INFORMATION AND COMPUTATION 85, 76–134 (1990)

The Semantics of Second-Order Lambda Calculus

KIM B. BRUCE*

Department of Computer Science, Williams College, Williamstown, Massachusetts 01267

ALBERT R. MEYER†

Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

AND

JOHN C. MITCHELL‡

Department of Computer Science, Stanford University, Stanford, California 94306

In the second-order (polymorphic) typed lambda calculus, lambda abstraction over type variables leads to terms denoting polymorphic functions. Straightforward cardinality considerations show that a naive set-theoretic interpretation of the calculus is impossible. We give two definitions of semantic models for this language and prove them equivalent. Our syntactical "environment model" definition and a more algebraic "combinatory model" definition for the polymorphic calculus correspond to analogous model definitions for untyped lambda calculus. Soundness and completeness theorems are proved using the environment model definition. We verify that some specific interpretations of the calculus proposed in the literature indeed yield models in our sense.

1. INTRODUCTION

The second-order lambda calculus, formulated independently by Girard (1972) and Reynolds (1974), is an extension of the usual typed lambda calculus. Like other kinds of lambda calculus, the ordinary parameter-binding mechanism of this language corresponds closely to parameter

* Partially supported by NSF Grants DCR-8402700, DCR-8603890, and a grant from Williams College.

† Partially supported by NSF Grant MCS80-10707.

‡ Partially supported by an NSF Presidential Young Investigator Award.

0890-5401/90 $3.00

binding in many programming languages (cf. Landin, 1965; Reynolds, 1981; Trakhtenbrot, Halpern, and Meyer, 1983). The particular type structure of the second-order system corresponds to the type structures of programming languages with polymorphism and data abstraction (Fortune *et al.*, 1983, Mitchell and Plotkin, 1988). Like Ada *generics* and parameterized modules in CLU (U.S. Department of Defense, 1980, Liskov *et al.* 1981), polymorphic functions in the second-order lambda calculus are formed by explicit lambda abstraction over types.[1] Since the calculus is composed of only a few constructs, second-order lambda calculus is a useful tool for studying and giving semantics to programming languages where types appear explicitly as parameters. In this paper, we examine the mathematical semantics of second-order lambda calculus, proving a completeness theorem and providing two characterizations of models.

The syntax of second-order lambda calculus, which is defined precisely in Sections 2 and 3, may be separated into three parts. The first is the set of second-order lambda expressions, or *terms*. Intuitively, terms are the "ordnary expressions" that describe computable functions and results of computation. The second syntactic class contains the type expressions. Expressions of the third class, the *kinds*, are used to describe the functionality of subexpressions of type expressions. For example, if t is any type, then the term

$$\lambda x\colon t.x$$

denotes the identity function on type t. The type of this term is $t \to t$, the type of functions from t to t. Given any argument y of type t, the value of the function application $(\lambda x\colon t.x)\, y = y$ may be computed by substituting y for the bound variable (formal parameter) x in the body of the term. Second-order lambda calculus allows us to lambda abstract over types, which produces polymorphic functions. Since we made no assumptions about the type t in writing $\lambda x\colon t.x$, we may regard t as a free type variable. (We will use $r, s, t, \ldots$ for type variables and $x, y, z, \ldots$ for ordinary variables.) The polymorphic identity function

$$I ::= \lambda t.\lambda x\colon t.x$$

is formed by lambda abstracting the type variable t. We may apply (or "instantiate") the polymorphic identity I to any type σ, computing the value of the application $I\sigma$ by substituting σ for t in the body $\lambda x\colon t.x$. Thus

$$I\sigma = \lambda x\colon \sigma.x.$$

[1] An alternative to second-order lambda calculus is to introduce polymorphism "implicitly" by assignment of more than one type to a single expression. The reader is referred to Barendregt, Coppo, and Dezani (1983), Leivant (1983a), Milner (1978), MacQueen, Plotkin, and Sethi (1986), Mitchell (1984a), Mitchell (1988) for further discussion of this alternative.

The polymorphic function I has a polymorphic type. Intuitively, the domain of I is the collection of all types, and the range of I is the union of all types of the form $t \to t$. We can express the type of I more specifically using the mapping $\lambda t.t \to t$ from types to types. Given an argument σ, we can compute the type of $I\sigma$ by applying the function $\lambda t.t \to t$ to σ. Thus we expect the type of the polymorphic identity to be derived from the function $\lambda t.t \to t$ in some way. We use the operator $\forall$ to produce a type from any function mapping types to types, and write $\forall(\lambda t.t \to t)$ for the type of the polymorphic identity I. In general, if M has type $\forall(\lambda t.\tau)$ then the type of the application $M\sigma$ is $(\lambda t.\tau)\sigma$. We usually abbreviate $\forall(\lambda t.t \to t)$ to $\forall t.t \to t$. The difference between $\lambda t.t \to t$ and $\forall t.t \to t$ is that $\lambda t.t \to t$ is a function from types to types, while $\forall t.t \to t$ is a type.

In second-order lambda calculus, each term has a type, and types are written using higher-order symbols (type constructors), $\to$ and $\forall$. The function-type constructor $\to$ is an infix binary operator on types. The polymorphic-type constructor $\forall$ takes a function from types to types and produces a type. If we wish to expand the language to allow product types (ordered pairs or records), sum types, and so on, then we will need to add new type constructors. Anticipating these and other extensions to the language, we will define second-order lambda terms with respect to any set of type constructors. Therefore, in addition to terms and type expressions, we will also have a general class of constructor expressions. To keep the syntax of constructor expressions straight, we use "kinds," which were called "orders" in Girard (1972). Kinds were introduced independently in McCracken (1979) and used subsequently in MacQueen and Sethi (1982), MacQueen, Plotkin, and Sethi (1986). Essentially, kinds are the "types" of things that appear in type expressions.

Subexpressions of type expressions may denote types, functions from types to types, functions from type functions to types, and so on. We will use T to denote the *kind* consisting of all types and $\kappa_1 \Rightarrow \kappa_2$ for the kind consisting of functions from kind κ_1 to κ_2. Thus we regard a function like $\lambda t.t \to t$ from types to types as a constructor expression of kind $T \Rightarrow T$. Similarly the constructor expression "$\to$" is of kind $T \Rightarrow (T \Rightarrow T)$ and $\forall$ has kind $(T \Rightarrow T) \Rightarrow T$. In effect, we use the ordinary typed lambda calculus in the syntax of type expressions. However, to reduce confusion between types and kinds, we use $\Rightarrow$ instead of $\to$ and call the types of this language kinds. Thus we have a hierarchy from lambda expressions to constructor expressions (which include the type expressions) to kind expressions. Lambda expressions have types and constructor expressions have kinds. While our main focus is on terms and their types, kinds play an important role in organizing the subexpressions of type expressions.

A number of proof-theoretic properties of second-order lambda calculus have been studied. The class of functions that can be represented in the

calculus, the normalization theorem, and other proof theoretic results are described in Girard (1972), Statman (1981), Fortune *et al.* (1983). However, the semantics of second-order lambda calculus is not entirely straightforward. The reason for this is illustrated by the fact that terms may be applied to their own types. For example, the polymorphic identity I can be applied to its own type $\forall t.t \to t$. If we think of a type as the set of all objects having that type, we are led to a contradiction with classical mathematics: the polymorphic identity I must denote a function whose domain contains the set $\forall t.t \to t$ and, at the same time, the set $\forall t.t \to t$ must contain I. We will see that we can make mathematical sense of second-order lambda calculus, but we must depart from the naive approach of letting λ-terms denote functions and types sets of functions.

Although general descriptions of models (essentially based on terms over arbitrary sets of constant symbols) were already given in (Girard, 1972; Martin-Löf, 1975; Stenlund, 1972), and a semantic model based on recursive function application was presented in (Girard, 1972), this was not known to many American computer scientists studying the system. Reynolds (1974) attempted to construct a domain-theoretic model for the language but ran into difficulties and later demonstrated that no model in which the function-space constructor $\to$ behaves set-theoretically is possible (Reynolds, 1984). Donahue (1979) attempted to construct a model using retracts over complete lattices, but ran afoul at a rather technical step where a retract of all retracts seemed to be necessary. McCracken (1979), building on ideas from Scott (1976) and working independently of Donahue, produced the first correct domain-theoretic model of the second-order polymorphic lambda calculus. This model was constructed from Scott's universal domain $\mathscr{P}\omega$, using closures (a special kind of retract) to represent types. In $\mathscr{P}\omega$, the set of all closures is the range of a closure, so that the problem encountered by Donahue may be avoided. McCracken (1984a), following a suggestion of Scott (1980b), has also shown that finitary retracts over certain finitary complete partial orders can be used to represent types. Bruce and Longo (Amadio *et al.*, 1986), again using ideas appearing in several papers by Scott, have also constructed a model using finitary projections over complete partial orders. In a somewhat different vein, Leivant (1983b) suggested a framework for the "structural semantics" of the second-order polymorphic lambda calculus. Since the types are the closed type expressions, Leivant's general model definition is an amalgam of a mathematical model for the elements and a syntactic model for the types.

We will give two definitions of model, the environment model and the more algebraic combinatory model. Our environment model definition first appeared in Bruce and Meyer (1984) and the combinatory model definition in Mitchell (1984b). In support of our definitions, we will prove soundness

and completeness theorems and show that the two definitions of model are equivalent. (For simplicity, we assume in the soundness and completeness theorems that no type is empty.) We also indicate how our general notion of semantics relates to known examples of models such as Girard's HEO_2 based on recursive function application and the domain-theoretic models of McCracken and others. As mentioned earlier, Girard, Stenlund, Martin-Löf, and Leivant have also proposed general model definitions (Girard, 1972; Stenlund, 1972; Martin-Löf, 1975; Leivant, 1983b), and polymorphic combinators were discussed in Stenlund (1972). Our semantics encompasses the earlier general descriptions of models (which use type expressions and/or terms over arbitrary sets of constants) and was originally formulated without knowledge of the earlier work of Girard, Stenlund, or Martin-Löf.

In Section 2 we describe the syntax and typing rules and in Section 3 we present the axiom system for proving equations between terms. The relationship between the particular calculus we have chosen to use and other similar systems presented in the literature is discussed at the end of Section 2. In Section 4, the definition of environment model and the semantics of second-order lambda terms and constructor expressions are presented. We prove soundness and completeness theorems in Section 5. Section 6 introduces combinatory algebras and models and establishes the equivalence of combinatory and environment model definitions. We explain how the models of Girard, McCracken, and others fit our framework in Section 7. In the concluding Section 8, we discuss some extensions of this work as well as some open problems. It is worth repeating that we assume, throughout the paper, that all types are nonempty. Empty types introduce a number of complications which are considered in (Meyer, Mitchell, Moggi, and Statman, 1987; Mitchell and Moggi, 1987).

2. Syntax

2.1. *Constructors and Kinds*

As described in the Introduction, every term has a type and every subexpression of a type expression has a kind. The subexpressions of type expressions, which may be type expressions or operators like $\rightarrow$ and $\forall$, will be called constructors. We will define the sets of kinds and constructor expressions before introducing the syntax and type checking rules for terms.

We will use the constant T to denote the kind consisting of all types. The set of kind expressions is given by

$$\kappa ::= T \mid \kappa_1 \Rightarrow \kappa_2.$$

Intuitively, the kind $\kappa_1 \Rightarrow \kappa_2$ is the kind of functions from κ_1 to κ_2. For example, functions from types to types have kind $T \Rightarrow T$. We define the set of constructor expressions, beginning with a set of constructor constants. Let $\mathscr{C}_{\text{cst}}$ be a set of constant symbols c^{κ}, each with a specified kind (which we write as a superscript when necessary) and let $\mathscr{V}_{\text{cst}}$ be a set of variables v^{κ}, each with a specified kind. We assume we have infinitely many variables of each kind. The constructor expressions over $\mathscr{C}_{\text{cst}}$ and $\mathscr{V}_{\text{cst}}$, and their kinds, are defined by the derivation system

$$c^{\kappa}\colon \kappa,\ v^{\kappa}\colon \kappa$$

$$\frac{\mu\colon \kappa_1 \Rightarrow \kappa_2,\ \nu\colon \kappa_1}{\mu\nu\colon \kappa_2}$$

$$\frac{\mu\colon \kappa_2}{\lambda v^{\kappa_1}.\mu\colon \kappa_1 \Rightarrow \kappa_2}.$$

For example, $(\lambda v^T.v^T)c^T$ is a constructor expression with kind T. Free and bound variables are defined as usual. Substitution $\{\mu/u\}\nu$ of μ for free occurrences of u in ν has the usual inductive definition, including renaming bound variables in ν to avoid capture of free variables in μ.

A subset of the constructor expressions are the type expressions, the constructor expressions of kind T. Since we will often be concerned with type expressions rather than arbitrary constructor expressions, it will be useful to distinguish them by notational conventions. We adopt the conventions that

$r, s, t, \ldots$ stand for arbitrary type variables

$\rho, \sigma, \tau, \ldots$ stand for arbitrary type expressions.

As in the definition above, we will generally use μ and ν for constructor expressions. We include the usual second-order types in the language by assuming that $\mathscr{C}_{\text{cst}}$ contains the function-type constructor constant

$$\rightarrow\colon T \Rightarrow (T \Rightarrow T)$$

and the polymorphic-type constructor constant

$$\forall\colon (T \Rightarrow T) \Rightarrow T.$$

As usual, we write $\rightarrow$ as an infix operator, as in the type expression $\sigma \rightarrow \tau$, and write $\forall t.\sigma$ for $\forall(\lambda t.\sigma)$. In extensions of the basic language with direct products or disjoint sums, for example, we would include additional constants $\times, +\colon T \Rightarrow (T \Rightarrow T)$ in $\mathscr{C}_{\text{cst}}$. In this paper, we will be concerned primarily with $\rightarrow$ and $\forall$.

Since we have a "kinded" lambda calculus, there are many nontrivial equations between types and constructors. While it would be more general to allow non-logical axioms for constructor equality, this would complicate the syntax nontrivially, as discussed briefly in the conclusion of the paper. For simplicity, we will only consider the "pure" constructor equations that follow from the logical axioms below. The axioms and inference rules for constructors are essentially the familiar rules of the ordinary simple typed lambda calculus.

Constructor Axioms.

(α_κ) $\lambda v^\kappa.\mu = \lambda u^\kappa.\{u^\kappa/v^\kappa\}\mu$, u^κ not free in μ

(β_κ) $(\lambda v^\kappa.\mu)\nu = \{\nu/v^\kappa\}\mu$

(η_κ) $\lambda v^\kappa.(\mu v^\kappa) = \mu$, v^κ not free in μ.

The inference rules are the usual congruence rules and are similar to the inference rules (sym), (trans), (cong), and (ξ) given for terms in the next section. If $\mu = \nu$ is provable from the axioms and rules for constructors, we write $\vdash_c \mu = \nu$. The constructor axiom system will be used to assign types to terms, since equal types will be associated with the same set of terms. It is worth mentioning that since we will only consider the pure theory of constructor equality, every constructor is provably equal to a unique normal form constructor with no subexpression matching the left-hand side of axiom (β) or (η). Consequently, we have the following lemma.

LEMMA 1. *If* $\vdash_c \sigma_1 \to \tau_1 = \sigma_2 \to \tau_2$, *then* $\vdash_c \sigma_1 = \sigma_2$ *and* $\vdash_c \tau_1 = \tau_2$. *Similarly, if* $\vdash_c \forall\mu = \forall\nu$, *then* $\vdash_c \mu = \nu$.

2.2. *Terms and Their Types*

We follow Reynolds (1974) and write free variables without type labels. However, we will always assign types to free variables using a technical device we call a type assignment. Since a constant must name a specific semantic value, we will require each constant to have a fixed type without free constructor variables.

Let $\mathscr{V}_{\text{term}}$ be an infinite collection of variables, which will remain fixed throughout the paper. Let $\mathscr{C}_{\text{term}}$ be a set of constants, each with a fixed, closed type. The set $PreTerm(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ of *pre-terms* over variables from $\mathscr{V}_{\text{cst}}$ and $\mathscr{V}_{\text{term}}$ and the indicated sets of constants is defined by

$$M ::= c \mid x \mid \lambda x{:}\,\sigma.M \mid MN \mid \lambda t.M \mid M\sigma,$$

where $c \in \mathscr{C}_{\text{term}}$, $x \in \mathscr{V}_{\text{term}}$, t is a type variable, and σ is a type expression over $\mathscr{C}_{\text{cst}}$ and $\mathscr{V}_{\text{cst}}$. We will define the well-typed terms below. The usual definitions of free and bound variables in lambda expressions may be stated

without reference to typing: λ binds x in $\lambda x{:}\sigma.M$ and t in $\lambda t.M$. Substitutions $\{N/x\}M$ of N for x and $\{\sigma/t\}M$ of σ for t are defined as usual to include renaming of bound variables in M to avoid capture.

As in most typed programming languages, the type of a second-order lambda term will depend on the context in which it occurs. We must know the types of all free variables before assigning a type. A *syntactic type assignment* B is a finite set

$$B = \{x_1{:}\sigma_1, ..., x_k{:}\sigma_k\}$$

of associations of types to variables,with no variable x appearing twice in B. If x does not occur in a syntactic type assignment B, then we write $B, x{:}\sigma$ for the type assignment

$$B, x{:}\sigma = B \cup \{x{:}\sigma\}.$$

If x occurs in B, then it is sometimes convenient to write $B(x)$ for the unique σ with $x{:}\sigma \in B$.

The typing relation is a three-place relation between type assignments, pre-terms, and type expressions. Let B be a syntactic type assigment, $M \in PreTerm(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$, and $\sigma{:}T$ a type expression. We define $B \vdash M{:}\sigma$, which is read "M has type σ with respect to B," by the derivation system below. The axioms about the typing relation are

$$\vdash c^{\tau}{:}\tau \qquad \text{and} \qquad x{:}\sigma \vdash x{:}\sigma$$

The type derivation rules are

$$(\to E)\ \frac{B \vdash M{:}\sigma \to \tau,\ B \vdash N{:}\sigma}{B \vdash MN{:}\tau}$$

$$(\to I)\ \frac{B, x{:}\sigma \vdash M{:}\tau}{B \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau}$$

$$(\forall E)\ \frac{B \vdash M{:}\forall\mu}{B \vdash M\tau{:}\mu\tau}$$

$$(\forall I)\ \frac{B \vdash M{:}\tau}{B \vdash \lambda t.M{:}\forall t.\tau}\ t \text{ not free in } B$$

and two rules that apply to terms of any form. A few comments are in order before discussing the remaining two typing rules.

In rule $(\forall E)$, we know that μ must have kind $T \Rightarrow T$, since $\forall\mu$ is assumed to be a type, and $\forall$ has kind $(T \Rightarrow T) \Rightarrow T$. Therefore, $\mu\tau$ will be a well-formed type expression. A related point about $(\forall I)$ is that while we can

only introduce $\forall$-types of the form $\forall t.\sigma ::= \forall(\lambda t.\sigma)$, we will be able to use the type equality rule below to derive typings of the form $B \vdash \lambda t.M:\forall\mu$, where μ is not of the form $(\lambda t.\sigma)$.

The restriction in the rule $(\forall I)$ is basically a matter of scope. In

$$x:t \vdash \lambda y:t \to t.\, yx:(t \to t) \to t,$$

for example, the type variable t refers to the same type on both sides of the turnstile. Therefore, it would not make sense to bind the occurences on the right-hand side without binding those on the left at the same time. If we were to allow the variable t to be bound on the right only, giving us

$$x:t \vdash \lambda t.\lambda y:t \to t.\, yx:\forall t.(t \to t) \to t;$$

then using $(\forall E)$ we could derive $x:t \vdash \lambda y:s \to s.yx:(s \to s) \to s$, which does not make any sense at all. This pathology is also discussed in Section 5.2 of Fortune *et al.* (1983).

Since additional hypotheses about the types of variables do not effect the type of a term, we have the rule

$$\text{(add hyp)}\ \frac{B \vdash M:\tau}{B, x:\sigma \vdash M:\tau}, \qquad x \text{ not in } B$$

for adding typing hypotheses. In addition, we have the type equality rule

$$\text{(type eq)}\ \frac{B \vdash M:\sigma, \vdash_c \sigma = \tau}{B \vdash M:\tau}.$$

We say M is a *term* if $B \vdash M:\sigma$ for some B and σ. However, we will seldom have occasion to write terms without also writing the relevant type assignment and type as well. In writing $B \vdash M:\sigma$ in the rest of the paper, we will mean that the typing $B \vdash M:\sigma$ is derivable, unless explicitly stated otherwise.

A simple induction on type derivations shows that if a term M has two types σ and τ, then these types are probably equal. Rule (type eq) guarantees the converse, so that for any type assignment B and pre-term M, either M has no type with respect to B or else the type of M is unique, up to equality. Furthermore, any derivation of a typing $B \vdash M:\sigma$ only uses the free variables of M and only depends on $B(x)$ up to type equality. Therefore, we have the following lemma.

LEMMA 2. *Suppose $B \vdash M:\sigma$ is well typed and let A be any syntactic type assignment such that $\vdash_c A(x) = B(x)$ for all x free in M. Then $A \vdash M:\tau$ iff $\vdash_c \sigma = \tau$.*

Since we have chosen the pure theory of β, η-conversion between constructor expressions, every equivalence class of constructor expressions has a unique normal form. Therefore, for each syntactic type assignment B and $M \in PreTerm(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$, if $B \vdash M:\sigma$ and $B \vdash M:\tau$ then σ and τ may be simplified to the same normal form. Assuming the types given in B are in normal form,[2] it is easy to write an efficient algorithm which computes the normal form type of M with respect to B when it exists, and returns *error* if it is not (cf. Leivant, 1983a). Typings have some natural substitution properties. For example, if $B \vdash N:\rho$ and $B, x:\rho \vdash M:\sigma$, then $B \vdash \{N/x\}\, M:\sigma$. In addition, if we define $\{\sigma/t\}B$ by substituting σ for t in every type occurring in B, then whenever $B \vdash M:\rho$, we have $\{\sigma/t\}\, B \vdash \{\sigma/t\}\, M:\{\sigma/t\}\rho$. In particular, if t is not free in B, then $B \vdash \{\gamma/t\}\, M:\{\gamma/t\}\rho$. Another useful substitution property is summarized by the following lemma.

LEMMA 3. *Let S be a substitution of constructor expressions for constructor variables and pre-terms for ordinary variables such that $A \vdash Sx:S\sigma$ is derivable for every $x:\sigma \in B$. If $B \vdash M:\tau$ is derivable, then so is $A \vdash SM:S\tau$.*

2.3. *Relationship to Other Systems*

It is best to think of the second-order lambda calculus as a family of related systems, rather than a single calculus. The particular calculus we have chosen is a compromise between the most basic calculus presented in Reynolds (1974) and the extensions considered in (Girard, 1972; McCracken, 1979). The types used in the second-order lambda calculus of Reynolds (1974), studied in (Donahue, 1979; Fortune *et al.*, 1983; Leivant, 1983a, 1983b; Reynolds, 1984) are a subset of ours. Specifically, only normal form type expressions are used, and no constructor symbols besides type variables, type constants, $\rightarrow$ and $\forall$ are allowed. It is possible to show that our typing rules and equational proof rules are conservative over Reynolds', and so we consider the variables of higher kinds an essentially benign extension. However, because of lambda abstraction in type expressions, type equality in our system becomes more complicated.

One straightforward extension of Reynolds' calculus is to add cartesian product types $\sigma \times \tau$. This system may be obtained from ours by adding a type constructor $\times:T \Rightarrow (T \Rightarrow T)$ and constants for pairing and projection functions. Girard also considers a system with existential types, a calculus with type constructor $\exists$ of kind $(T \Rightarrow T) \Rightarrow T$ which is "dual" to $\forall$ and related to existential quantification in logical formulas (see Girard, 1972;

[2] If types in B are not given in normal form, then they may have to be simplified, which cannot necessarily be done efficiently (Statman, 1979).

Mitchell and Plotkin, 1988 for further discussion). One advantage of the calculus we have chosen is that it is easy to extend the syntax to include additional type constructors of any kind, and it will be quite easy to see how to modify the model definitions accordingly.

A more significant extension of the basic second-order calculus, which Girard called F_2, is obtained be allowing lambda abstraction in terms over variables of higher kinds. For example, if $f^{T \Rightarrow T}$ is a variable of kind $T \Rightarrow T$, then the system we have defined allows the term $\lambda x{:}ft.x{:}ft \to ft$. It is quite sensible to allow the variable f to be lambda bound, giving us the term $\lambda f^{T \Rightarrow T}.\lambda x{:}ft.x$. To type this term, we need a "higher order" $\forall$ of kind $((T \Rightarrow T) \Rightarrow T) \Rightarrow T)$. Adding this constructor constant and allowing the associated lambda abstraction leads to Girard's "higher order" lambda calculus F_3. By adding type quantification over successively higher kinds, we obtain the languages $F_4, F_5, \ldots$; the union of all these languages is F_ω. (See Section I.9 of Girard, 1972 for further discussion.) We hope that by including variables of higher kinds, we will provide enough information to allow the reader to extend our model definition and completeness proof to any of Girard's higher order calculi or the calculus of the theory of species discussed in Stenlund (1972).

In addition to the generality of considering constructor expressions of all kinds, constructors will be used in the discussion of combinatory models to write down the types of polymorphic combinators. Another subtle function of variables of higher kind will be mentioned after the definition of environment models and summarized in Lemma 11.

3. Equations Between Terms

Since we write terms with type assignments, it is natural to include type assignments in equations as well. By *equation*, we will mean an expression

$$B \vdash M = N{:}\sigma,$$

where $B \vdash M{:}\sigma$ and $B \vdash N{:}\sigma$. Intuitively, an equation $\{x_1{:}\sigma_1, \ldots, x_k{:}\sigma_k\} \vdash M = N{:}\sigma$ means, "if the variables $x_1, \ldots, x_k$ have types $\sigma_1, \ldots, \sigma_k$ (respectively), then terms M and N denote the same element of type σ." Since $\vdash$ is considered an implication, an equation may hold vacuously if it is impossible to assign the variables meaning of the correct types. This may happen when types are empty, a complication we will avoid by assuming that every type is nonempty. (Empty types are discussed in Meyer *et al.*, 1987; Mitchell and Moggi, 1987; see also the discussion following inference rule (remove hyp) below.)

The axioms and inference rules for equations between second-order

lambda terms are similar to the axioms and rules of the ordinary typed lambda calculus. The main difference is that we tend to have two versions of each axiom or rule, one for ordinary function abstraction or application, and another for type abstraction or application.

Axioms for Terms.

(α) $B \vdash \lambda x{:}\sigma.M = \lambda y{:}\sigma.\{y/x\}\, M{:}\sigma \to \tau$, y not in B,
$B \vdash \lambda t.M = \lambda s.\{s/t\}\, M{:}\forall t.\sigma$, s not free in $\lambda t.M$,

(β) $B \vdash (\lambda x{:}\sigma.M)\, N = \{N/x\}\, M{:}\sigma$,
$B \vdash (\lambda t.M)\, \tau = \{\tau/t\}\, M{:}\sigma$,

(η) $B \vdash \lambda x{:}\sigma.Mx = M{:}\sigma \to \tau$, x not free in M,
$B \vdash \lambda t.Mt = M{:}\forall t.\sigma$, t not free in M.

Although some authors prefer to omit it, we have included the extensionality axiom (η). This axiom is used to prove that if $Mx = Nx$ for a fresh variable x not appearing in M or N, then $M = N$. Models satisfying (η) seem more natural, since (η) (in combination with the other axioms and rules) implies that two elements of functional type $\sigma \to \tau$ are equal whenever they give equal results for all arguments of type σ. In addition, assuming extensionality will simplify much of the discussion of combinatory models in Section 6. Non-extensional models will be discussed briefly in Section 6.5.

It is not necessary to include a reflexivity axiom because $M = M$ follows from (β) by the symmetry and transitivity rules below. In (α) for ordinary variables, the assumption that y is not declared in B may be weakened to y not free in M. However, the axiom as stated is slightly easier to work with (see the soundness proof in Section 5), and the alternative axiom is easily derived using the inference rules below.

Inference Rules for Terms.

$$\text{(sym)} \qquad \frac{B \vdash M = N{:}\sigma}{B \vdash N = M{:}\sigma}$$

$$\text{(trans)} \qquad \frac{B \vdash M = N{:}\sigma,\ B \vdash N = P{:}\sigma}{B \vdash M = P{:}\sigma}$$

$$(\text{cong})_1 \qquad \frac{B \vdash M = N{:}\sigma \to \tau,\ B \vdash P = Q{:}\sigma}{B \vdash MP = NQ{:}\tau}$$

$$(\text{cong})_2 \qquad \frac{B \vdash M = N{:}\forall\mu,\ \vdash_c \sigma = \tau}{B \vdash M\sigma = N\tau{:}\mu\sigma}$$

$$(\xi)_1 \qquad \frac{B, x{:}\sigma \vdash M = N{:}\rho,\ \vdash_c \sigma = \tau}{B \vdash \lambda x{:}\sigma.M = \lambda x{:}\tau.N{:}\sigma \to \rho}$$

$$(\xi)_2 \qquad \frac{B \vdash M = N : \sigma}{B \vdash \lambda t.M = \lambda t.N : \forall t.\sigma} \qquad t \text{ not free in } B$$

$$\text{(constr sub)} \quad \frac{B \vdash M = N : \sigma,\ \mu : \kappa}{B \vdash \{\mu/v^\kappa\}\, M = \{\mu/v^\kappa\}\, N : \{\mu/v^\kappa\}\sigma}$$

$$v^\kappa \text{ not free in } B \text{ and } \kappa \not\equiv T.$$

Since type assignments and types are included in the syntax of equations, we need equational versions of the (add hyp) and (type eq) typing rules:

$$\text{(add hyp)} \quad \frac{B \vdash M = N : \sigma}{B, x : \sigma \vdash M = N : \sigma} \qquad x \text{ not in } B$$

$$\text{(type eq)} \quad \frac{B \vdash M = N : \sigma,\ \vdash_c \sigma = \tau}{B \vdash M = N : \tau}.$$

In addition, we will adopt an inference rule for removing typing hypotheses. This rule allows us to eliminate assumptions about variables that do not occur free in either term:

$$\text{(remove hyp)} \quad \frac{B, x : \sigma \vdash M = N : \tau}{B \vdash M = N : \tau} \qquad x \text{ not free in } M \text{ or } N.$$

While the analogous typing rule is an admissible rule of the language (Lemma 2), this equational rule is only sound if we assume that every type is nonempty. For example, the equation

$$z : \sigma \vdash \lambda x : t.\lambda y : t.x = \lambda x : t.\lambda y : t.y : t \to t \to t$$

may hold vacuously in some nontrivial model if σ is an empty type. However, the equation

$$\varnothing \vdash \lambda x : t.\lambda y : t.x = \lambda x : t.\lambda y : t.y : t \to t \to t,$$

which follows by rule (remove hyp), only holds in trivial models with no more than one element of each type.

It is easy to check that for each of the inference rules, if the antecedents are well-typed equations, then the consequent is a well-typed equation. The only slightly nontrivial cases are $(\text{cong})_2$ and $(\xi)_1$, in which we must consider type equality. In rule $(\text{cong})_2$, if $B \vdash M = N : \forall\mu$ and $\vdash_c \sigma = \tau$, then $\vdash_c \mu\sigma = \mu\tau$ and so $B \vdash M\sigma : \mu\sigma$ and $B \vdash N\tau : \mu\tau$ have probably equal types. The verification of $(\xi)_1$ is similar, but uses Lemma 2 to show that if $B, x : \sigma \vdash M = N : \rho$ is well typed and $\vdash_c \sigma = \tau$, then we have $B, x : \tau \vdash N : \rho$, and so $B \vdash \lambda x : \sigma.M = \lambda x : \tau.N : \sigma \to \rho$ is well typed. The reason for

including type equality in these two inference rules is so that term equality respects constructor equality. More precisely, term equality has the following substitution property.

LEMMA 4. *If* $B \vdash M:\sigma$ *is well typed,* $\vdash_c \mu = \nu$, *and* N *is obtained from* M *by substituting* $\nu:\kappa$ *for one or more occurences of* $\mu:\kappa$, *then we can prove* $B \vdash M = N:\sigma$ *from the axioms and inference rules above.*

This lemma is easily proved by induction on M, using Lemma 2 to show that the equation $B \vdash M = N:\sigma$ is well typed. Rule (constr sub) is used to show that equality is closed under substitution. Since we have lambda abstraction and application for type variables and ordinary variables, we can prove substitution instances of equations using (β). However, since we cannot lambda-abstract constructor variables of kind κ different from T, we need rule (constr sub) to complete the proof of the following lemma.

LEMMA 5. *Let* S *be a substitution of constructor expressions for constructor variables and pre-terms for ordinary variables such that* $A \vdash Sx:S\sigma$ *for every* $x:\sigma \in B$. *Then from any well-typed equation* $B \vdash M = N:\tau$ *we can prove* $A \vdash SM = SN:S\tau$.

Lemma 3 may be used to show that the equation $A \vdash SM = SN:S\tau$ in the statement of Lemma 5 is well typed.

A *second-order lambda theory* Γ is a set of equations containing all instances of the term axioms and closed under the inference rules. We will not include equations between constructors in theories, since we will always use the same constructor equations.

4. SECOND-ORDER ENVIRONMENT MODELS

4.1. *Introduction*

Models for second-order lambda calculus will have several parts: we use "kind frames" to interpret kinds and constructors and additional sets indexed by types to interpret terms. All of these parts will be collected together in what we call a frame (after Henkin, 1950). We define models as frames which satisfy an additional condition involving the meanings of terms. This form of definition is similar to the "environment model" definition for untyped lambda calculus given in (Meyer, 1982). Since the definition of second-order model is fairly complicated, we will try to illustrate some of the underlying ideas using untyped lambda calculus.

Untyped lambda calculus has untyped applications MN and function expressions $\lambda x.M$. If we think of M and N as denoting elements of some

"domain" D, then the application MN of M to N makes sense if we have some way of turning M into a function. This is accomplished using an *element-to-function* map Φ. Conversely, we can easily regard $\lambda x.M$ as a function from D to D, since M specifies a single function value for every value of x. But in order to find a meaning for $\lambda x.M$ in D, we need a *function-to-element*[3] map $\Psi = \Phi^{-1}$. An *extensional applicative structure* $\langle D, \Phi \rangle$ consists of a set D together with a mapping Φ such that for some set $[D \to D]$ of functions from D to D,

$$\Phi : D \to [D \to D] \text{ is } \textit{one-to-one} \text{ and } \textit{onto}.$$

In other words, an extensional applicative structure $\langle D, \Phi \rangle$ consists of a set D together with a bijection Φ between D and a set $[D \to D]$ of functions from D to D. In general, we will be a bit informal about Φ and abbreviate $(\Phi(d))(e)$ to de.

If η is an environment mapping untyped varibles to D, then the meaning $[\![M]\!]\eta$ of term M in environment η is defined by

$$[\![x]\!]\eta = \eta(x)$$

$$[\![MN]\!]\eta = \Phi([\![M]\!]\eta)([\![N]\!]\eta)$$

$$[\![\lambda x.M]\!]\eta = \Phi^{-1}(f), \qquad \text{where} \quad f : D \to D \text{ satisfies } f(d) = [\![M]\!]\eta[d/x].$$

Although this definition may look fine, there is a serious problem with the meanings of terms. The meaning of a lambda term $\lambda x.M$ is defined by applying Φ^{-1} to some function f. The function f is well defined, but f may not be in the domain $[D \to D]$ of Φ^{-1}. Consequently, the meaning of $\lambda x.M$ may not be defined. Thus we must distinguish models, structures in which every term has a meaning, from arbitrary applicative structures. One straightforward model definition is the environment model definition. We say an applicative structure is an *environment model* if the meaning of every term M in every environment η is a well-defined element of D. Some equivalent model definitions are discussed in (Barendregt, 1984; Koymans, 1982; Meyer, 1982).

A similar definition can be given for the ordinary typed lambda calculus. With typed application, we need an "element-to-function" map $\Phi_{a,b}$ for each pair of types a and b. The function $\Phi_{a,b}$ maps the domain $\mathrm{Dom}^{a \to b}$ of elements of type $a \to b$ to some set $[\mathrm{Dom}^a \to \mathrm{Dom}^b]$ of functions from

[3] Since we are only concerned with extensional models (see Section 6.5), we assume that $\Psi = \Phi^{-1}$. In nonextensional models, there may be two elements $d_1, d_2 \in D$ representing the same function $f = \Phi(d_1) = \Phi(d_2)$. In this case, Φ has no inverse and we rely on a second function Ψ to choose a particular $d = \Psi(f)$ representing f. See (Barendregt, 1984; Meyer, 1982) for further discussion.

Dom^a to Dom^b, and we use $\Phi_{a,b}^{-1}$ to give meaning to typed lambda abstractions $\lambda x{:}\sigma.M{:}\sigma \to \tau$. Since constructor expressions are a notational variant of simple typed lambda terms, we will interpret constructor expressions in structures like this.

4.2. *Semantics of Constructor Expressions*

Constructor expressions are interpreted using kind frames, which are essentially frames for the simple typed lambda calculus. A *kind frame*, *Kind*, for a set $\mathscr{C}_{\mathrm{cst}}$ of constructor constants is a tuple

$$Kind = \langle \{\mathrm{Kind}^\kappa \mid \kappa \text{ a kind}\}, \{\Phi_{\kappa_1,\kappa_2} \mid \kappa_1, \kappa_2 \text{ kinds}\}, \mathscr{I} \rangle,$$

where

$$\Phi_{\kappa_1,\kappa_2}: \mathrm{Kind}^{\kappa_1 \Rightarrow \kappa_2} \to [\mathrm{Kind}^{\kappa_1} \to \mathrm{Kind}^{\kappa_2}]$$

is a bijection between $\mathrm{Kind}^{\kappa_1 \Rightarrow \kappa_2}$ and some set $[\mathrm{Kind}^{\kappa_1} \to \mathrm{Kind}^{\kappa_2}]$ of functions from Kind^{κ_1} to Kind^{κ_2}, and

$$\mathscr{I}: \mathscr{C}_{\mathrm{cst}} \to \bigcup_\kappa \mathrm{Kind}^\kappa$$

preserves kinds, i.e., $\mathscr{I}(c^\kappa) \in \mathrm{Kind}^\kappa$. Since constructor expressions include all typed lambda expressions, we will be interested in kind frames which are models of the simple typed lambda calculus.

Let η be an environment mapping constructor variables to $\bigcup_\kappa \mathrm{Kind}^\kappa$ such that for each v^κ, we have $\eta(v^\kappa) \in \mathrm{Kind}^\kappa$. The meaning $[\![\mu]\!]\eta$ of a constructor expression μ in environment η is defined as follows (see Barendregt, 1984; Friedman, 1975; Henkin, 1950; Statman, 1985):

$$[\![v^\kappa]\!]\eta = \eta(v^\kappa),$$

$$[\![c^\kappa]\!]\eta = \mathscr{I}(c^\kappa),$$

$$[\![\mu\nu]\!]\eta = (\Phi_{\kappa_1,\kappa_2}[\![\mu]\!]\eta)[\![\nu]\!]\eta,$$

$$[\![\lambda v^\kappa.\mu]\!]\eta = \Phi_{\kappa_1,\kappa_2}^{-1} f, \qquad \text{where} \quad f(a) = [\![\mu]\!]\eta[a/v^\kappa] \text{ for all } a \in \mathrm{Kind}^\kappa.$$

We say *Kind* is a *kind environment model* for $\mathscr{C}_{\mathrm{cst}}$ if every constructor expression over $\mathscr{C}_{\mathrm{cst}}$ has a meaning in every environment for *Kind*. We will give an equivalent algebraic definition in Section 6.

Note that we have not had to distinguish $\to$ and $\forall$ from other constructor constants. It is implicit in the definition of kind frame that Kind^T must be closed under $\to$ (viewed as a binary operation) and that the result of applying $\forall$ to any function in $\mathrm{Kind}^{T \Rightarrow T}$ is also an element of Kind^T. The advantage of working with constructors and kinds is that our definition

applies to any set of type constructors. If we also have a "product-type" constructor

$$\times: \mathrm{Kind}^{T \Rightarrow (T \Rightarrow T)}$$

among our constants $\mathscr{C}_{\mathrm{cst}}$, then the definition of kind frame also requires that Kind^T be closed under $\times$ (viewed as a binary operation).

Since it is a very convenient way of making definitions more readable, we will often use $\forall$, $\rightarrow$, and other constants for their denotations in *Kind* when there is no danger of confusion. For example, if $f \in \mathrm{Kind}^{T \Rightarrow T}$, we write $\forall f$ rather than $(\mathscr{I}(\forall))f$. It is worth mentioning that we could dispense with the mappings $\Phi_{\kappa_1, \kappa_2}$ in kind frames by letting $\mathrm{Kind}^{\kappa_1 \Rightarrow \kappa_2}$ be a set of functions from Kind^{κ_1} to Kind^{κ_2}. However, the slightly more involved setting described above provides more motivation for the interpretation of terms below. (In interpreting polymorphic terms, we cannot eliminate the Φ functions.) In addition, the functions $\Phi_{\kappa_1, \kappa_2}$ and $\Phi^{-1}_{\kappa_1, \kappa_2}$ simplify the completeness proof slightly.

4.3. *Frames and Environment Models*

As in the definition of untyped environment model, we first define a structure, called a frame, and then define models by distinguishing frames which interpret all terms from those that do not. Second-order frames will include typed versions of Φ, plus an additional collection of Φ's for polymorphic types. Intuitively, a polymorphic term $\lambda t.M$ denotes a function from the set of types to elements of types. More precisely, we will be able to regard the meaning of $\lambda t.M$ as an element of the cartesian product $\Pi_{a \in \mathrm{Kind}^T}.\mathrm{Dom}^{f(a)}$ for some function $f: \mathrm{Kind}^{T \Rightarrow T}$ determined from the typing of M. Therefore, for every function $f \in \mathrm{Kind}^{T \Rightarrow T}$, a second-order model will have a function Φ_f mapping $\mathrm{Dom}^{\forall f}$ to some subset $[\Pi_{a \in \mathrm{Kind}^T}.\mathrm{Dom}^{f(a)}]$ of $\Pi_{a \in \mathrm{Kind}^T}.\mathrm{Dom}^{f(a)}$.

A *second-order frame* $\mathscr{F}$ for terms over constants from $\mathscr{C}_{\mathrm{cst}}$ and $\mathscr{C}_{\mathrm{term}}$ is a tuple

$$\mathscr{F} = \langle \mathit{Kind}, \mathit{Dom}, \{\Phi_{a,b} \mid a, b \in \mathrm{Kind}^T\}, \{\Phi_f \mid f \in \mathrm{Kind}^{T \Rightarrow T}\} \rangle$$

satisfying conditions (i) through (iv):

(i) $\mathit{Kind} = \langle \{\mathrm{Kind}^\kappa\}, \{\Phi_{\kappa_1, \kappa_2}\}, \mathscr{I} \rangle$ is a kind frame for $\mathscr{C}_{\mathrm{cst}}$

(ii) $\mathit{Dom} = \langle \{\mathrm{Dom}^a \mid a \in \mathrm{Kind}^T\}, \mathscr{I}_{\mathit{Dom}} \rangle$ is a family of nonempty sets Dom^a indexed by elements $a \in \mathrm{Kind}^T$, together with a function

$$\mathscr{I}_{\mathit{Dom}}: \mathscr{C}_{\mathrm{term}} \rightarrow \bigcup_a \mathrm{Dom}^a \qquad \text{with} \quad \mathscr{I}_{\mathit{Dom}}(c^\tau) \in \mathrm{Dom}^{[\![\tau]\!]} \text{ for all } c^\tau \text{ in } \mathscr{C}_{\mathrm{term}},$$

(iii) For each $a, b \in \mathrm{Kind}^T$, we have a set $[\mathrm{Dom}^a \rightarrow \mathrm{Dom}^b]$ of func-

tions from Dom^a to Dom^b with bijection $\Phi_{a,b}: \mathrm{Dom}^{a\to b} \to [\mathrm{Dom}^a \to \mathrm{Dom}^b]$.

(iv) For every $f \in \mathrm{Kind}^{[T\Rightarrow T]}$, we have a subset $[\Pi_{a\in \mathrm{Kind}^T}\mathrm{Dom}^{f(a)}] \subseteq \Pi_{a\in \mathrm{Kind}^T}\mathrm{Dom}^{f(a)}$ with bijection $\Phi_f: \mathrm{Dom}^{\forall f} \to [\Pi_{a\in \mathrm{Kind}^T}\mathrm{Dom}^{f(a)}]$.

Essentially, condition (iii) states that $\mathrm{Dom}^{a\to b}$ must "represent" some set $[\mathrm{Dom}^a \to \mathrm{Dom}^b]$ of functions from Dom^a to Dom^b. Similarly, condition (iv) specifies that $\mathrm{Dom}^{\forall f}$ must represent some subset $[\Pi_{a\in \mathrm{Kind}^T}\mathrm{Dom}^{f(a)}]$ of the product $\Pi_{a\in \mathrm{Kind}^T}\mathrm{Dom}^{f(a)}$.

Terms are interpreted using Φ's for application and Φ^{-1}'s for abstraction. Since different Φ and Φ^{-1} functions are used, depending on the types of terms, the type of a term will be used to define its meaning. If B is a type assignment and η an environment mapping $\mathscr{V}_{\mathrm{cst}}$ to elements of the appropriate kinds and $\mathscr{V}_{\mathrm{term}}$ to elements of $\bigcup Dom$, we say that η *satisfies* B, written $\eta \models B$, if

$$\eta(x) \in \mathrm{Dom}^{[\![\sigma]\!]\eta}$$

for every $x{:}\sigma \in B$.

Let $\mathscr{F}$ be a second-order frame. For any well-typed term $B \vdash M{:}\sigma$ and environment $\eta \models B$, we will define the meaning $[\![B \vdash M{:}\sigma]\!]\eta$ inductively below. Although it may seem unnecessarily complicated, the simplest way to define meanings seems to be by induction on the derivation of typings, rather that the structure of terms. This is simply a technical device. Since any derivation of $B \vdash M{:}\sigma$ must follow the structure of M fairly closely, there is not much difference between the two forms of induction. However, since there is some flexibility in where rules (add hyp) and (type eq) might be used, there is a little more structure in the derivation of a typing of $B \vdash M{:}\sigma$ than in the expression $B \vdash M{:}\sigma$ itself. In particular, a derivation gives specific typings to each of the subterms, while the fact that $B \vdash M{:}\sigma$ is derivable only determines the types of subterms of M up to type equality.

The lambda abstraction case illustrates some of the advantages of induction on typing derivations. If we define the meaning of $B \vdash \lambda x{:}\sigma.M{:}\rho$ using induction on the structure of terms, we must argue that $\vdash_c \rho = \sigma \to \tau$ for some τ and that it does not matter which τ we pick. We need $\rho = \sigma \to \tau$ so that we know the domain and range types, and we need to show that the choise of τ is inessential so that it is clear that the meaning of each term is uniquely determined. These arguments are not entirely trivial since rule (type eq) allows the syntactic type of the lambda abstraction to have almost any form. In addition, we need to find some type assignment A with $A \vdash M{:}\tau$ so that we may apply the induction hypothesis and argue that the choice of A is inessential. However, using induction on typing derivations, the inductive assumption for rule $(\to I)$ is that $B \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau$ follows from typing $B, x{:}\sigma \vdash M{:}\tau$ and that the meaning of $B, x{:}\sigma \vdash M{:}\tau$

is defined for any environment satisfying $B, x:\sigma$. This gives us specific domain and range types for the lambda term and also guarantees that x does not occur in B, so that $B, x:\sigma$ is a well-formed type assignment. Some similar points apply in the $(\forall I)$ case, and will be mentioned below. Once we have given the definition of meaning, it will be easy to prove that the meaning of a well-typed term $B \vdash M:\sigma$ does not depend on the way this typing is derived.

The inductive clauses of the meaning function are given in the same order as the typing rules in Section 2.2, with rules $(\to E)$, $(\to I)$, $(\forall E)$, and $(\forall I)$ preceding rules (add hyp) and (type eq) which do not rely on the forms of terms:

$$[\![B \vdash x:\sigma]\!]\eta = \eta(x),$$

$$[\![B \vdash c:\sigma]\!]\eta = \mathscr{I}_{Dom}(c),$$

$$[\![B \vdash MN:\tau]\!]\eta = (\Phi_{a,b}[\![B \vdash M:\sigma \to \tau]\!]\eta)[\![B \vdash N:\sigma]\!]\eta,$$

where $a = [\![\sigma]\!]\eta$ and $b = [\![\tau]\!]\eta$,

$$[\![B \vdash \lambda x:\sigma.M:\sigma \to \tau]\!]\eta = \Phi_{a,b}^{-1}\, g, \text{ where}$$

$$g(d) = [\![B, x:\sigma \vdash M:\tau]\!]\,\eta[d/x] \qquad \text{for all } d \in \mathrm{Dom}^a,$$

$$a = [\![\sigma]\!]\,\eta \qquad \text{and} \qquad b = [\![\tau]\!]\,\eta$$

$$[\![B \vdash M\tau:\mu\tau]\!]\eta = (\Phi_f [\![\vdash M:\forall\mu]\!]\eta)[\![\tau]\!]\eta, \qquad \text{where } f = [\![\mu]\!]\eta,$$

$$[\![B \vdash \lambda t.M:\forall t.\sigma]\!]\eta = \Phi_f^{-1}\, g, \text{ where}$$

$$g(a) = [\![B \vdash M:\sigma]\!]\,\eta[a/t] \qquad \text{for all } a \in \mathrm{Kind}^T, \text{ and}$$

$f \in \mathrm{Kind}^{T \Rightarrow T}$ is the function $[\![\lambda t.\sigma]\!]\eta$

$$[\![B, x:\sigma \vdash M:\tau]\!]\eta = [\![B \vdash M:\tau]\!]\eta, \qquad \text{where the left-hand typing}$$

follows by the rule (add hyp)

$$[\![B \vdash M:\tau]\!]\eta = [\![B \vdash M:\sigma]\!]\eta, \qquad \text{where the left-hand typing follows}$$

by rule (type eq).

It is relatively easy to see that the environments mentioned on the right-hand sides of these clauses all satisfy the appropriate syntactic type assignments. One nontrivial case is type abstraction by rule $(\forall I)$. Since we assume that $B \vdash \lambda t.M:\forall t.\sigma$ follows from $B \vdash M:\sigma$, we know that t does not occur free in B. Therefore, if $\eta \models B$, then any $\eta[a/t]$ satisfies B as well.

In the inductive definition of meaning, there is no guarantee that $[\![B \vdash M:\sigma]\!]\eta$ exists for every well-typed term. For example, g in the $\lambda x{:}\sigma.M$ case may not be in the domain of $\Phi_{a,b}^{-1}$, and similarly for g in the $\lambda t.M$ case. Therefore, we make the following definition. A second-order frame

$$\mathscr{F} = \langle Kind, Dom, \{\Phi_{a,b} \mid a, b \in \mathrm{Kind}^T\}, \{\Phi_f \mid f \in \mathrm{Kind}^{T \Rightarrow T}\}\rangle$$

is an *environment model* if (i) *Kind* is a kind environment model and (ii) for every term $B \vdash M:\sigma$ and every environment $\eta \models B$, the meaning $[\![B \vdash M:\sigma]\!]\eta$ exists as defined above.

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

LEMMA 6. *Let η be an environment for a model $\langle Kind, Dom, \{\Phi_{a,b}\}, \{\Phi_f\}\rangle$. If $\eta \models B$, then $[\![B \vdash M:\sigma]\!]\eta \in Dom^{[\![\sigma]\!]\eta}$.*

In addition, we can show that the meaning of a well-typed term $B \vdash M:\sigma$ does not depend on the derivation of the typing. This is the intent of the following two lemmas. It will be helpful to name typing derivations and write, e.g., Δ, Δ_1 for the derivation Δ followed by derivation Δ_1. An easy induction on typing derivations shows that rules (add hyp) and (type eq) do not effect the meaning of terms.

LEMMA 7. *Suppose Δ is a derivation of $A \vdash M:\sigma$ and Δ, Δ_1 is a derivation of $B \vdash M:\tau$ such that only rules (add hyp) and (type eq) appear in Δ_1. Then for any $\eta \models B$, we have*

$$[\![A \vdash M:\sigma]\!]\eta = [\![B \vdash M:\tau]\!]\eta,$$

where the meanings are taken with respect to derivations Δ and Δ, Δ_1.

Using induction on the structure of terms, and Lemmas 1, 2, and 7, we can now show that the meanings of "compatible" typings of a term are equal.

LEMMA 8. *Suppose Δ and Δ_1 are derivations of typings $A \vdash M:\sigma$ and $B \vdash M:\tau$, respectively, and that $\vdash_c A(x) = B(x)$ for every x free in M. Then*

$$[\![A \vdash M:\sigma]\!]\eta = [\![B \vdash M:\tau]\!]\eta,$$

where the meanings are defined using Δ and Δ_1, respectively.

It follows that the meaning of any well-typed term is independent of the typing derivation.

COROLLARY 9. *Suppose Δ and Δ_1 are derivations of a typing $B \vdash M:\sigma$. Then for any environment $\eta \models B$, the meaning $[\![B \vdash M:\sigma]\!]\eta$ defined using induction on Δ is the same as the meaning defined using Δ_1.*

This corollary allows us to regard an equation $B \vdash M = N:\sigma$ as an equation between M and N, rather than derivations of typings $B \vdash M:\sigma$ and $B \vdash N:\sigma$. In addition to the corollary, Lemma 8 shows that meaning is a congruence with respect to type equality, which will be useful in showing the soundness of the equational proof rules for terms.

A very useful fact is the following substitution lemma.

LEMMA 10 (Substitution). (i) *Suppose $B, x:\sigma \vdash M:\tau$ and $B \vdash N:\sigma$. If $\eta \models B$ then*

$$[\![B \vdash \{N/x\}\, M:\sigma]\!]\eta = [\![B, x:\sigma \vdash M:\tau]\!]\,\eta[[\![B \vdash N:\sigma]\!]\,\eta/x].$$

(ii) *If $B \vdash M:\sigma$ and t is not free in B, then*

$$[\![B \vdash \{\tau/t\}\, M:\{\tau/t\}\,\sigma]\!]\eta = [\![B \vdash M:\sigma]\!]\,\eta[[\![\tau]\!]\,\eta/t].$$

(iii) *If μ, ν are constructor expressions with $\nu:\kappa$ and $\nu \in \mathscr{V}_{\text{cst}}$ a variable of kind κ, then*

$$[\![\{\nu/v\}\,\mu]\!]\eta = [\![\mu]\!]\,\eta[[\![\nu]\!]\,\eta/v].$$

Parts (i) and (ii) of the lemma are easily proved by induction on terms. Part (iii) is a well-known property of the simple typed lambda calculus. It is also easy to prove that the meaning $[\![B \vdash M:\sigma]\!]\eta$ does not depend on $\eta(x)$ or $\eta(t)$ for x or t not free in M.

An important lemma about the environment model condition is that it does not depend on the set of constants of the language.

LEMMA 11. *Let $\mathscr{F}$ be an environment model for terms over constants $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$. If we expand $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$ to $\mathscr{C}'_{\text{cst}}$ and $\mathscr{C}'_{\text{term}}$, and interpret the fresh constants as any elements of $\mathscr{F}$ of the appropriate kinds or types, then we obtain an environment model for terms over constants from $\mathscr{C}'_{\text{cst}}$ and $\mathscr{C}'_{\text{term}}$.*

This lemma, which will be used in the proof of the combinatory model theorem (Theorem 18), is easily proved using the fact that every constant is equal to some variable in some environment. More specifically, if we want to know that a term $B \vdash M:\sigma$ with constants has a meaning in some environment $\eta \models B$ for frame $\mathscr{F}$, then we begin by replacing the constants

with fresh variables. Then, we choose some environment η_1 which is identical to η on the free variables of $B \vdash M:\sigma$, and which gives the new variables the values of the constants they replace. If $\mathscr{F}$ is an environment model, then the new term must have a meaning in the chosen environment, and so it is easy to show that $B \vdash M:\sigma$ must have a meaning in η. However, this argument applies only if we have variables of all kinds; without this, the hypothesis that $\mathscr{F}$ is an environment model for constants $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$ is not enough. In particular, the lemma fails if our frames include an arbitrary set of functions in $\text{Kind}^{T \Rightarrow T}$, but do not have variables of kind $T \Rightarrow T$. This was overlooked in (Bruce and Meyer, 1984).

5. Completeness

In this section, we show that the axioms and inference rules are sound and complete for deducing equations between terms. We need the usual definitions of satisfaction and semantic implication to state the soundness and completeness theorems. An environment $\eta \models B$ for model $\mathscr{F}$ *satisfies* an equation $B \vdash M = N:\sigma$, written

$$\mathscr{F}, \eta \models B \vdash M = N:\sigma.$$

if $[\![B \vdash M:\sigma]\!]\eta = [\![B \vdash N:\sigma]\!]\eta$. A model $\mathscr{F}$ *satisfies* an equation $B \vdash M = N:\sigma$, written

$$\mathscr{F} \models B \vdash M = N:\sigma$$

if $\mathscr{F}$ and η satisfy $B \vdash M = N:\sigma$ for all $\eta \models B$. Similarly, a model $\mathscr{F}$ *satisfies* a set Γ of equations if $\mathscr{F}$ satisfies every equation in Γ. A set Γ of equations *semantically implies* an equation $B \vdash M = N:\sigma$, written

$$\Gamma \models B \vdash M = N:\sigma,$$

if $\mathscr{F} \models B \vdash M = N:\sigma$ whenever $\mathscr{F} \models \Gamma$.

It is easy to verify that the axioms and inference rules are sound for models without empty types.

Lemma 12. (Soundness). *Let Γ be a set of equations and let $B \vdash M = N:\sigma$ be an equation. If Γ proves $B \vdash M = N:\sigma$, then $\Gamma \models B \vdash M = N:\sigma$.*

Proof. The proof is entirely straightforward. We will show that two axioms, (α) and (β), are valid, leaving the details for remaining axioms and inference rules to the reader. Suppose $B \vdash \lambda x:\sigma.M:\sigma \to \tau$ is well typed and

assume the variable y does not occur in B. Let $\eta \models B$. For $a = [\![\sigma]\!]\eta$ and $b = [\![\tau]\!]\eta$, we have

$$\begin{aligned}
&[\![B \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau]\!]\eta \\
&\quad = \Phi_{a,b}^{-1}(\lambda d \in D_a.\, [\![B, x{:}\sigma \vdash M{:}\tau]\!]\,\eta[d/x]) \\
&\quad = \Phi_{a,b}^{-1}(\lambda d \in D_a.\, [\![B, y{:}\sigma \vdash \{y/x\}\,M{:}\tau]\!]\,\eta[d/y]) \\
&\quad = [\![B \vdash \lambda y{:}\sigma.\{y/x\}\,M]\!]\eta.
\end{aligned}$$

The second equation follows from the substitution lemma 10(i). The soundness of (α) for type variables is proved similarly using Lemma 10(ii).

For (β), consider any term $B \vdash (\lambda y{:}\sigma.M)\,N{:}\tau$ with types $a, b \in \mathrm{Kind}^T$ the meanings of σ and τ as above. We have

$$\begin{aligned}
&[\![B \vdash (\lambda x{:}\sigma M)\,N{:}\tau]\!]\eta \\
&\quad = \Phi_{a,b}(\Phi_{a,b}^{-1}(\lambda d \in D_a.\, [\![B, x{:}\sigma \vdash M{:}\tau]\!]\,\eta[d/x]))[\![B \vdash N{:}\sigma]\!]\eta \\
&\quad = [\![B, x{:}\sigma \vdash M{:}\tau]\!]\,\eta[[\![B \vdash N{:}\sigma]\!]\,\eta/x] \\
&\quad = [\![B \vdash \{N/x\}\,M{:}\tau]\!]\eta,
\end{aligned}$$

using Lemma 10(i). The soundness of (β) for types is proved similarly. The extensionality axioms (η) depend on the fact that $\Phi_{a,b}$ and Φ_f are bijections. It is easy to prove that semantic equality is an equivalence relation. The only subtlety in the (cong) and (ξ) rules are in $(\mathrm{cong})_2$ and $(\xi)_1$, where we must use Lemma 8 to account for the typing differences. As mentioned earlier, rule (remove hyp) relies on our assumption that no Dom^a is empty. The remaining rules are straightforward. ∎

We now show that the axioms and inference rules are complete for environment models without empty types.

THEOREM 13 (Completeness). *Let Γ be a second-order theory over terms with constants from $\mathscr{C}_{\mathrm{cst}}$ and $\mathscr{C}_{\mathrm{term}}$. There is an environment model $\mathscr{F}$ for $\mathscr{C}_{\mathrm{cst}}$ and $\mathscr{C}_{\mathrm{term}}$ such that $\mathscr{F} \models (B \vdash M = N{:}\sigma)$ iff $B \vdash M = N{:}\sigma \in \Gamma$.*

Proof. The proof uses a term model construction as in (Barendregt, 1984; Friedman, 1975; Meyer, 1982). We begin by defining a kind frame $\mathit{Kind} = \langle \{\mathrm{Kind}^\kappa\}, \{\Phi_{\kappa_1,\kappa_2}\}, \mathscr{I} \rangle$ for $\mathscr{C}_{\mathrm{cst}}$. Let *Kind* be the "term model" for $\mathscr{C}_{\mathrm{cst}}$ built from equivalence classes of constructors as in (Friedman, 1975). Thus Kind^T is the set of equivalence classes of type expressions. We will use $\langle \mu \rangle$ to denote the equivalence class of the constructor μ. As usual, the interpretation of a constant $c \in \mathscr{C}_{\mathrm{cst}}$ is its equivalence class $\langle c \rangle$. In particular $\mathscr{I}(\forall) = \langle \forall \rangle$ and $\mathscr{I}(\to) = \langle \to \rangle$. An inductive argument, sketched in

the proof of the *Claim* below, shows that *Kind* is a kind environment model.

We will define *Dom* using equivalence classes of terms. We will start with infinitely many variables of each type, since this will make it possible to prove extensionality of *Dom* quite easily. Let A be an infinite "type assignment" $A = \{x_1 : \sigma_1, \ldots\}$ assigning each variable a single type and providing infinitely many variables of each type. Although the infinite set A is not a syntactic type assignment, we will abuse notation slightly and write $A \vdash M : \sigma$ to mean that $A_1 \vdash M : \sigma$ for some finite subset $A_1 \subseteq A$.

We now define $\mathrm{Dom}^{\langle\sigma\rangle}$ for each equivalence class $\langle\sigma\rangle$, using sets of terms proved equal by Γ. For any $A \vdash M : \sigma$, let $\langle M\rangle$ denote the equivalence class

$$\langle M\rangle = \{N \mid A \vdash M = N : \sigma \in \Gamma\}.$$

and for each $\langle\sigma\rangle \in \mathrm{Kind}^T$, let

$$\mathrm{Dom}^{\langle\sigma\rangle} = \{\langle M\rangle \mid A \vdash M : \sigma\}.$$

Note that by choice of A, no $\mathrm{Dom}^{\langle\sigma\rangle}$ is empty. In addition, by Lemma 2, $\mathrm{Dom}^{\langle\sigma\rangle}$ depends only on the equivalence class $\langle\sigma\rangle$, not the type expression σ. We define $\mathscr{I}$ by interpreting each constant $c \in \mathscr{C}_{\mathrm{term}}$ as its equivalence class $\mathscr{I}_{Dom}(c) = \langle c\rangle$. It remains to define the families of functions $\{\Phi_{a,b}\}$ and $\{\Phi_f\}$.

For each $\langle\sigma\rangle, \langle\tau\rangle \in \mathrm{Kind}^T$, define $\Phi_{\langle\sigma\rangle,\langle\tau\rangle}$ by

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

for all M and N of the appropriate types. Let $[\mathrm{Dom}^{\langle\sigma\rangle} \to \mathrm{Dom}^{\langle\tau\rangle}]$ be the range of $\Phi_{\langle\sigma\rangle,\langle\tau\rangle}$. The function $\Phi_{\langle\sigma\rangle,\langle\tau\rangle}$ is well defined by (cong) and can be shown to be one-to-one using $(\xi)_1$ and (η).

For each $\langle\mu\rangle \in \mathrm{Kind}^{T \Rightarrow T}$ define $\Phi_{\langle\mu\rangle}$ by

$$(\Phi_{\langle\mu\rangle}\langle M\rangle)\langle\tau\rangle = \langle M\tau\rangle$$

for every $A \vdash M : \forall\mu$ and $\langle\tau\rangle \in \mathrm{Kind}^T$. We take $[\Pi_{\langle\gamma\rangle \in \mathrm{Kind}^T}\mathrm{Dom}^{\langle\mu g\rangle}] \subseteq \Pi_{\langle\gamma\rangle \in \mathrm{Kind}^T}\mathrm{Dom}^{\langle\mu g\rangle}$ to be the range of $\Phi_{\langle\mu\rangle}$ and note that $\Phi_{\langle\mu\rangle}$ is one-to-one by $(\xi)_2$ and (η). Thus we have a frame $\mathscr{F} = \langle Kind, Dom, \{\Phi_{a,b}\}, \{\Phi_f\}\rangle$ for terms over constants from $\mathscr{C}_{\mathrm{cst}}$ and $\mathscr{C}_{\mathrm{term}}$.

It remains to show that $\mathscr{F}$ is an environment model and that $\mathscr{F}$ satisfies precisely the equations belonging to Γ. We will show that $\mathscr{F}$ is a model by giving an explicit description of the meaning of every constructor and term. If η is any environment for $\mathscr{F}$, we let $\{\eta\}$ be any substitution of construc-

tor expressions for constructor variables and terms for ordinary variables such that

$$\{\eta\}v \in \eta(v) \qquad \text{and} \qquad \{\eta\}x \in \eta(x)$$

for every constructor variable v and ordinary variable x. Although the value of $\{\eta\}x$ is not uniquely determined, the equivalence class $\langle\{\eta\}x\rangle$ is uniquely determined by η. This is sufficient, since we will only be concerned with the effect of $\{\eta\}$ up to provable equivalence from Γ. It is easy to verify that if $\langle M\rangle = \langle N\rangle$, then $\langle\{\eta\}M\rangle = \langle\{\eta\}N\rangle$. The proof that $\mathscr{F}$ is a model satisfying precisely the equations in Γ rests on the following claim.

CLAIM. *For any constructor* μ, *term* $B \vdash M:\sigma$, *and any environment* $\eta \models B$, *we have*

$$[\![\mu]\!]\eta = \langle\{\eta\}\mu\rangle \qquad \textit{and} \qquad [\![B \vdash M:\sigma]\!]\eta = \langle\{\eta\}M\rangle.$$

If we can verify the claim, then it is clear that $\mathscr{F}$ is an environment model; i.e., every constructor and term has a meaning in $\mathscr{F}$. We can also use the claim to show that $\mathscr{F}$ satisfies precisely the equations in Γ, as follows. First, suppose $B \vdash M = N:\sigma \in \Gamma$ and $\eta \models B$. Since $\eta \models B$, we have $A \vdash \{\eta\}x:\{\eta\}\tau$ for every $x:\tau \in B$, and so by Lemma 3, it follows that $A \vdash \{\eta\}M:\{\eta\}\sigma$, and similarly for N. By Lemma 5, it follows that

$$A \vdash \{\eta\}M = \{\eta\}N:\{\eta\}\sigma \text{ is provable from } B \vdash M = N:\sigma \in \Gamma.$$

Therefore $[\![B \vdash M:\sigma]\!]\eta = \langle\{\eta\}M\rangle = \langle\{\eta\}N\rangle = [\![B \vdash N:\sigma]\!]\eta$. Conversely, we must show that if $\mathscr{F} \models B \vdash M = N:\sigma$, then $B \vdash M = N:\sigma \in \Gamma$. For any B, we can choose an environment $\eta_0 \models B$ which maps every constructor variable v^κ to its equivalence class $\langle v^\kappa\rangle$ and maps every ordinary variable x with $x:\tau \in B$ to the equivalence class of some variable y with $y:\tau \in A$. If $\mathscr{F} \models B \vdash M = N:\sigma$, then we have

$$A \vdash \{\eta_0\}M = \{\eta_0\}N:\sigma \in \Gamma.$$

Since η_0 just renames ordinary variables. M is a substitution instance of $\{\eta_0\}M$ and similarly for N. Therefore, by Lemma 5, we conclude $B \vdash M = N:\sigma \in \Gamma$. This proves the theorem, except for the claim.

We verify the claim using induction on constructor expressions and induction on typing derivations for terms. It is not hard to show $[\![\mu]\!]\eta = \langle\{\eta\}\mu\rangle$ by induction on constructors, using essentially the same steps we use for terms below. In particular, we have $[\![\sigma]\!]\eta = \langle\{\eta\}\sigma\rangle$ for type expressions σ. We now consider terms. For any typing $x:\sigma \vdash x:\sigma$ of a variable, we have

$$[\![x:\sigma \vdash x:\sigma]\!]\eta = \eta(x) = \langle\{\eta\}x\rangle.$$

The application case $(\to E)$ is also straightforward. For any $B \vdash MN:\tau$ typed using $(\to E)$ as the last step, we have

$$\begin{aligned}[\![B \vdash MN:\tau]\!]\eta &= (\Phi_{a,b}[\![B \vdash M:\sigma \to \tau]\!]\eta)[\![B \vdash N:\sigma]\!]\eta \\ &= (\Phi_{a,b}\langle\{\eta\}M\rangle)\langle\{\eta\}N\rangle \\ &= \langle\{\eta\}\ M\{\eta\}N\rangle \\ &= \langle\{\eta\}(MN)\rangle,\end{aligned}$$

where $a = [\![\sigma]\!]\eta$ and $b = [\![\tau]\!]\eta$. For λ-abstractions typed by $(\to I)$, we have

$$[\![B \vdash \lambda x:\sigma.M:\sigma \to \tau]\!]\eta = \Phi_{a,b}^{-1}g,$$

where $a = [\![\sigma]\!]\eta$, $b = [\![\tau]\!]\eta$ and g is the function satisfying

$$g(d) = [\![B, x:\sigma \vdash M:\tau]\!]\,\eta[d/x]$$

for all $d \in \mathrm{Dom}^a$. We can see that $g = (\Phi_{a,b}\langle\{\eta\}\ \lambda x:\sigma.M\rangle)$ using the inductive hypothesis and the substitution lemma as follows. For any $\langle N\rangle \in \mathrm{Dom}^a$, we have

$$\begin{aligned}g\langle N\rangle &= [\![B, x:\sigma \vdash M:\tau]\!]\,\eta[\langle N\rangle/x] \\ &= \langle\{\eta[\langle N\rangle/x]\}M\rangle \\ &= \langle(\{\eta\}\ \lambda x:\sigma.M)N\rangle \\ &= (\Phi_{a,b}\langle\{\eta\}\ \lambda x:\sigma.M\rangle)\langle N\rangle.\end{aligned}$$

Since $\Phi_{a,b}^{-1}$ is the inverse of $\Phi_{a,b}$, it follows easily that

$$\begin{aligned}&[\![B \vdash \lambda x:\sigma.M:\sigma \to \tau]\!]\eta \\ &\quad = \Phi_{a,b}^{-1}(\Phi_{a,b}\langle\{\eta\}\ \lambda x:\sigma.M\rangle) \\ &\quad = \langle\{\eta\}\ \lambda x:\sigma.M\rangle.\end{aligned}$$

Similar arguments demonstrate the claim for the $(\forall E)$ and $(\forall I)$ cases, and rules (add hyp) and (type eq) are trivial. This finishes the proof of the claim and hence the theorem. ∎

6. Combinatory Algebras and Models

6.1. *Introduction*

In this section, we present an alternative to the environment model. The environment model definition has two parts: the definition of a frame and

the stipulation that a frame $\mathscr{F}$ is a model only if every term has a meaning in $\mathscr{F}$. While the definition of frame has the same mathematical flavor as, say, the definition of a group or vector space, the condition distinguishing environment models from frames is largely syntactic since it relies on the inductive definition of the meanings of terms. In this section, we present an equivalent "combinatory model" definition based on algebraic properties of elements. Since second-order combinatory models are analogous to untyped combinatory models, we will illustrate the basic ideas by reviewing the untyped definitions (see Barendregt, 1984; Meyer, 1982 for further discussion).

An applicative structure is said to be combinatorially complete if every implicitly definable function is represented in the model. More precisely, an untyped applicative structure $\mathscr{D} = \langle D, \Phi \rangle$ is *combinatorially complete* if, for every expression M with no occurrence of λ, all variables among $x_1, ..., x_n$, and possibly containing constants from D, there is a constant $d \in D$ such that

$$\mathscr{D} \models M = dx_1 \cdots x_n.$$

Intuitively, this means that for every implicit "polynomial" description M of a function of n variables, there is an element of D representing this function. For untyped extensional applicative structures, it can be shown that combinatory completeness is equivalent to the environment model condition that every term have a meaning (Barendregt, 1984; Meyer, 1982).

Combinatory completeness also has a relatively simple definition which is algebraic in nature. An untyped applicative structure $\mathscr{D} = \langle D, \Phi \rangle$ is a *combinatory algebra* if it has elements[4] $K, S \in D$ satisfying

$$Kxy = x$$

$$Sxyz = (xz)(yz)$$

for all $x, y, z \in D$. It can be shown that the combinators K and S are an "algebraic basis" for the implicitly definable functions. Consequently, an applicative structure $\mathscr{D}$ is combinatorially complete iff $\mathscr{D}$ is a combinatory algebra (Barendregt, 1984; Koymans, 1982; Lambek, 1980; Meyer, 1982). Since both K and S can be defined explicitly by lambda terms, an extensional applicative structure $\mathscr{D}$ is an untyped environment model iff $\mathscr{D}$ is an untyped combinatory algebra.

[4] Combinatory algebras are often defined as structures interpreting constants K and S such that $K^{\mathscr{D}}$ and $S^{\mathscr{D}}$ satisfy the equations above. If $\mathscr{D}$ is not extensional, then the equations may hold for many $K, S \in D$ and it may be useful to have a structure $\mathscr{D}$ single out specific K and S. Since our structures will be extensional, we will not require combinatory algebras to choose specific combinators.

Combinators for ordinary typed lambda calculus are similar to the untyped combinators, as we shall see in the discussion of kind structures below. Instead of using two untyped combinators K and S, typed combinatory algebras are characterized using an infinite family of typed K combinators and a similar family of typed S combinators.

In the discussion of second-order combinatory models, we will define second-order combinatory completeness and second-order combinatory algebras. Instead of infinite collections of typed K and S combinators, second-order combinatory algebras will be characterized using a single polymorphic K, a single polymorphic S, and infinite families of additional combinators. As in the untyped case, each combinator is characterized by an equation. We will see that every second-order lambda-definable function can be viewed as an applicative combination of the combinators and show that an extensional frame $\mathscr{F}$ is a second-order environment model iff $\mathscr{F}$ is a second-order combinatory algebra.

It is worth pointing out that the situation becomes more complicated if we do not assume extensionality. The correspondence between combinatory completeness and combinatory algebras holds in general, but nonextensional combinatory models are more complicated than nonextensional combinatory algebras (see Barendregt, 1984; Koymans, 1982; Meyer, 1982). Except for a brief discussion in Section 6.5, we will only consider extensional second-order frames.

6.2. *Constructor Combinators*

Recall that a kind environment model is a kind frame,

$$Kind = \langle \{\mathrm{Kind}^{\kappa} \mid \kappa \text{ a kind}\}, \{\Phi_{\kappa_1, \kappa_2} \mid \kappa_1, \kappa_2 \text{ kinds}\}, \mathscr{I} \},$$

in which every constructor has a meaning. As the first step towards giving a model definition that does not rely on the meanings of terms, we will substitute a condition involving "kinded" combinators K and S.

As described in (Barendregt, 1984; Friedman, 1975), the requirement that every constructor expression has a meaning in *Kind* is equivalent to stipulating that for all kinds κ_1, κ_2, and κ_3, there must be elements

$$K_{\kappa_1, \kappa_2} \in \mathrm{Kind}^{\kappa_1 \Rightarrow (\kappa_2 \Rightarrow \kappa_1)}$$

$$S_{\kappa_1, \kappa_2, \kappa_3} \in \mathrm{Kind}^{(\kappa_1 \Rightarrow \kappa_2 \Rightarrow \kappa_3) \Rightarrow (\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_3}$$

with the familiar properties

$$K_{\kappa_1, \kappa_2}\, uv = u$$

$$S_{\kappa_1, \kappa_2, \kappa_3}\, uvw = (uw)(vw)$$

for all u, v, and w of the appropriate kinds. (As usual, we have abbreviated

$(\Phi_{\kappa_1,\kappa_2}(\Phi_{\kappa_1,\kappa_2\Rightarrow\kappa_1}K_{\kappa_1,\kappa_2})u)v$ to $K_{\kappa_1,\kappa_2}uv$, and similarly for $S_{\kappa_1,\kappa_2,\kappa_3}$.) In the following discussion of frames and combinators, we will assume that every kind frame has combinators K_{κ_1,κ_2} and $S_{\kappa_1,\kappa_2,\kappa_3}$ for all kinds κ_1, κ_2, and κ_3. This will allow us to focus on combinators for terms.

6.3. *Second-Order Combinatory Completeness*

Intuitively, a second-order frame is second-order combinatorially complete if it is closed under definition by polynomials over ordinary variables and type variables. We will show that combinatory completeness is equivalent to the existence of a set of combinators, each characterized by an equational axiom. Later, in Section 6.4, we will see how to describe combinators formally without introducing extra constants into the language. However, constants for elements will be convenient for discussing combinatory completeness and for proving the equivalence of combinatory and environment model conditions.

If $\mathscr{F}$ is a second-order frame, then the *$\mathscr{F}$-terms* are the applicative terms (terms without $\lambda x{:}\sigma.M$ or $\lambda t.M$) of the language with a constant for each element of $\mathscr{F}$. It is understood that if c is the "constant for d," then $\mathscr{I}(c)=d$. Since the $\mathscr{F}$-terms do not involve any lambda abstraction, every $\mathscr{F}$ term has a meaning in the frame $\mathscr{F}$, regardless of whether $\mathscr{F}$ is an environment model. One minor complication with the $\mathscr{F}$-terms is that the syntactic type of an element is not determined uniquely (since our syntax does not allow arbitrary equations between constructors). For example, if $f, g\in \mathrm{Kind}^{T\Rightarrow T}$ are distinct, but $\forall f=\forall g$ are the same element of Kind^{T}, then a constant c for $d\in\mathrm{Dom}^{\forall f}$ could be given syntactic type $\forall f$ or $\forall g$. We will take the rather brute force approach of assuming we have many constants for each element, one for each equivalence class of type expressions over constants from *Kind*. It is important to have each typing included, since the syntactic type of a constant determines the way the constant may be used in terms.

A frame $\mathscr{F}=\langle \mathit{Kind}, \mathit{Dom}, \{\Phi_{a,b}\}, \{\Phi_f\}\rangle$ is *second-order combinatorially complete* if for every $\mathscr{F}$-term $B\vdash M{:}\sigma$ without free variables of higher kinds (kinds other than T) there is a constant d from $\mathscr{F}$ such that

$$\mathscr{F}\models B\vdash M=d\vec{s}\vec{x}{:}\sigma,$$

where $\vec{x}$ is a list of all ordinary variables in B and $\vec{s}$ lists all type variables of $B\vdash M{:}\sigma$. This definition is similar to the usual definition of combinatory completeness for untyped lambda calculus (Barendregt, 1984; Meyer, 1982), but with the added consideration of types and type variables. We do not consider implicit functions of variables of higher kinds since we cannot λ-bind variables of higher kinds in second-order terms.

We will see that a second-order frame is combinatorially complete iff it

has elements K, S, A, B, C, and D satisfying certain equational axioms. Since these elements may be defined as the meanings of closed terms (in a language with enough constructor constants to write down their types), they are called combinators. The combinators K and S are similar to the combinators of the same names used in untyped lambda calculus, while A, B, C, and D are related to type application and type abstraction. A useful abbreviation is to write $T^i \Rightarrow T$ for the kind $T \Rightarrow \cdots \Rightarrow T$ with $i+1$ occurences of T.

A second-order frame $\mathscr{F}$ is a *second-order combinatory algebra* if it contains elements

$$K \in \mathrm{Dom}^{\forall s.\forall t.s \to t \to s}$$

$$S \in \mathrm{Dom}^{\forall r.\forall s.\forall t.(r \to s \to t) \to (r \to s) \to r \to t}$$

and, for all integers $i, j, k \geqslant 0$ and all $f \in \mathrm{Kind}^{T^{i+1} \Rightarrow T}$, $g \in \mathrm{Kind}^{T^{j+1} \Rightarrow T}$, and $h \in \mathrm{Kind}^{T^{k+2} \Rightarrow T}$, elements

$$A_f \in \mathrm{Dom}^{\forall r.\forall \vec{s}[(r \to \forall t.f\vec{s}t) \to \forall t.(r \to f\vec{s}t)]}$$

$$B \in \mathrm{Dom}^{\forall r[r \to \forall t.r]}$$

$$C_{f,g} \in \mathrm{Dom}^{\forall \vec{r}.\forall \vec{s}[\forall t[(f\vec{r}t) \to (g\vec{s}t)] \to \forall t(f\vec{r}t) \to \forall t(g\vec{s}t)]}$$

$$D_{h,f} \in \mathrm{Dom}^{\forall \vec{r}.\forall \vec{s}[\forall t.\forall u(h\vec{r}tu) \to \forall t(h\vec{r}t(f\vec{s}t))]}$$

with the properties described below.

The combinators K and S must satisfy

$$Kstxy = x$$

$$Srstxyz = xz(yz)$$

for all types r, s, t and all elements x, y, and z of the appropriate types. The types of x, y, z, omitted to improve readability, are easily determined from the types of the combinators. For example, to be more specific, K must satisfy

$$Kstxy = x \qquad \text{for all} \quad s, t \in \mathrm{Kind}^T \text{ and all } x \in \mathrm{Dom}^s,\ y \in \mathrm{Dom}^t.$$

The combinator B and every A, C, and D must have the following equational properties, where again the types of x and y may be determined from the types given above:

$$(A_f\vec{r}\vec{s})\,xty = (xy)t$$

$$(Br)\,xt = x$$

$$(C_{f,g}\vec{r}\vec{s})\,xyt = xt(yt)$$

$$(D_{h,f}\vec{r}\vec{s})\,xt = xt(f\vec{s}t).$$

It is worth noting that this set of combinators has been chosen for ease in proofs and is not intended to be minimal.

We now introduce a language for describing elements of combinatory algebras. One complicating feature of second-order lambda calculus, mentioned briefly earlier, is that we cannot necessarily write closed expressions for all of the types of any given frame. Similarly, we cannot necessarily define all of the elements of each kind $T^i \Rightarrow T$. Since we need a closed expression μ for $f \in \mathrm{Kind}^{T^i \Rightarrow T}$ to make use of a constant for combinator A_f, the set of constructor constants of a language limits our ability to add combinator constants. Therefore, we will have to pay particularly close attention the set $\mathscr{C}_{\mathrm{cst}}$ of constructor constants in the following discussion. To describe the connection between lambda terms and combinators as generally as possible, we will use languages which contain as many combinator constants as the constructor expressions will allow.

For any set $\mathscr{C}_{\mathrm{cst}}$ of constructor constants and set $\mathscr{C}_{\mathrm{term}}$ of term constants, the *combinatory terms* $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$ are the applicative second-order terms over constants $\mathscr{C}_{\mathrm{cst}}$, $\mathscr{C}_{\mathrm{term}}$, and additional fresh constants for the combinators K, S, A, B, C, and D. Specifically, in addition to the constructor constants $\mathscr{C}_{\mathrm{cst}}$ and term constants $\mathscr{C}_{\mathrm{term}}$, the language $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$ has fresh constants

$$K : \forall s.\forall t.s \to t \to s$$

$$S : \forall r.\forall s.\forall t.(r \to s \to t) \to (r \to s) \to r \to t$$

and, for all closed constructor expressions $\mu : T^{i+1} \Rightarrow T$, $\nu : T^{j+1} \Rightarrow T$, and $\pi : T^{k+2} \Rightarrow T$ of $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$, constants A_μ, B, $C_{\mu,\nu}$, and $D_{\pi,\mu}$. Note that, as described above, the set of combinator constants depends on the set of closed constructor expressions. A special case of particular interest are the $\mathscr{CL}(\mathscr{F})$ terms, which are the combinatory terms over the language with a constant for every element of every Kind^κ and Dom^a of $\mathscr{F}$.

A model for combinatory terms will be called a combinatory frame. More precisely, a *second-order combinatory frame for* $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$ is a frame $\mathscr{F}$ for the constants of $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$ such that the combinator equations hold for all of the combinator constants in the language $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$. There are two differences between combinatory algebras and combinatory frames. The first is that a combinatory frame interprets constants of some $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$ with combinator constants, while a combinatory algebra need not interpret any combinator constants. The second difference is that a combinatory frame need only have those A_f, $C_{f,g}$ and $D_{h,f}$ which have f, g, h definable in $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$. If $f \in \mathrm{Kind}^{T^i \Rightarrow T}$ is not the meaning of any closed constructor expression of $\mathscr{CL}(\mathscr{C}_{\mathrm{cst}}, \mathscr{C}_{\mathrm{term}})$, then a combinatory frame might not have an element A_f satisfying the associated equation. Essentially, combinatory algebras are frames with a

set of semantic properties, while combinatory frames are frames that interpret certain languages in a certain way. The two notions are very similar, however, if we add enough constants to our language. Specifically, if $\mathscr{F}$ is a combinatory frame for $\mathscr{CL}(\mathscr{F})$, then every combinator of $\mathscr{F}$ has a constant in $\mathscr{CL}(\mathscr{F})$, and so $\mathscr{F}$ must be a combinatory algebra. Conversely, it follows easily from the definitions that any combinatory algebra $\mathscr{F}$ can be extended to a combinatory frame for $\mathscr{CL}(\mathscr{F})$ by interpreting the combinator constants of $\mathscr{CL}(\mathscr{F})$ appropriately. In the proof of the combinatory model theorem, we will use $\mathscr{CL}(\mathscr{F})$. However, we will use more general combinatory terms $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ in proving a general equivalence between lambda terms and combinatory terms.

We will now justify the name "combinatory algebra" by showing that every combinatory algebra is combinatorially complete. It is clear that every combinatorially complete frame is a combinatory algebra, since each combinator is defined by an equation involving variables and $\mathscr{F}$-terms. To show how combinators allow us to represent functions, we define "pseudo-abstraction" for ordinary variables and type variables. For every combinatory term $B, x{:}\sigma \vdash M{:}\tau$ without free variables of higher kinds, we define the combinatory term $B \vdash \langle x{:}\sigma\rangle M{:}\sigma \to \tau$ using induction on the derivation of typing $B, x{:}\sigma \vdash M{:}\tau$ as follows. We omit the trivial cases for (add hyp) and (type eq). Since B and $\sigma \to \tau$ are clear, we only specify $\langle x{:}\sigma\rangle M$:

$$\langle x{:}\sigma\rangle x = S\sigma(\sigma\to\sigma)\sigma(K\sigma(\sigma\to\sigma))(K\sigma\sigma),$$

$$\langle x{:}\sigma\rangle y = K\tau\sigma y, \qquad \text{where } \tau \text{ is the type of } y \text{ and } y \text{ is different from } x,$$

$$\langle x{:}\sigma\rangle c = K\tau\sigma c, \qquad \text{where } \tau \text{ is the type of constant } c,$$

$$\langle x{:}\sigma\rangle(MN) = S\sigma\rho\tau(\langle x{:}\sigma\rangle M)(\langle x{:}\sigma\rangle N), \qquad \text{where} \quad B, x{:}\sigma \vdash M{:}\rho\to\tau,$$

$$\langle x{:}\sigma\rangle(M\rho) = (A_f\sigma\vec{s})(\langle x{:}\sigma\rangle M)\rho,$$

where $B, x{:}\sigma \vdash M{:}\forall\mu$, all free variables of μ are among $\vec{s}$, and f is the closed constructor $\lambda\vec{s}.\lambda t.\tau$.

The definition of $\langle x{:}\sigma\rangle x$ is analogous to the usual untyped translation into combinators $\langle x\rangle x = SKK$. If $B \vdash M{:}\tau$ is well typed and t does not occur free in B, we define $B \vdash \langle t\rangle M{:}\forall t.\tau$ by induction on derivation of typing $B \vdash M{:}\tau$ as follows. Again, we omit the trivial cases and only specify $\langle t\rangle M$:

$$\langle t\rangle y = B\tau y, \qquad \text{where } \tau \text{ is the type of } y,$$

$$\langle t\rangle c = B\tau c, \qquad \text{where } \tau \text{ is the type of constant } c,$$

$$\langle t\rangle(MN) = C_{f,g}\vec{r}\vec{s}(\langle t\rangle M)(\langle t\rangle N),$$

where f and g are determined by the typing of MN as follows. If $B \vdash M:\sigma \to \tau$ and $B \vdash N:\sigma$, let $\vec{r}$ and $\vec{s}$ be lists of type variables so that $f \equiv \lambda\vec{r}.\lambda t.\sigma$ and $g \equiv \lambda\vec{s}.\lambda t.\tau$ are closed;

$$\langle t \rangle (M\tau) = D_{h,f}\vec{r}\vec{s}(\langle t \rangle M)$$

where f and h are determined from τ and the typing $B \vdash M:\forall u.\sigma$ by taking lists $\vec{r}$ and $\vec{s}$ of type variables so that $f \equiv \lambda\vec{s}.\lambda t.\tau$ and $h \equiv \lambda\vec{r}.\lambda t.\lambda u.\sigma$ are closed constructor expressions.

It is not hard to verify that $B \vdash \langle x:\sigma \rangle M:\sigma \to \tau$ and $B \vdash \langle t \rangle M:\forall t.\tau$ are well typed. The assumption that M has no free variables of higher kinds is needed to show that, for example, f and g in the definition of $\langle t \rangle (MN)$ may be closed. (If this were not possible, we could not give $C_{f,g}$ a closed type, and so $C_{f,g}$ would not really be constant.) The essential properties of pseudo-abstraction are described by the following lemma.

LEMMA 14. *Let $\mathscr{F}$ be a second-order combinatory frame for $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$. For any combinatory terms $B, x:\sigma \vdash M:\tau$ and $B \vdash N:\sigma$ of $\mathscr{CL}_A(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ without free variables of higher kinds, we have*

$$\mathscr{F} \models B \vdash (\langle x:\sigma \rangle M)N = \{N/x\}\, M:\tau.$$

Similarly, if $B \vdash M:\tau$ with t not free in B and σ is any type expression without free variables of higher kinds we have

$$\mathscr{F} \models B \vdash (\langle t \rangle M)\sigma = \{\sigma/t\}\, M:(\lambda t.\tau)\sigma.$$

The lemma is proved by an easy induction. Using Lemma 14 we can prove the following combinatory completeness theorem.

THEOREM 15 (Combinatory completeness). *A frame $\mathscr{F}$ is second-order combinatorially complete iff $\mathscr{F}$ is a second-order combinatory algebra.*

Proof. If $\mathscr{F} = \langle Kind, Dom, \{\Phi_{a,b}\}, \{\Phi_f\} \rangle$ is second-order combinatorially complete, then $\mathscr{F}$ is a combinatory algebra since each combinator is defined by a polynomial over $\mathscr{F}$. Conversely, suppose $\mathscr{F}$ is a combinatory algebra and let $\{x_1:\sigma_1, ..., x_j:\sigma_j\} \vdash M:\sigma$ be any $\mathscr{F}$-term whose free type variables are among $s_1, ..., s_i$. Using Lemma 14, it is easy to show that

$$N = \langle s_1 \rangle \cdots \langle s_i \rangle \langle x_1:\sigma_1 \rangle \cdots \langle x_j:\sigma_j \rangle M$$

is a closed term of $\mathscr{CL}(\mathscr{F})$ with

$$\mathscr{F} \models \{x_1:\sigma_1, ..., x_j:\sigma_j\} \vdash M = Ns_1 \cdots s_i x_1 \cdots x_j:\sigma.$$

Thus $\mathscr{F}$ is combinatorially complete. ∎

6.4. *Combinatory Models*

In this section, we show that an extensional frame $\mathscr{F}$ is an environment model iff $\mathscr{F}$ is a combinatory algebra. This will establish that the "algebraic" definition of combinatory algebra is equivalent to the syntactic condition in the environment model definition. We will use translations CL and LAM between lambda terms and combinatory terms over the same sets of constants, which may also be of independent interest.

Let $B \vdash M:\sigma$ be a second-order lambda term over constants from $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$, without free variables of higher kinds. We define the combinatory term $B \vdash \text{CL}(M):\sigma$ of $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ by induction on the derivation of $B \vdash M:\sigma$. As usual, the trivial cases (add hyp) and (type eq) are omitted, and, since B and σ are already determined, we mention only $\text{CL}(M)$.

$$\text{CL}(x) = x$$

$$\text{CL}(MN) = \text{CL}(M)\ \text{CL}(N)$$

$$\text{CL}(\lambda x{:}\sigma.M) = \langle x{:}\sigma \rangle\ \text{CL}(M)$$

$$\text{CL}(M\sigma) = \text{CL}(M)\sigma$$

$$\text{CL}(\lambda t.M) = \langle t \rangle\ \text{CL}(M).$$

We can use Lemma 14 to show

LEMMA 16. *Suppose $\mathscr{F}$ is an extensional combinatory frame for $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ and $B \vdash M:\sigma$ is a second-order lambda term over $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$ without free variables of higher kinds. If $\eta \models B$, then the meaning of $B \vdash M:\sigma$ exists in $\mathscr{F}$ and is given by*

$$[\![B \vdash M:\sigma]\!]\eta = [\![B \vdash \text{CL}(M):\sigma]\!]\eta.$$

We will use the lemma later to show that every combinatory model is an environment model. In doing so, we will eliminate the restriction on free variables of higher kinds.

Proof. The lemma is proved by induction on the typing of terms. The only nontrivial cases are $(\to I)$ and $(\forall I)$. Since these two cases are similar, we only consider the first. Recall that the meaning of $B \vdash \lambda x{:}\sigma.M:\sigma \to \tau$ typed by $(\to I)$ is

$$[\![B \vdash \lambda x{:}\sigma.M:\sigma \to \tau]\!]\eta = \Phi^{-1}_{a,b}\, g,$$

where $g(d) = [\![B, x{:}\sigma \vdash M:\tau]\!]\,\eta[d/x]$ for all $d \in \text{Dom}^a$ and a, b are the meanings of σ and τ in η.

By the inductive hypothesis,

$$g(d) = [\![B, x{:}\sigma \vdash \text{CL}(M){:}\tau]\!]\ \eta[d/x] \qquad \text{for all} \quad d \in \text{Dom}^a.$$

By the substitution lemma and Lemma 14,

$$(\Phi_{a,b}[\![B \vdash \langle x{:}\sigma \rangle\ \text{CL}(M){:}\sigma \to \tau]\!]\ \eta)d = [\![B, x{:}\sigma \vdash \text{CL}(M){:}\tau]\!]\ \eta[d/x] = g(d)$$

for all $d \in \text{Dom}^a$. Therefore,

$$[\![B \vdash \langle x{:}\sigma \rangle\ \text{CL}(M){:}\sigma \to \tau]\!]\ \eta = \Phi_{a,b}^{-1}\ g,$$

proving the lemma. ∎

We now show how to translate combinatory terms into lambda terms. For any combinatory term $B \vdash M{:}\tau$ of $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$, we define the lambda term $B \vdash \text{LAM}(M){:}\tau$ over $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$ as follows:

$$\begin{aligned}
\text{LAM}(x) &= x \\
\text{LAM}(c) &= c \qquad \text{for} \quad c \in \mathscr{C}_{\text{term}} \\
\text{LAM}(K) &= \lambda s.\lambda t.\lambda x{:}s.\lambda y{:}t.x \\
\text{LAM}(S) &= \lambda r.\lambda s.\lambda t.\lambda x{:}r \to s \to t.\lambda y{:}r \to s.\lambda z{:}r.xz(yz) \\
\text{LAM}(A_f) &= \lambda r.\lambda \vec{s}.\lambda x{:}r \to \forall t(f\vec{s}t).\lambda t.\lambda y{:}r.xyt, \\
\text{LAM}(B) &= \lambda r.\lambda x{:}r.\lambda t.x \\
\text{LAM}(C_{f,g}) &= \lambda \vec{r}.\lambda \vec{s}.\lambda x{:}\forall t((f\vec{r}t) \to (g\vec{s}t)).\lambda y{:}\forall t(f\vec{r}t).\lambda t.(xt)(yt), \\
\text{LAM}(D_{h,f}) &= \lambda \vec{r}.\lambda \vec{s}.\lambda x{:}\forall t.\forall u(h\vec{r}tu).\lambda t.xt(f\vec{s}t) \\
\text{LAM}(MN) &= \text{LAM}(M)\ \text{LAM}(N) \\
\text{LAM}(M\sigma) &= \text{LAM}(M)\sigma.
\end{aligned}$$

Note that for combinators indexed by constructors, such as A_f, we have A_f in $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ only if f is a closed constructor expression. Therefore, for every A_f in $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$, the term $\text{LAM}(A_f)$ will be a closed lambda term over $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$. In the special case that $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ is $\mathscr{CL}(\mathscr{F})$, then LAM translates every combinator of $\mathscr{F}$ into a closed lambda term over constructor constants from $\mathscr{F}$.

If $\mathscr{F} = \langle \mathit{Kind}, \mathit{Dom}, \{\Phi_{a,b}\}, \{\Phi_f\} \rangle$ is an environment model for terms over $\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$, then we define $\mathscr{F}^+$ to be the result of extending $\mathscr{I}_{\mathit{Dom}}$ to interpret the fresh combinator constants of $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ as the lambda terms above. It is easy to prove the following lemma.

LEMMA 17. *Let $\mathscr{F}$ be an extensional environment model for terms over*

$\mathscr{C}_{\text{cst}}$ and $\mathscr{C}_{\text{term}}$. Then $\mathscr{F}^+$ is a combinatory frame for $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$ such that for every $B \vdash M:\sigma$ of $\mathscr{CL}(\mathscr{C}_{\text{cst}}, \mathscr{C}_{\text{term}})$, we have

$$\mathscr{F}^+ \models B \vdash M = \text{LAM}(M):\sigma.$$

Using Lemmas 16 and 17, we can now prove the combinatory model theorem. This theorem is analogous to the combinatory model theorem of (Meyer, 1982, but somewhat more simply stated since we have only considered extensional structures.

THEOREM 18 (Combinatory model theorem). *An extensional second-order frame $\mathscr{F}$ is an environment model iff $\mathscr{F}$ is a combinatory algebra.*

Proof. First, suppose $\mathscr{F}$ is an environment model. By Lemma 11, the environment model condition does not depend on the choice of constants, and so we may assume $\mathscr{F}$ is an environment model for lambda terms over constants from $\mathscr{F}$. Therefore, by Lemma 17, $\mathscr{F}^+$ is a combinatory frame for $\mathscr{CL}(\mathscr{F})$, and so $\mathscr{F}$ must be a combinatory algebra.

We now suppose $\mathscr{F}$ is a combinatory algebra and show that $\mathscr{F}$ must be an environment model. To prove this, we must remove the restriction of Lemma 17 to terms without free variables of higher kinds. This will be done by substituting constants for variables. To this end, we first extend $\mathscr{F}$ to be a combinatory frame for $\mathscr{CL}(\mathscr{F})$ with constants for every element of $\mathscr{F}$.

Let $\mathscr{V}$ be any set of constructor variables, not containing any type variables, let η_0 be any environment for $\mathscr{F}$, and let $\mathscr{E}_{\mathscr{V},\eta_0}$ be the class of all environments for $\mathscr{F}$ which agree with η_0 on all variables from $\mathscr{V}$, i.e.,

$$\eta(v) = \eta_0(v) \qquad \text{for all} \quad \eta \in \mathscr{E}_{\mathscr{V},\eta_0} \text{ and } v \in \mathscr{V}.$$

We will say that $B \vdash M:\sigma$ is a $\mathscr{V}$-term if all free variables of $B \vdash M:\sigma$ are ordinary variables, type variables, or variables from $\mathscr{V}$. If $B \vdash M:\sigma$ is a $\mathscr{V}$-term, then let $M_{\mathscr{V},\eta_0}$ by the result of replacing each variable v from $\mathscr{V}$ by the constant for $\eta_0(v)$. By Lemma 16, we know that every $B \vdash M_{\mathscr{V},\eta_0}:\sigma$ has a meaning in $\mathscr{F}$. An easy induction shows that for every $\mathscr{V}$-term $B \vdash M:\sigma$, and every environment $\eta \in \mathscr{E}_{\mathscr{V},\eta_0}$, we have

$$[\![B \vdash M:\sigma]\!]\eta = [\![B \vdash M_{\mathscr{V},\eta_0}:\sigma]\!]\eta.$$

Thus every $\mathscr{V}$-term has a meaning in $\mathscr{F}$. Since every term is a $\mathscr{V}$-term for some $\mathscr{V}$, it follows that $\mathscr{F}$ is an environment model. ∎

6.5. *Second-Order Type Theory*

The combinatory characterization of untyped lambda models shows how to reduce untyped lambda calculus to first-order logic. Specifically, when combined with the extensionality axiom,

$$\forall z(xz = yz) \supset x = y,$$

the combinator axioms

$$\forall x,\ y.Kxy = x,$$

$$\forall x,\ y,\ z.Sxyz = (xz)(yz)$$

provide a first-order axiomatization of extensional models for untyped lambda calculus (Barendregt, 1984; Meyer, 1982). Second-order combinatory algebras and models may also be defined in first-order logic. However, since the details of interpreting second-order lambda calculus in first-order logic are not very enlightening, we will show how to axiomatize combinatory algebras and models in the logical system $\mathscr{S}\mathscr{T}$ of "second-order type theory." This axiomatization is relatively natural since the type structure of $\mathscr{S}\mathscr{T}$ matches that of second-order lambda calculus. By a further reduction of $\mathscr{S}\mathscr{T}$ to first-order logic, which is entirely routine, one may see that the semantics of second-order lambda calculus is reducible to first-order model theory. However, we will not go into the details of the reduction to first-order logic.

The language $\mathscr{S}\mathscr{T}$ is built from applicative second-order terms using equality, the logical connectives, and quantification. To be more precise, an $\mathscr{S}\mathscr{T}$ *atomic formula* is an equation $B \vdash M = N:\sigma$ without lambda binding of ordinary variables or type variables in M or N. If $B \vdash G_1$ and $B \vdash G_2$ are $\mathscr{S}\mathscr{T}$ formulas, then so are

$$B \vdash G_1 \wedge G_2 \qquad \text{and} \qquad B \vdash \neg G_1.$$

In addition, if v^κ does not appear free in B, then

$$B \vdash \forall v^\kappa.G$$

is an $\mathscr{S}\mathscr{T}$ formula. Similarly, if $B, x:\sigma \vdash G$ is an $\mathscr{S}\mathscr{T}$ formula, then

$$B \vdash \forall x:\sigma.G$$

is an $\mathscr{S}\mathscr{T}$ formula as well. Finally, we need an (add hyp) rule for formulas since formulas include type assignments for ordinary variables. We do not need (type eq) for formulas since the types of formulas are not part of the syntax. Formulas of $\mathscr{S}\mathscr{T}$ are interpreted by giving the logical connectives

$\wedge$ and $\neg$ their usual meanings and by interpreting quantifiers as ranging over the appropriate sets of the frame. Since only applicative terms appear in formulas of $\mathscr{ST}$, we can interpret logical formulas over any second-order frame for the appropriate set of constants.

It is easy to read through the definition of combinatory algebra and see that all the combinator axioms may be formalized in $\mathscr{ST}$. In axiomatizing combinatory algebras, we may replace constants by existential quantifiers. For example, K is described by the axiom

$$\exists K{:}\forall s\forall t(s \to t \to s).\forall s.\forall t.\forall x{:}s.\forall y{:}t.Kstxy = x.$$

The axioms for A, C, and D involve variables of higher kinds. In addition, we need infinitely many axioms for each family of combinators. For example, for each $i \geqslant 0$, we need the A axiom

$$\forall f^{T^{i+1} \Rightarrow T}.\exists A_f{:}\forall r, \vec{s}[(r \to \forall t.f\vec{s}t) \to \forall t.(r \to f\vec{s}t)].$$
$$\forall r.\forall \vec{s}.\forall x{:}(r \to \forall t.f\vec{s}t)\,\forall t.\forall y{:}r.(A_f r\vec{s})\,xty = (xy)t.$$

It should be clear that an extensional frame $\mathscr{F}$ satisfies the collection of $\mathscr{ST}$ combinator axioms iff $\mathscr{F}$ is a combinatory algebra. Therefore, we may axiomatize combinatory algebras without introducing constants into the language.

The language $\mathscr{ST}$ may be reduced to first-order logic using a relatively straightforward method, similar to the reduction of ordinary type theory to first-order logic outlined in (Monk, 1976). However, in the reduction to first-order logic, we must be careful to specify that frames are extensional. Essentially, this involves introducing axioms

$$\forall \mu^{T \Rightarrow T}.\forall f{:}(\forall \mu).\forall g{:}(\forall \mu).(\forall t.ft = gt) \supset f = g$$
$$\forall s.\forall t.\forall f{:}s \to t.\forall g{:}s \to t.(\forall x{:}s.fx = gx) \supset f = g$$

to say that elements which have identical functional behavior must be equal. By including typed extensionality axioms and reducing to first-order logic, we can show that second-order lambda calculus is reducible to first-order logic. It is worth emphasizing that, as with other versions of lambda calculus, the first-order axioms are not equational (due here to extensionality), so second-order lambda calculus is not an algebraic theory. One consequence is that the class of second-order lambda models is not closed under homomorphism (cf. Barendregt, 1984; Meyer, 1982).

6.6. *Nonextensional Models*

Throughout this paper, we have emphasized extensional models. Essentially, the extensionality axioms (η) state that if two elements d and e

behave the same way as functions (i.e., if $dx = ex$ for all x of the appropriate type or kind), then d and e must be equal. These axioms are quite reasonable, but nonextensional models are occasionally of interest also. Semantically, the extensionality axiom is reflected in the fact that we have assumed *a bijection* $\Phi_{a,b}$ between $\mathrm{Dom}^{a\to b}$ and $[\mathrm{Dom}_a \to \mathrm{Dom}^b]$, and similarly for each Φ_f. Thus every function $g \in [\mathrm{Dom}^a \to \mathrm{Dom}^b]$ corresponds to precisely one element $\Phi^{-1}(g) \in \mathrm{Dom}^{a\to b}$. In nonextensional models, we let two different elements $d \neq e \in \mathrm{Dom}^{a\to b}$ represent the same function $\Phi_{a,b}(d) = \Phi_{a,b}(e)$. This leads to some complication, since we used Φ^{-1} to find the meanings of lambda abstractions.

In nonextensional models, the main difficulty in interpreting lambda terms becomes the *weak extensionality* property of second-order lambda calculus. Intuitively, weak extensionality states that if M and N both define the same function of $x{:}\sigma$, then $\lambda x{:}\sigma.M$ must equal $\lambda x{:}\sigma.N$, and similarly if M and N both define the same function of t, then $\lambda t.M$ must equal $\lambda t.N$. This is formalized in the inference rules

$$(\xi)_1 \quad \frac{B, x{:}\sigma \vdash M = N{:}\rho, \vdash_c \sigma = \tau}{B \vdash \lambda x{:}\sigma.M = \lambda x{:}\tau.N{:}\sigma \to \rho}$$

$$(\xi)_2 \quad \frac{B \vdash M = N{:}\sigma}{B \vdash \lambda t.M = \lambda t.N{:}\forall t.\sigma} \qquad t \text{ not free in } B.$$

We satisfy weak extensionality using "choice functions" to determine the meaning of $\lambda x{:}\sigma.M$ or $\lambda t.M$. Specifically, we wish to define the meaning of $B \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau$ from the function

$$g(d) = [\![B, x{:}\sigma \vdash M{:}\tau]\!]\,\eta[d/x].$$

In a nonextensional structure, there may be several elements representing g, so we need some extra machinery to choose which one. We need to make sure that if $B, x{:}\sigma \vdash M{:}\tau$ and $B, x{:}\sigma \vdash N{:}\tau$ give us the same function g, then we choose the same meaning for $B \vdash \lambda x{:}\sigma.M{:}\sigma \to \tau$ and $B \vdash \lambda x{:}\sigma.N{:}\sigma \to \tau$. The simplest way to do this is to use a choice function $\Psi_{a,b}$ to select the meaning $\Psi_{a,b}(g)$ of a lambda abstraction.

To be more precise, a nonextensional frame is defined in the same way as an extensional frame, except that instead of requiring each $\Phi_{a,b}$ and Φ_f to be bijections, we require additional functions $\Psi_{a,b}$ and Ψ_f such that

$$\Psi_{a,b} \circ \Phi_{a,b} = Id_{[\mathrm{Dom}^a \to \mathrm{Dom}^b]} \qquad \text{and} \qquad \Psi_f \circ \Phi_f = Id_{[\Pi_{a \in \mathrm{Kind}^T}.\,\mathrm{Dom}^{fa}]}.$$

(We make no assumptions about the reverse compositions $\Phi_{a,b} \circ \Psi_{a,b}$ and $\Phi_f \circ \Psi_f$.) The functions $\Psi_{a,b}$ and Ψ_f then replace $\Phi_{a,b}^{-1}$ and Φ_f^{-1} in defining the meanings of terms of the form $\lambda x{:}\sigma.M$ and $\lambda t.M$. The completeness

proof for the nonextensional case is a simple modification of the completeness proof given in Section 5 (see the completeness proof for nonextensional untyped lambda calculus in (Meyer, 1982)).

Nonextensional combinatory models are somewhat more complicated than extensional combinatory models. One important feature of the combinatory model definition is that it reduces the definition of model to a set of first-order axioms. It is therefore appropriate to remove the condition $\Psi \circ \Phi = Id$ from the definition of frame and incorporate it into the set of axioms. To do this, we use a family of "choice elements" $\{\varepsilon\}$ corresponding to the family of choice functions $\{\Psi\}$. The basic idea is illustrated in the following discussion of untyped combinatory models.

An *untyped combinatory model* is an untyped combinatory algebra $\mathscr{D} = \langle D, \Phi \rangle$ with element $\varepsilon^{\mathscr{D}} \in D$ satisfying

$(\varepsilon.1)$ $\forall d, e(\varepsilon d)e = de,$

$(\varepsilon.2)$ $\forall e(d_1 e = d_2 e) \supset \varepsilon d_1 = \varepsilon d_2,$

$(\varepsilon.3)$ $\varepsilon\varepsilon = \varepsilon.$

Untyped lambda abstraction can be interpreted using ε as

$$[\![\lambda x.M]\!]\eta = \varepsilon d,$$

where

$$de = [\![M]\!]\,\eta[e/x] \qquad \text{for all} \quad e \in D.$$

Since $\mathscr{D}$ is a combinatory algebra, an element $d \in D$ with $de = [\![M]\!]\,\eta[e/x]$ for all $e \in D$ will exist for any M. Furthermore, weak extensionality (ξ) follows from the properties of ε. Note that

$$\varepsilon = \lambda x \lambda y.xy.$$

A comprehensive discussion of the equivalence between the environment and combinatory model definitions for untyped lambda calculus is given in Meyer (1982). Note that since the combinatory algebra axioms are equational, nonextensional combinatory algebras from an algebraic variety (Grätzer, 1968). However, the axioms for ε, like the extensionality axioms discussed in the preceding subsection, are not equational.

In the case of second-order lambda calculus, we need a family of typed ε's. At the very least we need an

$$\varepsilon_{a,b} = \lambda x{:}a \to b \lambda y{:}a.xy$$

for every $a, b \in \mathrm{Kind}^T$ and, for every $f \in \mathrm{Kind}^{T \to T}$,

$$\varepsilon_f = \lambda x{:}\forall f.\lambda t.xt$$

to serve the roles of $\Psi_{a,b}$ and Ψ_f. However, these are not quite enough since we have no way of defining, say,

$$\lambda t.\lambda x{:}t \to t.\lambda y{:}t.xy$$

from combinators and the above ε's. Essentially, we need to be able to define $\varepsilon_{a,b}$ as a function of a and b.

There seem to be a number of ways of defining nonextensional second-order combinatory models and we have not made a thorough study of the possibilities. One set of choice elements that yields a combinatory characterization of nonextensional models includes

$$\varepsilon_0 = \lambda s.\lambda t.\lambda x{:}s \to t.\lambda y{:}s.xy,$$

from which we can define any $\varepsilon_{a,b}$ by application, and for each $f{:}T^i \Rightarrow T$, an

$$\varepsilon_f = \lambda \vec{s}.\lambda x{:}\forall t(f\vec{s}t).\lambda t.x\vec{s}t.$$

The types and axioms for these ε's are easily derived from the lambda terms above. The details of this combinatory model definition are cumbersome, but essentially straightforward. (The family of choice elements ε_0 and ε_f for all $f{:}T^i \Rightarrow T$ repair an oversight in Mitchell, 1984b.)

7. Examples of Models

7.1. *Introduction*

We will discuss models of second-order lambda calculus that are constructed from untyped structures. These are the simplest examples of models, and historically the first. The models fall into two groups. In the first class of models, types are represented by elements of a "universal" domain. This allows us to use ordinary untyped lambda calculus to define operations on types. In the second class of models, types are quotients of subsets of an untyped value space. Another class of models, Girard's qualitative domains, are too recent for us to survey here (Girard, 1986). In addition to describing two kinds of second-order models, we will also see that the ideal model of type inference (MacQueen and Sethi, 1982; MacQueen, Plotkin, and Sethi, 1986) is not a model, and that there are no nontrivial finite models.

There are several variations on both universal domain and HEO models, but we will not take the time to discuss all of them. Universal domain models may be constructed using closures, finitary retracts, or finitary projections of certain domains. We will discuss the closure model and refer to

the literature on finitary retracts and projections. Girards' HEO_2 is a specific model based on recursive function application. The main idea may be used to construct a second-order model over any "partial combinatory algebra," a class of structures which includes all models of untyped lambda calculus (Plotkin, 1985; Rosolini, 1986). Some variations are discussed in (Mitchell, 1986b) and some connections between retracts and equivalence relations are discussed in Section 7 of (Scott, 1976).

Although it is not very exciting, it is probably worth mentioning a trivial model construction. Any model $\mathscr{D}$ of the untyped λ-calculus may be viewed as a second-order model by taking $\mathscr{D}$ as the sole element of Kind^T.

7.2. *Retract Models*

There are three kinds of models built using retracts of universal domains. A *retraction* is a function f with the property that $f \circ f = f$, and the range of a retraction is called a *retract*. In the retract models, the types are chosen to be some class of retractions of a model of untyped lambda calculus. One of the reasons why retract models are easy to work with is that type operators like $\to$ and $\forall$ may be represented by lambda-definable functions on retractions (see Scott, 1976 for further discussion). An important property of the three models below is that in each case, a very rich class of retractions is itself a retract of the untyped value space. Because of this, each model will have a "type of all types," something which is not generally required of second-order lambda models. The three models will differ primarily in the class of retractions used as the type of types.

The first model construction was based on Scott's $\mathscr{P}\omega$ model of untyped lambda calculus, with types represented by a special class of retracts called closures. This model is due to McCracken (1979), drawing on ideas presented in (Scott, 1976). We assume that the reader is familiar with the $\mathscr{P}\omega$ model of the untyped lambda calculus (Scott, 1976), with

$$\Phi = \mathbf{fun} : \mathscr{P}\omega \to [\mathscr{P}\omega \to \mathscr{P}\omega]$$

mapping each element of $\mathscr{P}\omega$ to a continuous function on $\mathscr{P}\omega$, and

$$\Psi = \mathbf{graph} : [\mathscr{P}\omega \to \mathscr{P}\omega] \to \mathscr{P}\omega$$

mapping every continuous function to some element of $\mathscr{P}\omega$. (An important relationship between Φ and Ψ is that $\Phi \circ \Psi$ is the identity function on $[\mathscr{P}\omega \to \mathscr{P}\omega]$.) A certain amount of notational clutter will be eliminated by writing closed lambda terms to describe elements of $\mathscr{P}\omega$, as well as writing

$$de \quad \text{for } (\Phi d)(e) \quad \text{when } d, e \in \mathscr{P}\omega,$$

$$d \circ e \text{ for } \lambda x.\, d(ex) \text{ when } d, e \in \mathscr{P}\omega.$$

A *retraction in* $\mathscr{P}\omega$ is an element $\Psi(f) \in \mathscr{P}\omega$ such that the function $f: \mathscr{P}\omega \to \mathscr{P}\omega$ is a retraction. This is equivalent to saying that a retraction in $\mathscr{P}\omega$ is an element $a \in \mathscr{P}\omega$ with $a \circ a = a$ in the notation above.

We can build a second-order model from $\mathscr{P}\omega$ by using the closures as types. We say a retraction $a \in \mathscr{P}\omega$ is a *closure* if

$$ad \geqslant d \qquad \text{for all} \quad d \in \mathscr{P}\omega$$

and let

$$\text{Kind}^T = \{a \in \mathscr{P}\omega \mid a \text{ is a closure}\}$$

be the collection of closures. The elements of type a are the elements fixed by a, i.e., for every $a \in \text{Kind}^T$, we let

$$\text{Dom}^a = \{ad \mid d \in \mathscr{P}\omega\}.$$

We may think of the closure a as coercing untyped elements of $\mathscr{P}\omega$ into elements of type a. Since a is a retraction, this coercion leaves elements of type a unchanged. As shown in (Scott, 1976), there is a closure $V \in \text{Kind}^T$ of all closures, so that $\text{Dom}^V = \text{Kind}^T$. This is a particular property of closures of $\mathscr{P}\omega$ which fails for retractions. Specifically, the collection of all retractions in $\mathscr{P}\omega$ is not a retract of $\mathscr{P}\omega$ (Scott, 1976). If a is a closure, then it will be convenient to write

$$d{:}a \qquad \text{for} \quad d = ad,$$

which is equivalent to saying $d \in \text{Dom}^a$. In addition, a useful abbreviation is

$$\lambda x{:}a.M \qquad \text{for} \quad \lambda y.\{ay/x\}M.$$

Intuitively, $\lambda x{:}a.M$ is the function $\lambda x.M$, restricted to the range of closure a.

If a and b are closures, then we want $a \to b$ to be a closure which coerces every element d of $\mathscr{P}\omega$ to a mapping from Dom^a to Dom^b. In addition, we would like to have each function $\Phi(d)$ mapping Dom^a into Dom^b represented exactly once in the range of $a \to b$, so that $\text{Dom}^{a \to b}$ is an extensional collection of functions (see Scott, 1976 for further discussion). Both of these goals may be accomplished by taking

$$a \to b = \lambda x.b \circ x \circ a.$$

Intuitively, $a \to b$ works by taking any element d and producing the element $b \circ d \circ a$ which, when used as a function, first coerces its argument to an element of type a, then applies d, and then coerces the result to type b.

It is easy to see that if $x:a$, then $((b \circ d \circ a)x):b$, so $(b \circ d \circ a)$ represents a function from Dom^a to Dom^b. In addition, if d already represents a function from Dom^a to Dom^b, so $dx:b$ whenever $x:a$, then $(b \circ d \circ a)x = dx$ for all $x:a$. What is a little less obvious is that if $(b \circ d_1 \circ a)x = (b \circ d_2 \circ a)x$ for all $x:a$, then $b \circ d_1 \circ a = b \circ d_2 \circ a$[5]. This means that range of $a \to b$ contains exactly one representative for each continuous function on $\mathscr{P}\omega$ that maps a into b. Based on this discussion, we define

$$\to = \lambda a{:}V.\lambda b{:}V.(\lambda x.b \circ x \circ a)$$

and write $\to$ as an infix operator, as in $a \to b$. It is easy to verify that for any $a, b \in \mathscr{P}\omega$, the element $a \to b$ is a closure (see Scott, 1976).

Since $\mathrm{Kind}^T = \mathrm{Dom}^V$, we will use the same function space constructor $\to$ for both types and kinds. For each kind expression κ, we let $\mathrm{TP}(\kappa)$ be the expression obtained by replacing all occurrences of T in κ by V and all occurrences of $\Rightarrow$ by $\to$. Using the definition of $\to$ above, we may interpret $\mathrm{TP}(\kappa)$ as a closure, and so we take $\mathrm{Kind}^\kappa = \mathrm{Dom}^{\mathrm{TP}(\kappa)}$. It is now easy to show that $\to$ is in $\mathrm{Kind}^{T \Rightarrow (T \Rightarrow T)}$. We leave this to the reader.

The intuition behind $\forall$ is quite straightforward. If $f:V \to V$ is a function from closures to closures, then every element $x:\forall f$ should map each closure (type) $t:V$, to some element xt of type ft. Therefore, $\forall f$ should be a function that coerces any $x \in \mathscr{P}\omega$ to a function which, given any $t:V$, returns an element $xt:ft$. Writing this out as a lambda term (including the type assumptions), we are led to the definition

$$\forall = \lambda f{:}V \to V.\lambda x.\lambda t{:}V.(ft)(xt).$$

Recall that $xt:ft$ is an abbreviation for $xt = (ft)(xt)$, so we have used $(ft)(xt)$ in the definition of $\forall$. It is easy to verify that if $f \in \mathrm{Kind}^{T \Rightarrow T}$, then $\forall f \in \mathrm{Kind}^T$, and that $\forall \in \mathrm{Kind}^{(T \Rightarrow T) \Rightarrow T}$.

To complete the definition of a second-order frame, it remains to define a family of $\Phi_{\kappa_1, \kappa_2}$ functions that give us a kind structure, and $\Phi_{a,b}$ and Φ_f for every $a, b \in \mathrm{Kind}^T$ and $f \in \mathrm{Kind}^{T \Rightarrow T}$. Surprisingly, all of these may be obtained as restrictions of the untyped Φ mapping $\mathscr{P}\omega$ to $[\mathscr{P}\omega \to \mathscr{P}\omega]$. With $\Phi_{\kappa_1, \kappa_2}$ defined to be the restriction of Φ to $\mathrm{Kind}^{\kappa_1 \Rightarrow \kappa_2}$ and $\mathscr{I}$ defined as above on $\to$ and $\forall$, it is not hard to verify that $Kind_{\mathscr{P}\omega} = \langle \{\mathrm{Kind}^\kappa, \{\Phi_{\kappa_1, \kappa_2}\}, \mathscr{I}\rangle$ is a kind structure. Assuming $\mathscr{C}_{\mathrm{term}}$ is empty, we leave $\mathscr{I}_{\mathrm{Dom}}$ empty and take

$$Dom_{\mathscr{P}\omega} = \langle \{\mathrm{Dom}^a \mid a \in \mathrm{Kind}^T\}, \mathscr{I}_{\mathrm{Dom}}\rangle.$$

[5] This is proved by noticing that the restriction to $x:a$ is inessential, and so $(b \circ d_1 \circ a)x = (b \circ d_2 \circ a)x$ holds in $\mathscr{P}\omega$. Therefore, by rule (ξ) of untyped lambda calculus, $\lambda x.(b \circ d_1 \circ a)x = \lambda x.(b \circ d_2 \circ a)x$. Working out the definition of $\circ$ gives the desired equation.

For $a, b \in \mathrm{Kind}^T$, $d \in \mathrm{Dom}^{a \to b}$, $f \in \mathrm{Kind}^{T \Rightarrow T}$, and $e \in \mathrm{Dom}^{\forall f}$, we let

$$\Phi_{a,b}(d) = \Phi(d)|_{\mathrm{Dom}^a} \qquad \text{and} \qquad \Phi_f(e) = \Phi(e)|_{\mathrm{Kind}^T}.$$

These restrictions of the "untyped" Φ have some remarkable properties:

LEMMA 19. (i) *$\Phi_{a,b}$ is a one-to-one and onto function from* $\mathrm{Dom}^{a \to b}$ *to* $[\mathrm{Dom}^a \to \mathrm{Dom}^b]$, *where* $[\mathrm{Dom}^a \to \mathrm{Dom}^b]$ *is the set of continuous functions from* Dom^a *to* Dom^b.

(ii) *Φ_f is a one-to-one and onto function from* $\mathrm{Dom}^{\forall f}$ *to* $[\Pi_{a \in \mathrm{Kind}^T} \mathrm{Dom}^{f(a)}]$, *where* $[\Pi_{a \in \mathrm{Kind}^T} \mathrm{Dom}^{f(a)}]$ *is the set of continuous functions from* $\Pi_{a \in \mathrm{Kind}^T} \mathrm{Dom}^{f(a)}$.

Proof. (i) (Sketch) Note that $d \in \mathrm{Dom}^{a \to b}$ implies $d = b \circ d \circ a$. Thus $\Phi(d)(e) = b(d(ae)) \in \mathrm{Dom}^b$. It can also be shown that the range of $\Phi_{a,b}$ is all of $[\mathrm{Dom}^a \to \mathrm{Dom}^b]$. Suppose there are $d_1, d_2 \in \mathrm{Dom}^{a \to b}$ such that $\Phi_{a,b}(d_1) = \Phi_{a,b}(d_2)$. It is then easy to show that $\Phi(d_1) = \Phi(d_2)$. Hence, $\Psi(\Phi(d_1)) = \Psi(\Phi(d_2))$. However $d_1, d_2 \in \mathrm{Dom}^{a \to b}$ implies $d_i = b \circ d_i \circ a$ for $i = 1, 2$, and therefore $\Psi(\Phi(d_i)) = \Psi(\Phi(b \circ d_i \circ a)) = b \circ d_i \circ a = d_i$, where the middle equation holds since $b \circ d_i \circ a$ is of the form $\Psi(g)$ and $\Psi(\Phi(\Psi(g))) = \Psi(g)$. Thus $d_1 = \Psi(\Phi(d_1)) = \Psi(\Phi(d_2)) = d_2$, and it follows that $\Phi_{a,b}$ is one to one. (ii) Similar. ∎

This shows that $\mathscr{F}_{\mathscr{P}\omega} = \langle \mathit{Kind}_{\mathscr{P}\omega}, \mathit{Dom}_{\mathscr{P}\omega}, \{\Phi_{a,b}, \Phi_f\} \rangle$ is a second-order frame. Since $[\mathrm{Dom}^a \to \mathrm{Dom}^b]$ and $[\Pi_{a \in \mathrm{Kind}^T} \mathrm{Dom}^{f(a)}]$ consist of all continuous functions of the appropriate functionality, it is easy to verify that this is an extensional second-order model.

THEOREM 20 [McCracken, 1979]. *$\mathscr{F}_{\mathscr{P}\omega} = \langle \mathrm{Kind}_{\mathscr{P}\omega}, \mathrm{Dom}_{\mathscr{P}\omega}, \{\Phi_{a,b}, \Phi_f\} \rangle$, as defined above, is an extensional second-order model.*

The model $\mathscr{F}_{\mathscr{P}\omega}$ has several interesting features. Perhaps most interesting is that $\mathrm{Kind}^T \in \mathit{Dom}$, giving the set of types a very rich structure. In particular we can solve recursive domain equations in this model. Somewhat surprisingly, given the definition of Kind^T, the correspondence between Kind^T and *Dom* is bijective, i.e., $a = b$ iff $\mathrm{Dom}^a = \mathrm{Dom}^b$.

A similar argument shows that the class of finitrary retract models, developed in McCracken (1984b) using ideas of Scott (1980b), is also an extensional second-order model. We say a cpo is a *domain* if it is ω-algebraic and consistently complete, and a retraction r is *finitary* if the range of r is a domain. The finitary retract model is built from a domain model of untyped lambda calculus by taking finitary retracts as types. While similar to the closure model described above (e.g., in the definitions of $\to$ and $\forall$), there are some differences. For example, the relation between

Kind^T and *Dom* is not bijective. A similar extensional model using finitary projections appears in Amadio *et al.* (1986); the ideas behind this model also appear implicitly in several papers of Scott. The finitary projection model is again bijective in the relationship between Kind^T and *Dom*. In Amadio *et al* (1986) it is shown how to solve higher order recursive domain equations in this model. The same paper also shows that the type structures of all three models are very similar.

7.3. *The Ideal Model of Polymorphic Type Inference*

The ideal model proposed in (MacQueen and Sethi, 1982; MacQueen, Plotkin, and Sethi, 1986) was designed to explain polymorphic type inference for untyped lambda calculus. In the programming language ML, for example, the untyped identity function $\lambda x.x$ is given all types of the form $\alpha \to \alpha$, where α may be any "monotype" without $\forall$ (Milner, 1978). The assignment of types to untyped terms is formalized as a deductive system for assertions like $\lambda x.x : \alpha \to \alpha$. A natural extension of ML typing is to assign second-order types to untyped lambda terms, giving the untyped identity type $\forall t.t \to t$, from which all of the ML typings can be obtained. A semantic explanation of the deductive system for polymorphic type assignment involves a structure for interpreting untyped lambda expressions and a way of associating a predicate with each second-order type (Mitchell, 1984b). The ideal model is an example of such a structure, using ideals over complete partial orders as types (Milner, 1978; Shamir and Wadge, 1977). It is sometimes thought that the ideal model is in fact a model of second-order lambda calculus. However, we will see that it is not. The shortcomings of the ideal model (as a second-order model) will be used to motivate the HEO model in the next subsection. We should emphasize that this model was only intended to explain type membership, not equality between typed terms. So it is through no fault of the authors of (MacQueen and Sethi, 1982; MacQueen, Plotkin, and Sethi, 1986) that the ideal model is not a second-order lambda model.

Although there may be trivial or contrived ways of treating the ideal model as a model of second-order lambda calculus, the most natural way would be to interpret each typed term as the term obtained by erasing all type information. This makes some sense, since the meaning of any typed term ends up belonging to the correct semantic type. The problem with this interpretation of terms is that it is not even weakly extensional, let alone extensional. Every model of second-order lambda calculus (or any other typed lambda calculus) must satisfy the axiom

$$\text{If } M = N \text{ for all } x:t, \text{ then } \lambda x:t.M = \lambda x:t.N.$$

We will construct a counterexample to this axiom in the ideal model. For

concreteness, the counterexample will use the type $\forall t.t$. However, the argument is quite general.

Let P and Q be any two closed terms of the same type τ which have different meanings in the ideal model. (To be more concrete, we could take $P = \lambda t.\lambda x{:}t.\lambda y{:}t.x$ and $Q = \lambda t.\lambda x{:}t.\lambda y{:}t.y$ of type $\tau = \forall t.t \to t \to t$.) Since $\forall t.t$ has only one element $\perp$, we have

$$x\tau P = x\tau Q = \perp$$

for all $x{:}\forall t.t$. Therefore, we would like to obtain equal lambda terms by lambda abstracting $x{:}\forall t.t$ on both sides of the equation. However, we have

$$\lambda x{:}(\forall t.t).x\tau P \neq \lambda x{:}(\forall t.t).x\tau Q$$

in the ideal model, since $\lambda x.xP$ and $\lambda x.xQ$ are distinct. Thus weak extensionality fails. For similar reasons, weak extensionality generally fails in interpretations of second-order typed lambda calculus based on the models of type inference discussed in (Mitchell, 1984b).

7.4. *HEO_2 and Related Models*

The main reason that ideal model and related structures do not form models of second-order lambda calculus is that equality is untyped, or independent of type. However, we can construct second-order models in much the same spirit if, in addition to a membership predicate, we also associate an equivalence relation with each type. Essentially, the predicates say what the elements of each type are, and the equivalence relations say when two elements are to be regarded as equal with respect to that type. One intuitive explanation for this view of types is based on computer implementation. In compiling a typed language, we might choose to represent characters or boolean values as single bytes. Any byte would be accepted as a valid representation of either a boolean or a character. However, we are likely to regard any byte with least significant bit 1 as a representation of *true*, and so any two bytes ending with 1 will be regarded as equal booleans. However, all bytes with the same least significant bit will not be considered equal characters. Thus, although characters and booleans may have the same membership predicate on machine-level representations, the equality relations are different.

When we formalize the two-part interpretation of types, it is technically convenient to combine membership predicates and equivalence relations into a single notion. Intuitively, a partial equivalence relation on a set S is intended to be an equivalence relation R on a subset $S_1 \subseteq S$. However, it turns out that the predicate S_1 is superfluous. To begin with $S_1 = \{s \in S \mid \langle s, s\rangle \in R\}$, so it is easy to see that S_1 is determined by R. In addi-

tion, R is an equivalence relation on some subset of S iff R is a symmetric and transitive relation on all of S. Therefore, we will simplify the technical details by working with symmetric, transitive relations instead of predicates and equivalence relations. If S is any set, we say a binary relation R on S is a *partial equivalence relation* if R is symmetric and transitive.

The use of partial equivalence relations has a significant history. Partial equivalence relations for first-order function types were introduced in (Myhill and Shepherdson, 1955) and extended to higher order functional types $\sigma \to \tau$ in (Kreisel, 1959). Kreisel's structure is also described in (Troelstra, 1973), where it is called HEO, for the *hereditarily effective operations*. The structure HEO was extended to an interpretation of predicative polymorphic types in (Beeson, 1982) and to a second-order model HEO_2 in (Girard, 1972; Troelstra, 1973). The structure HEO_2 was also discovered independently by Moggi and Plotkin (personal communication, 1985). A partial equivalence relation interpretation of functional types (in a somewhat more general setting) is also discussed in (Scott, 1976) and taken up in the study of polymorphic type inference in, e.g., (Hindley, 1983a; Coppo and Zacchi, 1986). Further discussion and some general results about partial equivalence relation models of second-order lambda calculus are given in Mitchell (1986b).

We will now concentrate on Girard's model HEO_2, which is a particular model of second-order lambda calculus based on partial equivalence relations. Instead of using partial equivalence relations over untyped lambda models, HEO_2 is based on the integers with partial recursive function application. We assume some enumeration of all partial recursive functions, and write $\{n\}m$ for the application of the nth recursive function to m. As in (Girard, 1972), we will assume that the recursive functions are numbered so that $\{0\}m = 0$ for every integer m. With this assumption on the coding of recursive functions, we will end up with at least one element of every type (namely, the equivalence class of 0).

The first step in describing HEO_2 is to define the kind frame $Kind_{\mathrm{HEO}}$. We let Kind^T be the set of all partial equivalence relations R over the integers, subject to the constraint that $\langle 0, 0\rangle \in R$. The remaining Kind^κ are defined inductively, with $\mathrm{Kind}^{\kappa_1 \Rightarrow \kappa_2}$ the set of all functions from Kind^{κ_1} to Kind^{κ_2}. Since $\mathrm{Kind}^{\kappa_1 \Rightarrow \kappa_2}$ is a set of functions, we let $\Phi_{\kappa_1, \kappa_2}$ be the identity. For any $R, S \in \mathrm{Kind}^T$, we let $R \to S$ be the relation

$$R \to S = \{\langle n_1, n_2\rangle \mid \text{if } \langle m_1, m_2\rangle \in R \text{ then } \langle \{n_1\}m_1, \{n_2\}m_2\rangle \in S\}.$$

It is easy to see that $\langle 0, 0\rangle \in R \to S$, and so $\to \in \mathrm{Kind}^{T \Rightarrow T \Rightarrow T}$. If $f \in \mathrm{Kind}^{T \Rightarrow T}$ is any function from types (partial equivalence relations) to types, then we define $\forall f$ by

$$\forall f = \bigcap_{R \in \mathrm{Kind}^T} f(R).$$

It is also quite easy to see that $\langle 0, 0\rangle \in \forall f$, and so $\forall$ is a function of the appropriate kind. We let $Kind = \langle \{\text{Kind}^\kappa, \{\Phi_{\kappa_1,\kappa_2}\}, \mathscr{I}\rangle$ with $\mathscr{I}(\to)$ and $\mathscr{I}(\forall)$ as above. Since *Kind* is a full function hierarchy, it is clear that every constructor expression has a meaning in *Kind*.

For every $R \in \text{Kind}^T$ and every integer n with $\langle n, n\rangle \in R$, we let $[n]_R$ be the equivalence class

$$[n]_R = \{m \mid \langle n, m\rangle \in R\},$$

and define Dom^R to be the set of all such equivalence classes

$$\text{Dom}^R = \{[n]_R \mid \langle n, n\rangle \in R\}.$$

We then take Dom_{HEO} to be the collection of all Dom^R. Note that since $\langle 0, 0\rangle \in R$ for every R, every Dom^R is nonempty. The functions $\Phi_{R,S}$ and Φ_f are defined by

$$\Phi_{R,S}[n]_{R\to S}[m]_R = [\{n\}m]_S$$

$$\Phi_f[n]_{\forall f} R = [n]_{f(R)}$$

and the sets $[\text{Dom}^R \to \text{Dom}^s]$ and $[\Pi_{R\in \text{Kind}^T}.\text{Dom}^{f(R)}]$ are defined to be the ranges of functions $\Phi_{R,S}$ and Φ_f, respectively. It is easy to see that both functions are well defined on equivalence classes. Although Φ_f may look trivial, it is not entirely so since $[n]_{f(R)}$ will generally be a larger equivalence class than $[n]_{\forall f}$. At this point, we have a frame

$$\text{HEO}_2 = \langle Kind_{\text{HEO}}, Dom_{\text{HEO}}, \{\Phi_{a,b}, \Phi_f\}\rangle$$

and it remains to show that every term has a meaning.

The proof that HEO_2 is a model relies on elementary facts from recursive function theory, such as recursive sequencing functions and the s_n^m theorem (see, e.g., Rogers, 1967). The main idea is to show inductively that for every term $B \vdash M:\sigma$, there is a recursive function for the meaning $[\![B \vdash M:\sigma]\!]$. This shows not only that every term has a meaning, but that we can compute the meaning of every term as a function of the environment. Since HEO_2 contains only recursive functions, we use the stronger induction hypothesis to show that lambda abstractions have meaning in the model.

Recall that the meaning of a term depends only on the finite sequence of values given to its free variables, not the entire environment. Consequently, we may regard $[\![B \vdash M:\sigma]\!]$ as a function on finite sequences of values, the types of these values given by B. Since every value in HEO_2 is an equivalence class of integers, we will consider $[\![B \vdash M:\sigma]\!]$ recursive if we have a corresponding recursive function on integer representatives (or "codes") for values. More precisely, we say f is *a recursive function for*

$[\![\{x_1:\sigma_1, ..., x_k:\sigma_k\} \vdash M:\sigma]\!]$ if, for any environment $\eta \models \{x_1:\sigma_1, ..., x_k:\sigma_k\}$ and any sequence $\langle n_1, ..., n_k\rangle$ of integers with $n_i \in \eta(x_i)$, the function f computes an integer

$$f(\langle n_1, ..., n_k\rangle) \in [\![\{x_1:\sigma_1, ..., x_k:\sigma_k\} \vdash M:\sigma]\!]\eta.$$

A straightforward induction shows that there is a recursive function for the meaning of every term and that the recursive function for $[\![B \vdash M:\sigma]\!]$ has the appropriate type. The only nontrivial case is lambda abstraction, which uses the s_n^m theorem. It follows that HEO_2 is a second-order lambda model.

7.5. *Finite Models*

One distinction between untyped lambda calculus and the ordinary typed lambda calculus is that typed lambda calculus has nontrivial models in which all types are finite, but the untyped lambda calculus has no finite models (see Barendregt, 1984, Proposition 5.1.15). A simple argument due to Gordon Plotkin (private communication, 1985) shows that there are no nontrivial models of second-order lambda calculus in which all types are finite. The *numerals* are the terms of the form

$$\lambda t.\lambda f:t \to t.\lambda x:t.f^n x,$$

where $f^n x$ is the term $f(f \cdots (fx) \cdots)$ with n occurences of f (see Statman, 1981; Fortune *et al.*, 1983). All of the numerals have type $\forall t.(t \to t) \to t \to t$. We write $\bar{n}$ for the numeral $\lambda t.\lambda f:t \to t.\lambda x:t.f^n x$.

Since every integer function which can be proved total recursive in analysis is definable in second-order lambda calculus (Girard, 1972; Statman, 1981), there is a term EQ with

$$\vdash \mathrm{EQ}\,\bar{m}\bar{n} = \mathrm{true} \qquad \text{if} \quad m = n$$

$$\vdash \mathrm{EQ}\,\bar{m}\bar{n} = \mathrm{false} \qquad \text{if} \quad m \neq n,$$

where true and false are the terms

$$\mathrm{true} ::= \lambda t\lambda x:t\lambda y:t.x$$

$$\mathrm{false} ::= \lambda t\lambda x:t\lambda y:t.y$$

with type $\forall t.t \to t \to t$. (The term EQ can also be constructed explicitly using the arithmetic functions given in (Fortune *et al.*, 1983, p. 167.) If $\mathscr{F}$ is a finite model, then clearly $\mathscr{F} \models \bar{m} = \bar{n}$ for some $m \neq n$, since there are only finitely many elements of type $\forall t.(t \to t) \to t \to t$. Therefore, in any finite model $\mathscr{F}$, we have

$$\mathscr{F} \models \mathrm{false} = \mathrm{EQ}\,\bar{m}\bar{n} = \mathrm{EQ}\,\bar{n}\bar{n} = \mathrm{true}.$$

However, it is easy to see that if true = false, then for any type t and $x, y:t$, we have

$$x = \text{true}\ txy = \text{false}\ txy = y.$$

It follows that every equation holds in $\mathscr{F}$, and $\mathscr{F}$ is a trivial model. This concludes the proof that the only second-order model with no infinite types is the trivial model.

8. Summary and Directions for Future Work

The second-order lambda calculus is a very expressive, explicitly typed extension of the ordinary typed lambda calculus. In this paper, we have examined the semantics of the language. Intuitively, a term $\lambda x{:}\sigma.M$ denotes a function from type σ to some type τ, and a term $\lambda t.M$ denotes a function from types to the union of all types. However, since terms like $\lambda t.\lambda x{:}t.x$ can be applied to their own types, the naive interpretation of second-order lambda terms contradicts standard set theory. Borrowing an idea from the semantics of untyped lambda calculus (Barendregt, 1984; Meyer, 1982; Scott, 1976), we use a set together with an "element-to-function" map Φ in place of a set of functions. We interpret a λ-abstraction with domain a and range b as an element d which we may regard as a function by applying the map $\Phi_{a,b}$ to d. Since the range of $\Phi_{a,b}$ need not be all set-theoretic functions from type a to b, we can associate a set with each type and avoid set-theoretic paradoxes.

A collection of sets, one for each type and kind, together with an appropriate collection of bijective "element-to-function" maps, is called an extensional second-order frame. Second-order frames are analogous to untyped functional domains (Meyer, 1982) for untyped lambda calculus and type frames for ordinary typed lambda calculus (Henkin, 1950). Like their analogs, second-order frames have the right structure for interpreting terms, but may not contain enough elements to give meanings to all terms.

Environment models are defined as frames in which every well-typed term has a meaning. Although it depends on the inductive definition of meanings of terms, this model definition is straightforward and useful. The soundness and completeness theorems suggest that the definition is reasonable and not too restrictive. The soundness theorem shows that every structure which meets our definition has the right equational properties, while completeness demonstrates that every theory has a model. More evidence that our model definition is useful for studying second-order lambda calculus is provided by observing that several models proposed in the literature also meet our definition. However, it is worth mentioning

that our soundness and completeness theorems apply only to models without empty types.

After preliminary work on this paper was completed, we became aware of a class of models based on partial equivalence relations, as described in Section 7.4. In some natural variations on these models, there exist empty types. While it is easy to remove the assumption that all types are non-empty from our model definition, the appropriate changes to the proof system are not entirely straightforward. To preserve soundness, the rule (remove hyp) must be discarded. In Meyer *et al.* (1987), additional proof rules for reasoning about empty types are given and a completeness theorem is proved.

In showing that certain structures are models in Section 7, we make use of independent characterizations of functions, like continuity or recursiveness. In the absence of such additional structure in the model, it might be more difficult to certify that a second-order frame is actually an environment model. Therefore, we provide an algebraic equivalent of the environment model condition that "everything must work out right." Combinatory models are defined as second-order frames which contain combinators S, K, A, B, C, D, where each combinator is characterized by an equation. We prove that the environment and combinatory model definitions are equivalent and, in the process, show how to translate between lambda terms and equivalent second-order combinatory terms. The combinatory model definition also shows that the model theory of second-order lambda calculus is reducible to the standard model theory of first-order logic.

Product types, sums, and existential types can be added to second-order lambda calculus by adding additional constructor constants and either term constants or additional term formation rules. Although we have not presented the details here, our model definition extends relatively easily. For example, a model of second-order lambda calculus with existential types is a model of the second-order lambda calculus with constructor constant $\exists \in \mathscr{C}_{\text{cst}}$ of kind $(T \Rightarrow T) \Rightarrow T$ and term formation rules given in (Mitchell and Plotkin, 1988). The extra constructor constant produces additional elements of Kind^T, and the additional structure associated with sets $\text{Dom}^{\exists f}$ is easily determined from the operations **rep** and **sum** discussed in (Mitchell and Plotkin, 1988).

The Automath languages (De Bruijn, 1980) are essentially extensions of the typed lambda calculus formed by allowing the types of terms to be functions of elements of other types. Automath expressions of "first-order dependent type" define elements of $\Pi_{d \in A} \text{Dom}^{f(d)}$ for $A \in Dom$ and $f: A \to \text{Kind}^T$. In (Barendregt and Rezus, 1983), a model for Classical Automath is constructed using closures, as in the models we discussed in Section 7. A general model definition for Classical Automath, along the lines we have proposed for second-order lambda calculus, seems relatively easy to work

out. We are grateful to Robert Harper for extending the semantics of second-order lambda calculus to include first-order dependent types (private communication, 1986). Other extensions of our language, presented in McCracken (1979), and the languages $F_3, F_4, \ldots$ of Girard mentioned earlier, allow terms of the form $\lambda v^{\kappa}.M$, where v^{κ} is a variable of higher kind. We believe that a straightforward extension of the model definition given here will suffice for this language, but have not worked out the details. The Calculus of Constructions developed by Coquand and Huet (1988) encompasses some of these extensions, but we have not worked out a precise model definition.

We have not considered second-order theories involving equations between constructors. The language is defined to suggest this possibility and the term model construction in the completeness proof does not seem to rely on the absence of constructor axioms. However, there are some complications that must be resolved. To begin with, Lemma 1 fails: if we take $\sigma_1 \to \tau_1 = \sigma_2 \to \tau_2$ as a nonlogical axiom, it does not follow that $\sigma_1 = \sigma_2$ or $\tau_1 = \tau_2$. Since Lemma 1 figures crucially in the proof of Lemma 8, it is no longer possible to show that the meaning of every well-typed term $B \vdash M : \sigma$ is independent of the way in which the typing is derived. Since Lemma 8 is important in our understanding of what an equation means (as discussed in Section 4.3), it would seem best to add more type information to terms so that Lemma 8 can be restored. Essentially, the syntax of terms would have to determine the types of all subterms up to constructor equivalence. For example, applications (MN) would have to be typed in a way that makes the types of M and N unambiguous. Some related discussion of ordinary typed lambda calculus with type equations is given in (Breazu-Tannen and Meyer, 1985).

Another extension of the second-order lambda calculus involving "bounded quantification" is proposed in (Cardelli and Wegner, 1985). A polymorphic function of the form $\lambda t \leqslant \tau.M$ will take any subtype of τ as a type parameter. In this calculus, we may write functions such as the identity $\lambda t \leqslant \mathrm{int}.\lambda x{:}t.t$ on all subtypes of int. (A related form of polymorphism was developed in Mitchell, 1984a.) This extension is designed to model the uses of subtypes and inheritance in object-oriented languages. A semantics of this language based on the PER model of Section 7.4 is developed in Bruce and Longo (1988), while a semantics based on interval models is presented in Martini (1988). While bounded quantification seems more expressive in an intuitive sense, there is an interpretation of bounded quantification into pure second-order lambda calculus (Breazu-Tannen *et al.*, 1989). This interpretation allows us to use models of second-order lambda calculus as models of bounded quantification.

There are a number of interesting open problems involving the semantics of the second-order lambda calculus. Recent work on a category theoretic

approach to second-order lambda calculus (Moggi, 1984; Seely, 1986; Pitts, 1987; Meseguer, 1989) seems quite promising in presenting another way of looking at this language. Based on recent work on categorical models of ordinary typed lambda calculus (Mitchell and Moggi, 1987), we expect the categorical models of second-order lambda calculus to be intuitionistic versions of our model definition. However, the details of this correspondence have yet to be worked out. Another direction for investigation is the relationship between our model definition and models based on Girard's qualitative domains (Girard, 1986), which were developed after the bulk of this paper was written. Since maps from types to types in Girard's model (and the related model of Coquand *et al.*, 1989) are functors rather than functions, extensionality may fail for kind $T \Rightarrow T$. We believe that if our extensionality requirement is dropped, then these models satisfy our definition. However, we have not checked the details. Although there are a growing number of examples of second-order models, we still do not know very much about them. It would be interesting to discover more models and study both the local structure (equational theories) and global structure of models.

One way to study the global structure of models is by examining the isomorphisms or retractions between types. In Reynolds (1984), it is shown that in every "set-theoretic" model there is some type S which is isomorphic to $(S \to B) \to B$ for some nontrivial B. This conflicts with classical set-theoretic function spaces quite clearly, implying that set-theoretic models (i.e., models in which the function space construction gives the full classical set-theoretic funclltion space) do not exist. In contrast, Bruce and Longo (1985) have characterized the class of isomorphisms that must hold in every second-order model. Essentially, these isomorphisms all follow from the "commutativity" of Cartesian product, i.e., $\sigma \times \tau$ is isomorphic to $\tau \times \sigma$. This property of $\times$ applies to the language without $\times$ since $\sigma \times \tau \to \rho$ is isomorphic to both $\sigma \to \tau \to \rho$ and $\tau \to \sigma \to \rho$, and so we expect $\sigma \to \tau \to \rho$ and $\tau \to \sigma \to \rho$ to be isomorphic. Similarly, regarding $\forall t(\sigma \to \tau)$ as a type of functions from types to σ to τ, we expect $\forall t(\sigma \to \tau)$ to be isomorphic to $\sigma \to \forall t.\tau$ when t does not appear free in σ. Since these are all the isomorphisms that hold in all models, Reynolds' problematic isomorphism does not hold, in general. This leaves open the possibility of relatively "natural" models which do not satisfy isomorphisms like S isomorphic to $(S \to B) \to B$. We might gain further insight by studying retractions instead of isomorphisms.

In our models, higher order operations on types are elements of higher kinds. For example, pair, sum, list stack, tree, etc. are all type constructors of kind $T \Rightarrow T$. We can avoid having a separate hierarchy of kinds by making the set of types a domain. Once we have $\mathrm{Kind}^T \in Dom$, it is natural to identify the function-space constructor $\Rightarrow$ on kinds with the function-

space constructor $\rightarrow$ on types, putting every Kind^κ into *Dom*. All of the retract models in Section 7.2 have this property. In the finitary projection model, with kinds as types and recursion over all types, Amadio, Bruce, and Longo (1986) have shown how to solve recursive domain equations. A natural question to ask is whether every second-order model $\mathscr{F}$ can be embedded in a model $\mathscr{G}$ with $\text{Kind}^T_{\mathscr{G}} \in Dom_{\mathscr{G}}$ so that $\mathscr{F}$ and $\mathscr{G}$ satisfy the same equations between second-order terms. Meyer and Reinhold have shown that adding a type of all types to the syntax of a related language has dramatic effects (Meyer and Reinhold, 1986), but this does not settle the question for second-order lambda calculus.

Since much of the interest in second-order lambda calculus stems from the similarity between the typing rules of the calculus and typing in programming languages like Ada, CLU, ML, and Russell, we expect the semantics of second-order lambda calculus to be useful for studying semantic properties of modern programming languages. One important property of typed programming languages is "representation independence," which has been studied by Reynolds and others (Donahue, 1979; Fokkinga, 1981; Haynes, 1984; Reynolds, 1974; Reynolds, 1983). Roughly speaking, representation independence ensures that the meaning of a program does not depend on whether the boolean value *true* is represented by 1 and *false* by 0, or vice versa. All that matters is that the operations on booleans behave properly. Two of the authors have studied representation independence for second-order lambda calculus using the model theory proposed in the present paper (Mitchell and Meyer, 1985; Mitchell, 1986a), proving general representation independence theorems. Another important topic in programming languages is full abstraction (Milner, 1977; Plotkin, 1977). While it is probably more difficult to construct fully abstract model for second-order lambda calculus than for ordinary typed lambda calculus (without polymorphism), this topic is well worth investigating.

ACKNOWLEDGMENTS

The authors would like to thank Giuseppe Longo, Eugenio Moggi, and Richard Statman for many helpful conversations.

RECEIVED January 10, 1989; FINAL MANUSCRIPT RECEIVED February 23, 1989

REFERENCES

AMADIO, R., BRUCE, K., AND LONGO, G. (1986), The finitary projection model for second order lambda calculus and solutions to higher order domain equations, *in* "IEEE Symp. Logic in Computer Science," pp. 122–130.

BARENDREGT, H. P. (1984), "The Lambda Calculus: Its Syntax and Semantics," rev. ed. North-Holland, Amsterdam.

BARENDREGT, H., AND REZUS, A. (1983), Semantics for classical automath and related systems, *Inform. and Control* **59**, Nos. 1–3.

BARENDREGT, H., COPPO, M., AND DEZANI-CIANCAGLINI, M. (1983), A filter lambda model and the completeness of type assignment, *J. Symbolic Logic* **48**, No. 4, 931–940.

BEESON, M. (1982), Recursive models for constructive set theories, *Ann. Math. Logic* **23**, 127–178.

BREAZU-TANNEN, V., AND MEYER, A. R. (1985), Lambda calculus with constrained types, *in* "Logics of Programs," pp. 23–40, Springer-Verlag, New York/Berlin.

BREAZU-TANNEN, V., COQUAND, T., GUNTER, C. A., AND SCEDROV, A. (1989), Inheritance and explicit coercion, *in* "Fourth IEEE Symp. Logic in Computer Science."

BRUCE, K., AND LONGO, G. (1985), Provable isomorphisms and domain equations in models of typed languages, *in* "17th ACM Symp. on Theory of Computing," pp. 263–272.

BRUCE, K., AND LONGO, G. (1988), A modest model of records, inheritance and bounded quantification, *in* "Third IEEE Symp. Logic in Computer Science," pp. 38–51.

BRUCE, K., AND MEYER, A. (1984), A completeness theorem for second-order polymorphic lambda calculus, *in* "Proceedings Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)," Lect. Notes in Comput. Sci., Vol. 173, pp. 131–144, Springer-Verlag, New York/Berlin.

CARDELLI, L., AND WEGNER, P. (1985), On understanding types, data abstraction, and polymorphism, *Comput. Surveys* **17**, No. 4, 471–522.

COPPO, M., AND ZACCHI, M. (1986), Type inference and logical relations, *in* "Proceedings, IEEE Symp. on Logic in Computer Science," pp. 218–226.

COQUAND, T., AND HUET, G. (1988), The calculus of constructions, *Inform. and Comput.* **76**, No. 2/3.

COQUAND, T., GUNTER, C. A., AND WINSKEL, G. (1989), Domain-theoretic models of polymorphism, *Inform. and Comput.* **81**, 123–167.

DE BRUIJN, N. G. (1980), A survey of the project Automath, *in* "To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism," pp. 579–607, Academic Press, New York.

U.S. Department of Defense (1980), "Reference Manual for the Ada Programming Language," GPO 008-000-00354-8.

DONAHUE, J. (1979), On the semantics of data type, *SIAM J. Comput.* **8**, 546–560.

FORTUNE, S., LEIVANT, D., AND O'DONNELL, M. (1983), The expressiveness of simple and second order type structures, *J. Assoc. Comput. Mach.* **30**, 151–185.

FOKKINGA, M. M. (1981), On the notion of strong typing, *in* "Algorithmic Languages" (De Bakker and van Vliet, Ed.), pp. 305–320, North-Holland, Amsterdam.

FRIEDMAN, H. (1975), Equality between functionals, *in* "Logic Colloquium," (R. Parikh, Ed.), pp. 22–37, Springer-Verlag, New York/Berlin.

GIRARD, J. Y. (1972), "Interpretation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur," These D'Etat, Université Paris VII.

GIRARD, J.-Y. (1986), The system *F* of variable types, fifteen years later, *Theoret. Comput. Sci.* **45**, No. 2, 159–192.

GRÄTZER, G. (1968), "Universal Algebra," Van Nostrand, New York.

HAYNES, C. T. (1984), A theory of data type representation independence, *in* "Proceedings, Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)," pp. 157–176, Lect. Notes in Comput. Sci. Vol. 173, Springer-Verlag, New York/Berlin.

HENKIN, L. (1950), Completeness in the theory of types, *J. Symbolic Logic* **15**, No. 2, 81–91.

HINDLEY, R. (1983), The completeness theorem for typing lambda terms, *Theoret. Comput. Sci.* **22**, 1–17.

KOYMANS, C. P. J. (1982), Models of the lambda calculus, *Inform. and Control* **52**, No. 3, 306–323.

KREISEL, G. (1959), Interpretation of analysis by means of constructive functionals of finite types, *in* "Constructivity in Mathematics," (A. Heyting, Ed.), pp. 101–128, North-Holland, Amsterdam.

LAMBEK, J. (1980), From lambda calculus to Cartesian closed categories, *in* "To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism," pp. 375–402, Academic Press, New York.

LANDIN, P. J. (1965), A correspondence between Algol 60 and Church's lambda notation, *Comm. ACM* **8**, 89–101, 158–165.

LEIVANT, D. (1983a), Polymorphic type inference, *in* "Proceedings, 10th ACM Symp. on Principles of Programming Languages," pp. 88–98.

LEIVANT, D. (1983b), Structural semantics for polymorphic types, *in* "Proceedings, 10th ACM Symp. on Principles of Programming Languages," pp. 155–166.

LISKOV, B., *et al.* (1981), "CLU Reference Manual," Lecture Notes in Computer Science, Vol. 114, Springer-Verlag, New York/Berlin.

MACQUEEN, D., AND SETHI, R. (1982), A semantic model of types for applicative languages, *in* "ACM Symp. on Lisp and Functional Programming," pp. 243–252.

MAC QUEEN, D., PLOTKIN, G., AND SETHI, R. (1986), An ideal model for recursive polymorphic types, *Inform. and Control* **71**, No. 1/2, 95–130.

MARTINI, S. (1988), Bounded quantifiers have interval models, *in* "ACM Conf. on LISP and Functional Programming," pp. 164–173.

MARTIN-LÖF, P. (1975), About models for intuitionistic type theories and the notion of definitional equality, *in* "3rd Scandinavian Logic Symposium," (S. Kanger, Ed.), pp. 81–109, North-Holland, Amsterdam.

MCCRACKEN, N. (1979), "An Investigation of a Programming Language with a Polymorphic Type Structure," Ph.D. thesis, Syracuse University.

MCCRACKEN, N. (1984a), The typechecking of programs with implicit type structure, *in* "Proceedings, Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)," Lect. Notes in Comput. Sci., Vol. 173, pp. 301–316, Springer-Verlag, New York/Berlin.

MCCRACKEN, N. (1984b), A finitary retract model for the polymorphic lambda calculus, manuscript.

MESEGUER, J. (1989), Relating models of polymorphism, *in* "Proceedings, 16th ACM Symp. on Principles of Programming Languages."

MEYER, A. R. (1982), What is a model of the lambda calculus?, *Inform. and Control* **52**, No. 1, 87–122.

MEYER, A. R., AND REINHOLD, M. B. (1986), Type is not a type, *in* "Proceedings, 13th ACM Symp. on Principles of Programming Languages," pp. 287–195.

MEYER, A. R., MITCHELL, J. C., MOGGI, E., AND STATMAN, R. (1987), Empty types in polymorphic lambda calculus, *in* "Proceedings, 14th ACM Symp. on Principles of Programming Languages," pp. 253–262.

MILNER, R. (1977), Fully abstract models of typed lambda calculi, *Theoret. Comput. Sci.* **4**, No. 1.

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

MITCHELL, J. C. (1984a), Coercion and type inference (Summary), *in* "Proceedings, 11th ACM Symp. on Principles of Programming Languages," pp. 175–185.

MITCHELL, J. C. (1984b), Semantic models for second-order lambda calculus, *in* "Proceedings, 25th IEEE Symp. on Foundations of Computer Science," pp. 289–299.

MITCHELL, J. C., AND PLOTKIN, G. D. (1988), Abstract types have existential types, *ACM Trans. Programming Languages Systems* **10**, Ni. 3, 470–502; *in* "Proc. 12th ACM Symp. on Principles of Programming Languages.

MITCHELL, J. C., AND MEYER, A. R. (1985), Second-order logical relations, *in* "Logics of

Programs," Lect. Notes in Comput. Sci., Vol. 193, pp. 225–236, Springer-Verlag, New York/Berlin.

MITCHELL, J. C. (1986a), Representation independence and data abstraction, *in* "Proceedings, 13th ACM Symp. on Principles of Programming Languages," pp. 263–276.

MITCHELL, J. C. (1986b), A type-inference approach to reduction properties and semantics of polymorphic expressions, *in* "ACM Conference on LISP and Functional Programming," pp. 308–319.

MITCHELL, J. C., AND MOGGI, E. (1987), Kripke-style models for typed lambda calculus, *in* "IEEE Symp. Logic in Computer Science," pp. 303–314; revised and expanded version, *J. Pure Appl. Logic*, in press.

MITCHELL, J. C. (1988), Polymorphic type inference and containment, *Inform. and Comput.* **76**, No. 2/3.

MOGGI, E. (1984), Internal category interpretation of second-order lambda calculus, manuscript.

MONK, J. D. (1976), "Mathematical Logic," Graduate Texts in Mathematics, Vol. 37, Springer-Verlag, New York/Berlin.

MYHILL, J. R., AND SHEPHERDSON, J. C. (1955), Effective operations on partial recursive functions. *Z. Math. Logik Grundlag. Math.* **1**.

PITTS, A. M. (1987), Polymorphism is set-theoretic, constructively, *in* "Proceedings, Summer Conf. on Category Theory and Computer Science," Lect. Notes in Comput. Sci., Springer-Verlag, New York/Berlin.

PLOTKIN, G. D. (1977), LCF considered as a programming language, *Theoret. Comput. Sci.* **13**.

PLOTKIN, G. (1985), Denotational semantics with partial functions, Lecture notes, C.S.L.I. Summer School, Stanford.

REYNOLDS, J. C. (1974), Towards a theory of type structure, *in* "Paris Colloq. on Programming," Lect. Notes in Comput. Sci., Vol. 19, pp. 408–425, Springer-Verlag, New York/Berlin.

REYNOLDS, J. C. (1981), The essence of algol, *in* "Algorithmic Languages," (de Bakker and van Vliet, Ed.), pp. 345–372, North-Holland, Amsterdam.

REYNOLDS, J. C. (1983), Types, abstraction, and parametric polymorphism, *in* "IFIP Congress."

REYNOLDS, J. C. (1984), Polymorpism is not set-theoretic, *in* "Proceedings, Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France)," Lect. Notes in Comput. Sci., Vol. 173, pp. 145–156, Springer-Verlag, New York/Berlin.

ROGERS, H. (1967), "Theory of Recursive Functions and Effective Computability," McGraw-Hill, New York.

ROSOLINI, G. (1986), "Continuity and Effectiveness in Topoi," Ph.D. thesis, Merton College, Oxford.

SCOTT, D. (1976), Data types as lattices, *Siam J. Comput.* **5**, No. 3, 522–587.

SCOTT, D. S. (1980), A space of retracts, manuscript, Merton College, Oxford.

SEELY, R. A. G. (1986), Categorical semantics for higher order polymorphic lambda calculus, manuscript, 1986.

SHAMIR, A., AND WADGE, W. (1977), Data types as objects, *in* "Proceedings, 4th ICALP Conference," Lect. Notes in Comput. Sci., Vol. 52, pp. 465–479, Springer-Verlag, New York/Berlin.

STATMAN, R. (1979), The typed lambda calculus is not elementary recursive, *Theoret. Comput. Sci.* **9**, 73–81.

STATMAN, R. (1981), Number theoretic functions computable by polymorphic programs, *in* "22nd IEEE Symp. on Foundations of Computer Science," pp. 279–282.

STATMAN, R. (1985), Equality between functionals, revisited, *in* "Harvey Friedman's Research on the Foundations of Mathematics," pp. 331–338, North-Holland, Amsterdam.

STENLUND, S. (1972), Combinators, λ-terms and Proof Theory, Reidel, Dordrecht.

TRAKHTENBROT, B. A., HALPERN, J. Y., AND MEYER, A. R. (1983), From denotational to operational and axiomatic semantics for algol-like languages: An overview, *in* "Logics of Programs, Proceedings," Lecture Notes in Computer Science, pp. 474–500, Springer-Verlag, New York/Berlin.

TOELSTRA, A. S. (1973), "Mathematical Investigation of Intuitionistic Arithmetic and Analysis," Lecture Notes in Mathematics, Vol. 344, Springer-Verlag, New York/Berlin.

Printed in Belgium