GR_{eq} \).

Lemma 5-4 is proved by a trivial induction on proofs. The following two lemmas are useful, and easy to prove.

120
Lemma 5.5: Let \( \overline{r} \) be a sequence of \( n \) type variables and \( \overline{\rho} \) a sequence of \( n \) type expressions. If \( \Gamma \vdash A \supset M : \sigma \), then \( \Gamma \vdash [\overline{\rho}/\overline{r}]A \supset [\overline{\rho}/\overline{r}]\sigma \).

Lemma 5.6: If \( \Gamma \vdash A[\sigma/x] \supset M : \tau \) and \( \Gamma \vdash A \supset N : \sigma \), then \( \Gamma \vdash A \supset [N/x]M : \tau \).

5.4.2 Soundness and Completeness

It is relatively straightforward to show that the inference rules are sound.

Lemma 5.7: If \( \Gamma \vdash A \supset M : \sigma \), then whenever \( \eta \vdash A \), we have
\[
\llbracket M \rrbracket \eta \in D_{\llbracket \sigma \rrbracket \eta} \quad \text{and}
\llbracket N \rrbracket \eta \in D_{\llbracket \sigma \rrbracket \eta}
\]

Therefore, by property (F) of inference models,
\[
\llbracket MN \rrbracket \eta \in D_{\llbracket \tau \rrbracket \eta}
\]

If \( A \vdash \lambda x. M : \sigma \rightarrow \tau \) follows from \( A[\sigma/x] \vdash M : \tau \), then by the inductive hypothesis the function
\[
g = \lambda d \in D_{\llbracket \sigma \rrbracket \eta} \cdot \llbracket M \rrbracket \eta[d/x]
\]

maps \( D_{\llbracket \sigma \rrbracket \eta} \) into \( D_{\llbracket \tau \rrbracket \eta} \) and so by property (F) we have \( \llbracket \lambda x. M \rrbracket \eta \in D_{\llbracket \sigma \rightarrow \tau \rrbracket \eta} \).

The cases for (v-inst) and (equal) are trivial. For (v-gen), suppose that
\[
\Gamma \vdash A \supset M : \forall t. \sigma
\]

by (v-gen) and assume that \( \eta \vdash A \). We may assume, by Lemma 5.4, that there is some type variable \( t \) which does not appear free in range(\( A \)). For any \( a \in T \), the environment \( \eta[a/t] \) satisfies \( A \). By the inductive hypothesis,
\[
\llbracket M \rrbracket \eta = \llbracket M \rrbracket \eta[a/t] \in D_{\llbracket \sigma \rrbracket \eta[a/t]}
\]

Therefore, by condition (II),
\[
\llbracket M \rrbracket \eta \in \bigcap_{a \in T} D_{\llbracket \sigma \rrbracket \eta[a/t]} \subseteq D_{\llbracket \forall t. \sigma \rrbracket \eta}
\]

This finishes the proof of the Lemma. \( \square \)

We now show that the rules are complete. The proof is similar in spirit to the completeness proof of [Hindley 83a].

Theorem 1: The \( \Gamma_{\text{eq}} \) inference rules are sound and complete for deducing the
typing statements that are valid over all inference models.

Proof: Let \( A_0 \supset M_0 : \sigma_0 \) be a typing statement that is not \( \text{GR}_{eq} \)-provable. We assume, by Lemma 5.4, that \( \text{dom}(A_0) \) is finite. Extend \( A_0 \) to an assignment \( A \) such that for every type expression \( \sigma \), there are infinitely many variables \( x \) with \( A(x) = \sigma \). We construct an inference model \( \langle \mathcal{G}, \mathcal{D} \rangle \) in which \( A_0 \supset M_0 : \sigma_0 \) fails.

Let \( \mathcal{G} = \langle D, \ast, \varepsilon \rangle \) be the term model, so that

\[
D = \{ [M] | M \text{ is an untyped term} \}.
\]

It remains to define \( \mathcal{D} = \langle T, F, \Pi \rangle \) and subsets \( D \subseteq D \). Let \( T \) be the set of type expressions (modulo renaming of bound variables) with \( [T \rightarrow T] \) the set of functions of the form \( \lambda t.\tau \), where \( \tau \) is any type expression. Define \( F \tau = \sigma \rightarrow \tau \) and \( \Pi(\lambda t.\tau) = \forall t.\tau \). Take

\[
D_\sigma = \{ [M] | \text{GR}_{eq} \vdash A \supset M : \sigma \}.
\]

Let \( \eta_0 \) be the environment mapping each ordinary variable \( x \) to its equivalence class \( [x] \) and mapping each type variable \( t \) to itself. Then, by the usual properties of term models, \( [M]_0 = [M] \) and \( [\sigma]_0 = \sigma \). Thus \( \eta_0 \vdash A_0 \) but

\[
[M_0]_0 = [M_0] \notin D \sum [\sigma]_0 = D_\sigma.
\]

In order to prove completeness, we only need to check that \( \langle \mathcal{G}, \mathcal{D} \rangle \) is an inference model. We verify property (F) first. Suppose that \( [M] \in D_{\sigma \rightarrow \tau} \). Then \( \text{GR}_{eq} \vdash A \supset M : \sigma \rightarrow \tau \) and, similarly, for any \( [N] \) in \( D_\sigma \), we have \( \text{GR}_{eq} \vdash A \supset N : \sigma \). Therefore, \( \text{GR}_{eq} \vdash A \supset MN : \tau \) by rule (app) and so \( [M] \ast [N] = [MN] \in D_\tau \). This shows that the first part of (F) holds. It remains to be shown that if \( [M] \ast D_\sigma \subseteq D_\tau \), then \( \varepsilon[M] \in D_{\sigma \rightarrow \tau} \). Here we use the fact that \( A \) supplies infinitely many variables of each type. For if \( [M] \ast D_\sigma \subseteq D_\tau \), then there is some variable \( x \) not free in \( M \) with \( A(x) = \sigma \). So, by definition of \( D_\tau \), we have

\[
\text{GR}_{eq} \vdash A \supset Mx : \tau.
\]

Therefore, by rule (abs),

\[
\text{GR}_{eq} \vdash A \supset \lambda x. Mx : \sigma \rightarrow \tau.
\]

This shows that \( [\lambda x. Mx] = \varepsilon[M] \in D_{\sigma \rightarrow \tau} \), demonstrating property (F).

We show that property (I) holds by demonstrating both containments. If \( [M] \in D_{\forall t. \sigma(t)} \), then for any \( \tau \), we have \( [M] \in D_{\sigma(\tau)} \) by rule (\( \forall \)-inst). Conversely, if \( [M] \) is an element of every \( D_{\sigma(\tau)} \), then for some variable \( s \) not free in \( A(x) \) for and \( x \) free in \( M \), we have

\[
\text{GR}_{eq} \vdash A \supset M : \sigma(s).
\]

Therefore, \( \text{GR}_{eq} \vdash A \supset M : \forall s. \sigma(s) \) by (\( \forall \)-gen) and \( [M] \in D_{\forall s. \sigma(s)} = D_{\forall t. \sigma(t)} \). This demonstrates (I) and concludes the proof of the Theorem. 

We gain a little more insight into the power of \( \text{GR} \) by noting that

122
Lemma 5-8: (Equality Postponement) If $GR_{eq} \vdash A \supseteq M: \sigma$, then there is some $N$ such that $GR \vdash A \supseteq N: \sigma$ and $M = \beta N$.

This lemma is the generalization of the Equality Postponement Theorem of [Curry and Feys 58] (see also [Hindley 83a]) to $GR_{eq}$. Lemma 5-8, together with the completeness theorem and Lemma 5-2, gives us the following corollary.

Corollary: The GR rules are complete for typing terms in $\beta$-normal form.

5.4.3 Typing Rules Based on Containment

We gain a better understanding of proofs in GR by reformulating the inference rules so that proofs correspond more closely to the structure of terms. In the Curry inference system (cf. [Barendregt, Coppo and Dezani 83, Curry and Feys 58, Hindley 69, Hindley 83a, Milner 78]) consisting only of (var), (app) and (abs), every proof of a statement $A \supseteq MN$, for example, must end with rule (app). There is no other way to type the application of two terms. Similarly, typing variables can only be done using (var) and abstractions only by (abs). It is this property of the Curry system, coupled with the fact that the inference rules are defined by schemes, that gives Curry typing principal type schemes (cf. [Hindley 69]). While it seems unlikely that GR is equivalent to an inference system consisting only of "syntax-directed" rules, we will be able to replace ($\forall$-inst) and ($\forall$-gen) by a single "syntax-independent" rule (cont). In order to do so, however, we will need to alter rules (app) and (abs) slightly.

Rule (cont) is based on set-theoretic containments between types in inference models. If $A \supseteq M: \sigma$ and $\sigma \subseteq \tau$, then (cont) yields $A \supseteq M: \tau$. Since $\subseteq$ will be a transitive relation on types, two consecutive uses of the rule can always be replaced by a single use. This gives us an easily computable bound on the length of proofs in the system. However, this bound does not imply that the theory of GR-typing is decidable.

The containment-based inference system $GRC$ consists of rules (var), $(app_{\forall})$, $(abs_{\forall})$, and (cont). The new typing rules are
(app♭) A ⊩ M: ∀s↑(σ → τ), A ⊩ N: ∀s↑.σ

    \vdash A ⊩ MN: ∀s↑.τ


    where the type variables s↑ are bindable in λx.M

(cont) A ⊩ M: σ, σ ⊆ τ \vdash A ⊩ M: τ

The single rule for deducing containments is the axiom scheme

(sub) ∀Γ↑.σ ⊆ ∀Γ↑[Γ'/Γ↑]σ

    where Γ↑ ∩ FV(∀Γ↑.σ) = ∅

The substitution axiom (sub), is similar to to the notion of generic instance considered in [Damas and Milner 82]. One important special case of (sub) is ∀t.σ(t) ⊆ σ(τ), and another is σ ⊆ ∀t.σ if t is not free in σ. Some other derived rules are

(ref) σ ⊆ σ

(trans) ρ ⊆ σ, σ ⊆ τ \vdash ρ ⊆ τ.

In Section 5, we will consider a more specialized inference system GRSC tailored to simple semantics inference models. In GRSC, we will be able to eliminate the quantifier manipulation in (app♭) in favor of an additional containment rule

(dist) ∀s↑(τ → σ) ⊆ ∀s↑.τ → ∀s↑.σ

that allows us to distribute ∀ over →. However, rule (dist) is not sound for all inference models, since the typing statement

{x:∀t(τ → σ)} ⊩ x:∀t.τ → ∀t.σ

is not provable in GR♭^{eq} On the other hand, we will see that if M:∀t(τ → σ) and x does not appear free in M, then λx.Mx:∀t.τ → ∀t.σ. We will not be able to eliminate the quantifier introduction in rule (abs♭) from GRSC.

The following lemmas are used to show that GR and GRC are equivalent.

---

19. This claim follows from the corollary to the completeness theorem; rule (equal) is not needed for typing terms in normal form.
Lemma 5-9: Let $x$ be an ordinary variable that does not occur free in term $M$. Then

$$A ⊢ M : \forall \bar{s}^\tau (\sigma \rightarrow \tau) \vdash A ⊢ \lambda x. Mx : \forall \bar{s}^\tau \sigma \rightarrow \forall \bar{s}^\tau \tau$$

and

$$A ⊢ M : \forall \bar{s}^\tau (\sigma \rightarrow \tau), A ⊢ N : \forall \bar{s}^\tau \sigma \vdash A ⊢ MN : \forall \bar{s}^\tau \tau$$

using the GR rules.

**Proof:** To simplify notation, let us assume that the type variables in $\bar{s}^\tau$ are all bindable in $M$. If not, then we just rename bound type variables. For $x$ as in the statement of the Lemma, we can prove $A[\forall \bar{s}^\tau. \sigma/x] \supset Mx : \tau$ using rules $(\forall$-inst), $(\forall$-gen) and $(\forall$-app). Therefore, using rules $(\forall$-gen) followed by $(\forall$-inst), we may prove $A ⊢ \lambda x. Mx : \forall \bar{s}^\tau. \sigma \rightarrow \forall \bar{s}^\tau. \tau$. The proof of the second part of the Lemma uses rules $(\forall$-inst), $(\forall$-app) and $(\forall$-gen).

Lemma 5-10: Suppose $A ⊢ M : \sigma \vdash A ⊢ M : \tau$ using only rules $(\forall$-inst) and $(\forall$-gen). Let $\bar{s}^\tau = s_1, ..., s_n$ contain all free type variables in $\sigma$ that are type parameters of $M$. Then $\forall \bar{s}^\tau. \sigma \subseteq \tau$ is an instance of the containment axiom (sub). Conversely, if $\forall \bar{s}^\tau. \sigma \subseteq \tau$ is an instance of the containment axiom (sub), then for any term $M$, we have

$$A ⊢ M : \forall \bar{s}^\tau \sigma \vdash A ⊢ M : \tau$$

using only rules $(\forall$-inst) and $(\forall$-gen).

**Proof:** Suppose $A ⊢ M : \sigma \vdash A ⊢ M : \tau$ using only $(\forall$-inst) and $(\forall$-gen). Let $\bar{s}^\tau$ be a list containing all free type variables in $\sigma$ that are type parameters of $M$. Let $\bar{\tau}$ be the set of bindable type variables of $M$. Note that by choice of $\bar{s}^\tau$, no $\nu \in \bar{\tau}$ occurs free in $\forall \bar{s}^\tau. \sigma$. The derivation $A ⊢ M : \sigma \vdash A ⊢ M : \tau$ is sequence

$$A ⊢ M : \sigma_0 \vdash ... \vdash A ⊢ M : \sigma_m$$

with $\sigma_0 = \sigma$ and $\sigma_m = \tau$, and each $\sigma_{i+1}$ the result of either substituting for a bound variable in $\sigma_i$ or binding some variable in $\bar{\tau}$. We use induction on $i$ to show that

(*) For any $\bar{\nu}^\tau$ from $\bar{\tau}$, the containment $\forall \bar{s}^\tau. \sigma \subseteq \forall \bar{\nu}^\tau. \sigma_i$ is an instance of the axiom scheme (sub).

The case $i = 0$ is trivial.

Assume that each $\forall \bar{s}^\tau. \sigma \subseteq \forall \bar{\nu}^\tau. \sigma_i$ is an instance of (sub). If $\sigma_{i+1}$ is the result of quantifying over some variable from $\bar{\tau}$, then the (*) follows immediately from the inductive hypothesis. Otherwise, note that

$$\sigma_i = \forall u. \bar{\nu}^\tau. [\bar{\rho}^\tau / \bar{\tau}^\tau] \sigma$$

for some $u$, $\bar{\nu}^\tau$, $\bar{\tau}^\tau$ and $\bar{\rho}^\tau$. If $\sigma_{i+1}$ follows by $(\forall$-inst), then

$$\sigma_{i+1} = [\mu/u] \forall \bar{\nu}^\tau. [\bar{\rho}^\tau / \bar{\tau}^\tau] \sigma.$$
\[ \sigma_{i+1} = \forall \bar{v}^t.\left[\mu / u\right]\left[\bar{p}^r / \bar{r}^t\right] \sigma, \]

and so

\[ \forall \bar{s}^t \sigma \subseteq \sigma_{i+1} \]

is an instance of (sub). In addition, since no variable in \( \forall \) appears free in \( \forall \bar{s}^t \sigma \), each containment \( \forall \bar{s}^t \sigma \subseteq \forall \bar{v}^t.\sigma_{i+1} \) with \( \bar{v}^t \) from \( \forall \) is an instance of (sub). This proves the first part of the Lemma. The converse statement is straightforward. ■

**Theorem 2**: The inference systems GR and GRC are equivalent.

**Proof**: We first show that if \( \text{GR} \vdash A \supset M: \sigma \), then \( \text{GRC} \vdash A \supset M: \sigma \). We argue, by induction on proofs, that if \( \text{GR} \vdash A \supset M: \sigma \) and \( \bar{s}^t \) is any sequence of type variables bindable in \( M \) with respect to \( A \), then \( \text{GRC} \vdash A \supset M: \forall \bar{s}^t.\sigma \). The addition of quantifiers into rules (\( \text{app}_\forall \)) and (\( \text{abs}_\forall \)) makes these two cases straightforward. Proofs using (\( \forall \)-inst) and (\( \forall \)-gen) are handled using Lemma 5-10.

Conversely, suppose that \( \text{GRC} \vdash A \supset M: \sigma \). We show that \( \text{GR} \vdash A \supset M: \sigma \). It is easy to see that (\( \text{abs}_\forall \)) is a derived rule of \( \text{GR} \), and (\( \text{app}_\forall \)) is a derived rule by Lemma 5-9. By Lemma 5-10, (cont) is also a derived rule of \( \text{GR} \). This proves the Theorem. ■

We use \( \text{GRC}_{eq} \) to denote the system \( \text{GRC} + (\text{equal}) \). Equality Postponement (cf. Lemma 5-8) is easily verified for \( \text{GRC}_{eq} \). It follows that \( \text{GR}_{eq} \) and \( \text{GRC}_{eq} \) are equivalent.

**Corollary**: The inference system \( \text{GRC}_{eq} \) is sound and complete for all inference models.

Equality Postponement for \( \text{GRC}_{eq} \) and completeness give us two interesting corollaries.

**Corollary**: The GRC rules are complete for typing terms in normal form.

**Corollary**: Every containment that is valid in all inference models is an instance of the axiom scheme (sub).

### 5.5 Simple Semantics and Containment

In this Section, we focus on inference models with the simple semantics for \( \rightarrow \). A *simple inference model* is an inference model satisfying

\[
(F_{\text{simple}}) \quad D_{\text{Fab}} = \{ d \in D \mid \forall e \in D_a, d \cdot e \in D_b \}.
\]

We will see expect more typing statements are valid in simple inference models than are valid in arbitrary inference models.
It is easy to see that in simple inference models,
\[ [M] \eta \in D_{Fab} \quad \text{iff} \quad [\lambda x. Mx] \eta \in D_{Fab}, \]
provided \( x \) is not free in \( M \). The inference rule
\[ A \vdash M: \sigma \rightarrow \tau \quad \vdash A \vdash \lambda x. Mx: \sigma \rightarrow \tau, \quad x \text{ not free in } M \]
is easily seen to be a derived rule of GR. We adopt the converse
\[ (\text{eta}) \quad A \vdash \lambda x. Mx: \sigma \rightarrow \tau \quad \vdash A \vdash M: \sigma \rightarrow \tau, \quad x \text{ not free in } M \]
as an additional inference rule for the simple semantics. The system consisting of the GR rules, plus (eta), will be called the GRS inference system. It can be shown that GRS \( \vdash A \supset M: \sigma \) iff there is some \( N \) that \( \eta \)-reduces to \( M \) with GR \( \vdash A \supset N: \sigma \). We use GRS_{eq} to denote the GRS rules together with the equality rule (equal).

A simple modification of the proof of Theorem 1 shows

**Theorem 3:** The GRS_{eq} inference rules are sound and complete for all simple inference models.

**Proof:** The proof of Theorem 3 differs from the proof of Theorem 1 only in the verification of condition (F). We show that
\[ [M] \in D_{u \rightarrow \tau} \text{ implies } [M] \cdot \sigma, \tau \subseteq D_{\sigma} \subseteq D_{\tau} \]
as in the proof of Theorem 1. To prove the converse implication, we note that
\[ [M] \cdot \sigma, \tau \subseteq D_{\sigma} \subseteq D_{\tau} \text{ implies } [\lambda x. Mx] \in D_{\sigma \rightarrow \tau}, \]
as in the proof of Theorem 1. Then, by (eta) and the definition of \( D_{\sigma \rightarrow \tau} \), we conclude \( [M] \in D_{\sigma \rightarrow \tau} \). \( \Box \)

5.5.1 Containments in Simple Inference Models

We now reformulate the typing rules GRS for simple inference models using type containments. The GRS\( ^{SC} \) inference system consists of the typing rules (var), (app), (abs_\( \eta \)) and (cont), together with the following axioms and inference rules for containments.

**Containment Axioms**

(sub) \( \forall \Gamma^+. \sigma \subseteq \forall \Gamma'[\bar{\Gamma}' / \bar{\Gamma}] \sigma \)

where \( \bar{\Gamma}' \cap FV(\forall \Gamma^+. \sigma) = \emptyset \)

(dist) \( \forall s^+. (\tau \rightarrow \sigma) \subseteq \forall s^+. \tau \rightarrow \forall s^+. \sigma \)
Containment Inference Rules

(arrow) $\sigma_1 \subseteq \sigma$, $\tau \subseteq \tau_1 \vdash \sigma \rightarrow \tau \subseteq \sigma_1 \rightarrow \tau_1$

(trans) $\rho \subseteq \sigma$, $\sigma \subseteq \tau \vdash \rho \subseteq \tau$

(congruence) $\sigma \subseteq \tau \vdash \forall \lambda \sigma \subseteq \forall \lambda \tau$

Note that we now use rule (app) rather than (app$\forall$). Rule (app$\forall$) may now be derived from (app) and (cont) using the containment axiom (dist).

The main difficulty in establishing the equivalence of GRS and GRSC lies in proving that (eta) is a derived rule of GRSC.

**Lemma 5.11:** If GRSC $\vdash A \supset M: \sigma$ and $M \beta, \eta$-reduces to $N$, then GRSC $\vdash A \supset N: \sigma$.

Lemmas 5-10 and 5-9 may be established for the simple semantics rules using similar proofs. In addition, the containment rule can be shown to be a derived rule of GRS using induction on derivations of containments.

**Theorem 4:** The inference systems GRS and GRSC are equivalent.

We use GRSC$_{eq}$ to denote the system GRSC$_{+}$ (equal). equality postponement (cf. Lemma 5-8) is easily verified for GRSC$_{eq}$. It follows that that GRS$_{eq}$ and GRSC$_{eq}$ are equivalent.

**Corollary:** The inference system GRSC$_{eq}$ is sound and complete for all simple inference models.

Again, equality postponement and completeness provide completeness results for normal forms and containments.

**Corollary:** The GRSC rules are complete for typing terms in normal form.

**Corollary:** The simple containment rules are sound and complete for deducing the containments that are valid in all simple inference models.

**5.6 Discussion**
5.6.1 Retyping Functions

In addition to the semantic motivation for the simple containment rules, there is also a syntactic one. These rules seem to be useful for characterizing the set of types associated with a single term. In fact, we are led to the simple containment rules by considering GR-provable typings of terms.

An intuitive, syntactic notion of embedding $\sigma$ into $\tau$ is that we can write down a typed function that maps $\sigma$ to $\tau$ without essentially changing the functionality of terms. Such an embedding only "changes the type" of an element. For example, any element of $\forall t.\sigma(t)$ is mapped to an element of $\sigma(\tau)$ by the function

$$f = \lambda x \in \forall t.\sigma(t). x \{ \tau \}.$$

It is easy to see that $\text{erase}(f)$ is the identity function $I = \lambda x. x$. This suggests that we can embed the type $\forall t.\sigma(t)$ into $\sigma(\tau)$, only "changing the type" of elements of $\forall t.\sigma(t)$.

A second example will provide a little more motivation for the definition of retyping functions. We compare two types that may be derived for the closed, untyped term $S = \lambda x.y.z. (xz)(yz)$.

The principal Curry type scheme of $S$ is

$$(r \rightarrow s \rightarrow t) \rightarrow (r \rightarrow s) \rightarrow r \rightarrow t.$$

In GR, we can prove that $S$ has the quantified type

$$\sigma_1 = \forall r.s.t.(r \rightarrow s \rightarrow t) \rightarrow (r \rightarrow s) \rightarrow r \rightarrow t.$$

Alternatively, we can type $S$ using the GR rules in such a way that the term $SIII$ is typable. To type $SIII$, we give $S$ the type

$$\sigma_2 = (\forall t.l \rightarrow t) \rightarrow (\forall l.l \rightarrow t) \rightarrow (\forall t.l \rightarrow t).$$

What is the relationship between $\sigma_1$ and $\sigma_2$? Surprisingly enough, there is a typed term $f$ with type $\sigma_1 \rightarrow \sigma_2$ which seems only to "change the type" of its argument, not alter its functionality. The term $f$ is

$$f = \lambda w \in \sigma_1. \lambda x,y \in \tau. (w \{ \tau \} \{ \tau \} \{ \tau \}) (\lambda v \in \tau. x \{ \tau \} v \{ \tau \}) (y \{ \tau \}),$$

where $\tau = \forall t.l \rightarrow t$. The erasure of this term is

$$\text{erase}(f) = \lambda w,x.y. w (\lambda v.x . v) y.$$

While $\text{erase}(f) \neq I$, it is easy to see that $\text{erase}(f)$ will $\eta$-reduce to $I$. Note that we cannot $\eta$-reduce $f$.

---

20This question was posed by Ravi Sethi.
before erasing type information. An interesting property of \( f \) is that if we apply \( f \) to a term \( S_1 \) of \( \mathcal{S} \mathcal{A} \) with type \( \sigma_1 \) and \( \text{erase}(S_1) = S \), we obtain a term that \( \beta \)-reduces to a term \( S_2 \) of \( \mathcal{S} \mathcal{A} \) with type \( \sigma_2 \) and \( \text{erase}(S_2) = S \).

We adopt the intuitive idea that if \( \text{erase}(f) \) \( \eta \)-reduces to \( I \), then \( f \) "only changes the type" of its argument. A retyping function is a closed typed term \( f \) of \( \mathcal{S} \mathcal{A} \) such that \( \text{erase}(f) \) \( \eta \)-reduces to \( I \). Retyping functions seem to be a useful a way of examining relationships between types. In addition, they characterize containments in simple inference models.

**Lemma 5.12:** The containment \( \sigma \subseteq \tau \) is valid in all simple inference models iff there is a retyping function from \( \sigma \) to \( \tau \).

The lemma is proved by observing that \( \text{GRS} \vdash \{x: \sigma\} \Rightarrow x: \tau \) iff \( \text{GR} \vdash \{x: \sigma\} \Rightarrow M: \tau \), where \( M \) \( \eta \)-reduces to \( x \).

Retyping functions can also be used to show that (eta) is not a derived rule of GR, as follows. Every typed term \( f \) of \( \mathcal{S} \mathcal{A} \) in normal form with \( \text{erase}(f) = I \) must be of the form

\[
\Pi^* \lambda x \in \forall \Gamma^* \tau \Pi S^* x \{\tau^*\}.
\]

Therefore, the only GR-provable types for \( I \) are those of the form

\[
\forall \Gamma^* (\forall \Gamma^* \tau \rightarrow \forall \delta^* (\tau \{\tau^* / \Gamma^*\})).
\]

Since the retyping function considered in the discussion of SIII above has a typing that is not of this form, it follows that (eta) is not a derived rule of GR.

### 5.6.2 Principal Type Schemes?

We say that a quantified type \( \sigma \) is **more general** than \( \tau \) if \( \sigma \subseteq \tau \) in all simple inference models. Unlike the usual notion of generality [Hindley 69, Milner 78] for type schemes without quantifiers, the most general quantified typing of a term is not unique up to renaming of variables. Among other possibilities, certain quantifier variations come into play. For example, the two types

\[
\forall s, t (t \rightarrow s \rightarrow t) \text{ and } \forall t (t \rightarrow \forall s (s \rightarrow t))
\]

are both similar to the principal type scheme \( t \rightarrow s \rightarrow t \) for \( K = \lambda x, y. x \). In fact, both are most general types for \( K \) and they denote the same set in any simple inference model.
Lemma 5.13: Let $M$ be a closed term of the form $\lambda \bar{x}^\tau.N$, where $N$ is a term without any occurrences of the binding operator $\lambda$ and no variable occurs twice in $N$. Suppose that $M$ has a Curry type scheme and let $\sigma$ be the universal closure of the principal type scheme of $M$. Then $M:\tau$ is GRS-provable iff $\sigma \subseteq \tau$ in all simple inference models.

The analogous proposition for GR is false, as shown by the two GRS-provable typings for the untyped combinator $S$. It seems that GRS is more likely to have principal type schemes than GR.

Many terms that are not of the form $\lambda \bar{x}^\tau.N$ as in the lemma may also be shown to have most general quantified GRS types. For example, the term $\lambda z.\tau(\lambda x.x)$ has most general type $\forall s((\forall t \to t) \to s) \to s$. Note that this is not the universal closure of the principal type scheme $((t \to t) \to s) \to s$.

5.6.3 Examples of Inference Models

There is a relatively straightforward method for constructing an inference model from any model $\mathcal{G} = \langle D, \cdot, e \rangle$ of the untyped lambda calculus. The construction requires a subset $Z \subseteq D$ with some special properties. A zero set $Z$ of $\mathcal{G}$ is a subset $Z \subseteq D$ such that

$\forall z \in Z. \forall d \in D. z \cdot d \in Z$.

For any nonempty zero set $Z \subseteq D$, we can construct an inference model in which the meaning of $\forall t.t$ is $Z$.

Given a zero set $Z$ for model $\mathcal{G} = \langle D, \cdot, e \rangle$, we define the inference model $\mathcal{G}_Z = \langle T, F, \Pi, \mathcal{G}, \text{subset} \rangle$ as follows. Let the set $T$ of types be the set of subsets of $D$ that contain $Z$, and define the binary operation $F$ on types using the simple semantics\(^{21}\). To see that $T$ is closed under the operation

$FAB = \{d \in D \mid d \cdot A \in B\}$,

note that for every $z \in Z$, we have $z \cdot A \subseteq Z \subseteq B$. Thus $Z \subseteq FAB \subseteq T$. There is no harm in letting the domain $[T \to T]$ of $\Pi$ be all functions from $T$ to $T$; this makes it easy to see that $[T \to T]$ is rich

---

\(^{21}\)We could also use the function-semantics, provided we choose $Z$ so that $e \cdot Z = Z$. 

131
enough to interpret all type expression. We define $\Pi$ using intersection: for any $f:\text{T}\rightarrow\text{T}$, let

$$\Pi f = \bigcap_{A \in \text{T}} f(A).$$

Since $Z \subseteq f(A)$ for all $A \in \text{T}$, we have $\Pi f \in \text{T}$. The association between elements of $\text{T}$ and subsets of $\text{D}$ is just the straightforward one. For all $A \in \text{T}$, let

$$D_A = \text{subset}(A) = A.$$

This completes the definition of the inference model $\mathcal{I}_Z$.

We now list several untyped models and zero sets. The first is trivial.

1. For any $\mathfrak{A} = \langle D, \cdot, \varepsilon \rangle$, the set $D$ is a zero set. In the zero set model $\mathfrak{A}_D$, there is only one type.

2. Let $\mathfrak{A} = \langle D, \cdot, \varepsilon \rangle$ be any ordered model of the untyped lambda calculus with least element $\bot \in D$ such that $\bot \cdot x = \bot$. The singleton $Z = \{ \bot \}$ is a zero set. The model based on this zero set is slightly reminiscent of the ideal model since $\{ \bot \}$ is the meaning of $\forall \bot$ in both models.

3. Let $\mathfrak{A} = \langle D, \cdot, \varepsilon \rangle$ be any untyped model and consider the term $YK$, where

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)), \ K = \lambda x \lambda y.x$$

as usual. The term $Y$ has the property that $Yx = x(Yx)$. In particular, $YK = K(YK)$. Consequently,

$$(YK)x = (K(YK))x = (YK)$$

and the singleton $Z = \{ \llbracket YK \rrbracket \}$ is a zero set. Since the term $YK$ has no head normal form, this term is equal to $\bot$ in $D_\infty$, $P^\omega$ and related models [Barendregt 81, Hyland 76, Wadsworth 76].

4. Let $\mathfrak{A} = \langle D, \cdot, \varepsilon \rangle$ be a term model. The elements of $D$ are equivalence classes $[M]$ of terms. For any variable $x$, consider the set

$$Z = \{[xM_1 \ldots M_n] \mid M_1, \ldots, M_n \text{ and terms, } n \geq 0 \}.$$

Since $[xM_1 \ldots M_n] = [M_1 \ldots M_n N]$, $Z$ is a zero set. In general, this $Z$ is an infinite proper subset of $D$.

It is worth noting that while the entire domain $D$ is a type in any zero-set model, this is not true of inference models in general. For example, there are inference models in which all types are contained in the interior of the model.
5.6.4 Comments on the Function-Semantics

An interesting open problem is to study inference models that satisfy

\[(F_{\text{function}}) \mathcal{D}_{F\text{a}} = \{d \in \mathcal{D} \mid \forall e \in \mathcal{D}_{a}, d \cdot c \in \mathcal{D}_{b}\} \cap \mathcal{F},\]

where \(\mathcal{F}\) is the range of \(\varepsilon\). As noted in Section 3, the significance of \(\mathcal{F}\) is that every function from \(\mathcal{D}\) to \(\mathcal{D}\) that is represented by some element of \(\mathcal{D}\) has a unique representative in \(\mathcal{F}\). Intuitively, then, \(\mathcal{F}\) is the set of functions in the model.

In [Hindley 83b], the Curry typing rules are shown to be complete for the function-semantics of \(\rightarrow\). There is an interesting difficulty in applying the ideas of [Hindley 83b] to typing with quantifiers. Intuitively, the difficulty has to do with the "arity" of functions. Consider the type \(\mathcal{D}_{F\text{a}(F\text{bc})}\) of functions from \(\mathcal{D}_{a}\) to \(\mathcal{D}_{b}\) to \(\mathcal{D}_{c}\). If \(\mathcal{F}_{\text{function}}\) is used, then since \(\mathcal{D}_{F\text{bc}}\) a subset of \(\mathcal{F}\), the elements of \(\mathcal{D}_{F\text{a}(F\text{bc})}\) must map \(\mathcal{D}\) to \(\mathcal{F}\). Elements of \(\mathcal{D}_{F\text{a}(F\text{bc})}\) must be "two-argument" functions. If \(\sigma\) is a type expression of the form

\[\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_{n+1}\]

with \(\sigma_{n+1}\) a type variable, then elements of the set associated with \(\sigma\) must be \(n\)-ary functions.

To be more precise, we define sets of \(n\)-ary functions, for each \(n\). The set \(\mathcal{F}_n\) of \(n\)-ary functions is defined inductively by \(\mathcal{F}_0 = \mathcal{D}\), \(\mathcal{F}_1 = \mathcal{F}\) and \(\mathcal{F}_{n+1} = \mathcal{D} \rightarrow_{\text{function}} \mathcal{F}_n\). Note that the set \(\mathcal{F}_n\) is the range of the function

\[\varepsilon_n = [\lambda z, x_1, \ldots, x_n, z x_1 \ldots x_n].\]

Thus each \(\mathcal{F}_n\) is the range of a lambda-definable function. (The notation \(\varepsilon_n\) is consistent with [Meyer 82]; see also \(\mathcal{B}^n(1)\) on page 423 of [Scott 80]).

Now, given a type \(\sigma\) without quantifiers, we can see which \(\mathcal{F}_n\) the elements of \(\sigma\) must belong to. All we need to do is count \(\rightarrow\)'s in the right way. In addition, we can use the \(\varepsilon\)'s to find an term model environment mapping a variable \(x\) into \(\mathcal{F}_n\), a necessary part of the completeness proof of [Hindley 83b]. However, we are stuck when we come to types like \(\forall \mathcal{L} \cdot\). What is the arity of an element of this type? Since

\[\forall \mathcal{L} \cdot \subseteq \sigma\]

for any \(\sigma\), elements of this type must have arbitrarily high arity. So we are led to consider some set \(\mathcal{F}_\infty\). But there does not seem to be a lambda-definable function \(\varepsilon_\infty\) that is GR-typable and
whose range is a reasonable set of "infinite-arity" functions.

5.6.5 Open Problems

There are many interesting, unanswered questions about type inference for quantified types. Since many useful GR-typable terms are not typable in the ML system [Gordon, et. al. 79, Milner 78], it would be useful to extend the ML type checker to an algorithm based on GR. However, we do not know whether GR, or any significant fragment of GR extending ML, is decidable.\footnote{R. Statman, in private communication (1983), has claimed that a system similar to GR is undecidable.} Thus a significant open problem is

Given a closed term $M$ and type $\sigma$, is it decidable whether $\text{GR} \vdash M: \sigma$ or $\text{GRS} \vdash M: \sigma$?

A related open problem is to determine whether all GR- or GRS-typable terms have most general quantified types? More specifically, a principal type for a term $M$ is a type $\sigma$ such that $M: \tau$ iff $\tau$ is may be obtained from $\sigma$ by some reasonably straightforward method. Thus determining whether there exist principal types involves exploring ways of deriving one type of a term from another. The containment rules provide one approach to characterizing the set of types of a term, but there are other possibilities.

We have proved completeness theorems for arbitrary inference models and the more specialized simple inference models. It remains to prove a completeness theorem for the function semantics. Some aspects of this problem are discussed in the preceding subsection of the paper.

The inference rules presented in this paper are complete for deducing the valid typing statements. A stronger theorem is to prove that given any set of typing statements, the rules are sufficient to derive all semantics consequences of this set of statements. Work is currently in progress to extend the systems of this paper to deductively complete systems.

Finally, a interesting line of inquiry is to explore further connections between inference models and models of the typed language $SA$. Perhaps further development of techniques for
constructing inference models, along with a better understanding of the relationship between
inference models and models of the second-order lambda calculus will lead to methods for
constructing second-order models.
Chapter Six

Directions for Further Investigation

A number of technical problems related to the second-order lambda calculus and type inference are described in the concluding sections of Chapters 2 through 5. The most significant of these problems are to develop elementary models of second-order lambda calculus (see Chapter 3) and to determine whether the semantically incomplete typing theory based on the typing rules for second-order lambda calculus is decidable (see Chapter 5). In addition to the specific problems listed in the thesis, there are important questions about types in programming languages that neither the second-order lambda calculus nor the type inference systems are designed to address. A few of these are presented below.

*Type Errors*

A simple-minded view of type checking is that "type checking prevents type errors." None of the formal systems used in the thesis seem to explain adequately, in a precise or semantic sense, what a type error really is. The type checking rules of the second-order lambda calculus and the type inference systems seem to guarantee the absence of type errors. However, in order to state this precisely, we need a precise characterization of type error. In particular, we would like to be able to distinguish type errors from other sorts of error conditions.

Milner has proposed a semantic model with *type-error* (or *wrong*) as an element of the domain [Milner 78]. This semantic model provides a partial explanation for a set of typing rules. Namely, whenever a term can be typed using his rules, the meaning of the term is not *type-error*. There are two lines of research that may make Milner's account of type errors more convincing. The first is by characterizing the valid typings, even though these are certain to be undecidable. This should lead to a more thorough appraisal of the properties of the model. If there are counterintuitive properties, then perhaps a variant of Milner's model could be developed. The second problem is to find an equivalence between the denotational notion of

136
type error and an operational definition of run-time type error. Perhaps by considering run-time errors in different operational semantics (cf. the reduction strategies in [Plotkin 75]) we can refine our understanding of type errors.

Principles for Programming Language Design

Another general area for further research is to formulate semantic principles that can be used to evaluate type constraints. While type checking rules are often designed to encourage certain programming styles, semantic analysis may lead to better understanding of various rules.

Reynolds and others [Donahue 79, Haynes 84, Reynolds 74, Reynolds 83] have studied the notion of representation independence. Intuitively, the type checking rules of many programming languages are designed to prevent programmers from taking advantage of specific representations of types [Reynolds 74]. For example, type checking should guarantee that any program that is "correct" when compiled onto a two s-complement machine will also be correct when compiled onto a one s-complement machine. It should not make any difference how numbers are represented as long as all the basic operations work correctly. The formal representation independence property is intended to describe this property of typed languages. Unfortunately, the technical definitions of representation independence seem to be difficult to summarize simply. Thus one important research problem is to clarify the property of representation independence.

It seems worthwhile to formulate and study other principles that characterize well-typed languages. For example, there is a well-known "type insecurity" in Pascal variant records. What well-justified semantic principles does this "bug" violate? One candidate is the principle of representation independence. Another possibility is a principle based on the formulas-as-type analogy discussed in Section 2.3. The treatment of variant records in Pascal is similar to the treatment of sums in ML if we ignore the fact that in ML, outleft and outright may raise run-time exceptions. In the formulas-as-types analogy, the types of outleft and outright of ML are not valid formulas. Specifically, the type of outleft is $\langle s \vee t \rangle \rightarrow s$. Perhaps the "type insecurity" of Pascal can be condemned as violation of a principal related to valid logical formulas. However, more work is needed to see if such a principle can be stated precisely and justified.

137
A Theory of Type Expressibility

Some programming languages are clearly more convenient to program in than others. In the field of program schematology, various techniques have been developed to compare the "expressive power" of constructs like iteration and recursion (cf. [Greibach 75]). Perhaps we are now in a position to study the expressive power of in various type disciplines.

One theory of the expressiveness of type disciplines is advanced in [Fortune, et. al. 83]. Basically, these authors study the number-theoretic functions that may be defined in typed languages without recursion. A related theory of expressiveness is to examine the sorts of types that are definable in various languages. For example, in Section 3.8 we show that existential types are definable in the second-order lambda calculus without existential types. This result may be interpreted as implying that the language with existential types added is no more powerful than the language without. The second-order lambda calculus provides examples of product types, polymorphic types, existential types and other sorts of types that may be defined in a "categorical" way. For example, a product of types $\sigma$ and $\tau$ in second-order lambda calculus is a type $\rho$ such that there exist a pairing function from $\sigma$ and $\tau$ into $\rho$ and projection functions from $\rho$ to $\sigma$ and $\tau$ with the appropriate properties. Thus we can ask whether a language "has" product types by asking whether, for any types $\sigma$ and $\tau$, there is another type $\rho$ with pairing and projection functions that can be defined in the language. Similarly, based on the second-order lambda calculus characterization of abstract data types, we can ask whether abstract data types can be defined in a language.

The study of definable functions has some implications on the study of definable types: since some functions definable in second-order lambda calculus may not be defined in the simple typed lambda calculus [Fortune, et. al. 83], we can see that second-order types cannot be defined in the simply-typed lambda calculus. However, the two theories are different. For example, surjective pairing cannot be defined in the untyped lambda calculus [Barendregt 81], even though all partial recursive functions can be defined in untyped lambda calculus.

Relationships Between Views of Types
We have not fully explored the possible semantic connections between second-order lambda calculus and type inference based on second-order lambda calculus. Some discussion of this topic appears in Section 3.3 of Chapter 5, and in the conclusion of that chapter. The typing rules of second-order lambda calculus are reflected in the semantics of second-order lambda calculus and in the inference models of Chapter 5. However, it is not clear what kind of general connections there might be between the two kinds of structures.

One way to explore the difference between the second-order lambda calculus and type inference might be to find structures that could be considered models of both systems. There are models of second-order lambda calculus built from models of untyped lambda calculus (cf. [Bruce and Meyer 84, Haynes 84, McCracken 79]). In these second-order models, all types are subsets of the untyped model. There are some immediate questions about these models. Can this family of subsets be used to construct an inference model? What relationship, if any, exists between the meaning of a second-order term $M$ in one of these models and the meaning of the untyped term obtained from $M$ by erasing type information? Perhaps a semantic understanding of the effect of erasing type information will lead to a better understanding of type inference and connections between type inference and typed languages.
References

GPO 008-000-0254-8, 1980.

[Aho, Hopcroft and Ullman 83]
Aho, A.V., Hopcroft, J.E. and Ullman, J.D.
*Data Structures and Algorithms.*
Addison-Wesley, 1983.

[Arbib and Manes 75]
Arbib, M.A. and E.G. Manes.
*Arrows, Structures, and Functors: The Categorical Imperative.*

[Barendregt 81] Barendregt, H.P.
*The Lambda Calculus: Its Syntax and Semantics.*
North Holland, 1981.

[Barendregt and Rezus 84]
Barendregt, H. and Rezus, A.
Semantics for Classical AUTOMATH and Related Systems.
To appear.

[Barendregt, Coppo and Dezani 83]
Barendregt, H., Coppo, M. and Dezani-Ciancaglini, M.
A Filter Lambda Model and the Completeness of Type Assignment.
pages 931-940.

[Bates and Constable 81]
*The definition of µprl.*
Technical Report TR 82-492, Cornell Univ., Dept. of Computer Science,
October, 1981.
[Bruce and Longo 84]
Bruce, K. and Longo, G.
Definable Isomorphisms and Domain Equations in the Second-Order Lambda Calculus.
Unpublished Manuscript.
1984

[Bruce and Meyer 84]
Bruce, K. and Meyer, A.
A Completeness Theorem for Second-Order Polymorphic Lambda Calculus.
In Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France),
pages 131-144. 1984.

[Burstall and Goguen 77]
Burstall, R.M. and J. Goguen.
Putting theories together to make specifications.
In Fifth International Joint Conf. on Artificial Intelligence. 1977.
pages 1045-1058.

[Burstall and Goguen 81]
Burstall, R.M. and J. Goguen.
An Informal Introduction to Specification Using CLEAR.
The Correctness Problem in Computer Science.
pages 185-213.

[Burstall and Lampson 84]
Burstall, R. and Lampson, B.
A Kernel Language for Abstract Data Types and Modules.

[Constable 80] Constable, Robert L.
Programs and types.
In 21st IEEE Symp. on Foundations of Computer Science, pages 118-128.
IEEE, 1980.

[Coquand and Huet 84]
Coquand, T. and Huet, G.
A Theory of Constructions.
Paper does not appear in proceedings.

141
[Curry and Feys 58]
Curry, H.B and Feys, R.
*Combinatory Logic I.*

[Damas and Milner 82]
Damas, L. and Milner, R.
Principal Type Schemes for Functional Programs.

[DeBruijn 80]
De Bruijn, N.G.
A survey of the project AUTOMATH.
In Seldin, J.P. and J.R. Hindley (editors),

[Demers and Donahue 80a]
Demers, A.J. and J.E. Donahue.
Data types, parameters and type checking.
In 7th ACM Symp. on Principles of Programming Languages, pages 12-23.
ACM, 1980.

[Demers and Donahue 80b]
Demers, A.J. and J.E. Donahue.
‘Type-completeness’ as a language principle.
In 7th ACM Symp. on Principles of Programming Languages, pages 234-244.
ACM, 1980.

[Demers et. al. 78]
Demers, A., J. Donahue and G. Skinner.
Data types as values: polymorphism, type-checking, encapsulation.
In 5th ACM Symp. on Principles of Programming Languages, pages 23-30.

[Donahue 79]
Donahue, J.
On the semantics of data type.

[Dwork, Kanellakis and Mitchell 84]
Dwork, C., Kanellakis, P. and Mitchell, J.C.
On the Sequential Nature of Unification.
[Fitting 69] Fitting, M.C.
*Intuitionistic Logic, Model Theory and Forcing.*
North Holland, 1969.

[Fortune, et. al. 83] Fortune, S., Leivant, D. and O'Donnel, M.
The Expressiveness of Simple and Second Order Type Structures.
*JACM* 30(1), 1983.
pages 151-185.

[Friedman 75] Friedman, H.
Equality Between Functionals.
*Logic Colloquium.*
Springer-Verlag, 1975, pages 22-37.

[Girard 71] Girard, J.-Y.
Une extension de l'interprétation de Gödel à l'analyse, et son application à
l'élimination des coupures dans l'analyse et la théorie des types.
In Fenstad, J.E. (editor), *2nd Scandinavian Logic Symp.* pages 63-92.

*The Denotational Description of Programming Languages.*
Springer-Verlag, 1979.

*Lecture Notes in Computer Science. Volume 78: Edinburgh LCF.*
Springer-Verlag, 1979.

[Gratzer 68] Gratzer, G.
*Universal Algebra.*

[Greibach 75] Greibach, S.A.
*Lecture Notes in Computer Science. Volume 36: Theory of Program
Structures: Schemes, Semantics, Verification.*
Springer-Verlag, 1975.

Abstract data types and software validation.
pages 1048-1064.
[Haynes 84] Haynes, C.T.
A Theory of Data Type Representation Independence.

[Henkin 50] Henkin, L.
Completeness in the Theory of Types.
pages 81-91.

[Herrtich and Strecker 73]
Herrlich, H. and G.E. Strecker.
*Category Theory.*

[Hindley 69] Hindley, R.
The Principal Type-Scheme of an Object in Combinatory Logic.
*Trans. AMS* 146, 1969.
pages 29-60.

[Hindley 83a] Hindley, R.
The Completeness Theorem for Typing Lambda Terms.
pages 1-17.

[Hindley 83b] Hindley, R.
Curry's Type Rules Are Complete with Respect to the F-Semantics Too.
pages 127-133.

[Hook 84] Hook, J.G.
Understanding Russell -- A First Attempt.

[Howard 80] Howard, W.
The formulas-as-types notion of construction.

[Hyland 76] Hyland, J.M.E.
A Syntactic Characterization of the Equality in Some Models of the Lambda Calculus.
pages 361-370.
[Kapur 80] Kapur, D.
Towards a Theory for Abstract Data Types.

[Lambek 80] Lambek, J.
From Lambda Calculus to Cartesian Closed Categories.
In Seldin, J.P. and J.R. Hindley (editors), To H.B. Curry: Essays on
Combinatory Logic, Lambda Calculus and Formalism, pages 375-402.

[Landin 65] Landin, P.J.
A Correspondence Between ALGOL 60 and Church's Lambda Notation.
CACM 8, 1965.
pages 89-101; 158-165.

[Landin 66] Landin, P.J.
The Next 700 Programming Languages.
CACM 9, 1966.
pages 157-166.

[Lauchli 65] Lauchli, H.
Intuitionistic propositional calculus and definably non-empty terms.
pages 263.

[Lauchli 70] Lauchli, H.
An abstract notion of realizability for which intuitionistic predicte calculus is complete.
pages 227-234.

[Leivant 81] Leivant, D.
The Complexity of Parameter Passing in Polymorphic Procedures.
pages 38-45.

[Leivant 83a] Leivant, D.
Polymorphic Type Inference.
[Leivant 83b] Leivant, D.
Structural Semantics for Polymorphic Types.

[Liskov et. al. 77] Liskov, B., A. Snyder, R. Atkinson and C. Schaffert.
Abstraction mechanisms in CLU.

[Liskov et. al. 81] Liskov, B. et. al.
Springer-Verlag, 1981.

[Mac Lane 71] Mac Lane, S.
Graduate Texts in Mathematics. Volume 5: Categories for the Working Mathematician.
Springer-Verlag, 1971.

[MacQueen] MacQueen, D.
Private Communication.
1983.

[MacQueen 84] MacQueen, D.B.
Modules for Standard ML (Draft).
Polymorphism (privately circulated), 1984.

[MacQueen and Sethi 82] MacQueen, D. and Sethi, R.
A Semantic Model of Types for Applicative Languages.

[MacQueen, Plotkin and Sethi 84] MacQueen, D., Plotkin, G and Sethi, R.
An Ideal Model for Polymorphic Types.

[Martin-Löf 75] Martin-Löf, P.
About models for intuitionistic type theories and the notion of definitional equality.
Constructive mathematics and computer programming.
1979.
Paper presented at 6th International Congress for Logic, Methodology and

[McCracken 79] McCracken, N.
An Investigation of a Programming Language with a Polymorphic Type
Structure.

[McCracken 84] McCracken, N.
The Typechecking of Programs with Implicit Type Structure.

What Is A Model of the Lambda Calculus?
Information and Control 52(1), 1982.
pages 87-122.

[Milner 78] Milner, R.
A Theory of Type Polymorphism in Programming.
JCSS 17, 1978.
pages 348-375.

[Mitchell 84a] Mitchell, J.C.
1984.
Chapter 3 of this thesis.

[Mitchell 84b] Mitchell, J.C.
Type Inference and Type Containment.
Chapter 5 of this thesis.

[Mitchell 84c] Mitchell, J.C.
Coercion and Type Inference (Summary).
In Proc. 11-th ACM Symp. on Principles of Programming Languages, pages
Chapter 4 of this thesis.

[Mitchell, James G. et. al. 79]
Mitchell, James G., Mayberry, W. and Sweet, R.
Mesa Language Manual.

147
[Monk 76] Monk, J. D.
Springer-Verlag, 1976.

[Morris 73] Morris, James H.
Types are not sets.
In 1st ACM Symp. on Principles of Programming Languages, pages 120-124.

[ODonnell 79] O'Donnell, M.
A practical programming theorem which is independent of Peano arithmetic.

[Paterson and Wegman 78]
Paterson, M.S. and Wegman, M.N.
Linear Unification.
pages 158-167.

[Plotkin 75] Plotkin, G.D.
Call-by-name, Call-by-value and the Lambda Calculus.
pages 125-159.

[Prawitz 65] Prawitz, D.
Natural Deduction.

[Prawitz 71] Prawitz, D.
Ideas and Results in Proof Theory.
In Fenstad, J.E. (editor), 2nd Scandinavian Logic Symp., pages 235-308. North-
Holland, 1971.

[Reynolds 74] Reynolds, J.C.
Towards a Theory of Type Structure.

[Reynolds 80] Reynolds, J.C.
Using Category Theory to Design Implicit Conversions and Generic
Operators.
Semantics-Directed Compiler Generation.
Springer-Verlag Lecture Notes in Computer Science, Vol. 94, 1980, pages
211-2580.
[Reynolds 81a] Reynolds, J.C.  
The Essence of ALGOL.  

[Reynolds 81b] Reynolds, J.C.  
*The Craft of Programming*.  
Prentice Hall, 1981.

[Reynolds 83] Reynolds, J.C.  
Types, Abstraction, and Parametric Polymorphism.  
In *IFIP Congress*, pages . , 1983.

[Reynolds 84] Reynolds, J.C.  
Polymorphism is not Set-Theoretic.  

[Robinson 65] Robinson, J.A.  
A Machine Oriented Logic Based on the Resolution Principle.  
*JACM* 12(1), 1965.  
pages 23-41.

[Russell 03] Russell, B.  
*The Principles of Mathematics*.  
Cambridge, 1903.

[Scott 76] Scott, D.  
Data Types as Lattices.  
pages 522-587.

[Scott 80] Scott, D.S.  
Relating theories of the lambda calculus.  

[Semantics of Data Types 84]  
Kahn, MacQueen, Plotkin (eds.).  
*Lecture Notes in Computer Science. Volume 173: Semantics of Data Types*.  

[Shaw 81] Shaw, M. (ed.).  
*ALPHARD: Form and Content*.  
Springer-Verlag, 1981.
[Sherman et. al. 82]
Sherman, M., Higsen, A. and Rosenberg, J.
A Methodology for programming abstract data types in Ada.
pages 66-75.

[Statman 79a]
Statman, R.
Intuitionistic Propositional Logic is Polynomial-Space Complete.
Theoretical Computer Science 9, 1979.
pages 67-72.

[Statman 79b]
Statman, R.
The Typed Lambda Calculus is Not Elementary Recursive.
Theoretical Computer Science 9, 1979.
pages 73-81.

[Statman 84]
Statman, R.
Equality Between Functionals, Revisited.
Friedman Volume, to appear.
1984

[Stenlund 72]
Stenlund, S.
Combinators, \(\lambda\)-terms and Proof Theory.

[Stoy 77]
Stoy, J.E.
Denotational Semantics: The Scott-Strachey Approach to Programming
Language Theory.

[Tanenbaum 76]
Tanenbaum, A.S.
A Tutorial on Algol 68.
pages 155-190.

[Trakhtenbrot, Halpern and Meyer 83]
From Denotational to Operational and Axiomatic Semantics for ALGOL-like
Languages: An Overview.
In Logics of Programs. Proceedings, Lecture Notes in Computer Science,
Springer. 1983.

[Troelstra 73]
Troelstra, A.S.
Lecture Notes in Mathematics. Volume 344: Mathematical Investigation of
Intuitionistic Arithmetic and Analysis.
[Wadsworth 76] Wadsworth, C.
The Relation Between Computational and Denotational Properties for Scott's
$D_\omega$ Models.
pages 488-521.

An introduction to the construction and verification of Alphard programs.
pages 253-264.