Semantic Models for Second-Order Lambda Calculus

John C. Mitchell[1]
MIT Lab. for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract

The second-order lambda calculus is a typed expression language with polymorphic functions and abstract data types. Several definitions of models for this language have been proposed, each relying on the syntax of terms to characterize closure under explicit definition. This paper aims to relieve the model theorist of syntactic considerations. Since it is natural to think of the second-order types $\forall t.\sigma(t)$ and $\exists t.\sigma(t)$ as the results of applying different operators to the "type function" $\lambda t.\sigma(t)$, the semantics of second-order lambda calculus naturally involves functions over the set of types. Second-order lambda calculus is studied here by considering a slightly more general *higher-order lambda calculus* $\mathcal{J}\mathbf{6}\Lambda$ with, e.g., functionals over type functions. A straightforward $\mathcal{J}\mathbf{6}\Lambda$-model definition is based on the model definition of [6] and a completeness theorem is proved. Models of $\mathcal{J}\mathbf{6}\Lambda$ are then characterized without reference to the meanings of terms. This characterization is formalized by introducing a *higher-order type theory* $\mathcal{J}\mathbf{6}\mathcal{T}$. Since the binding operators of $\mathcal{J}\mathbf{6}\Lambda$ are omitted from $\mathcal{J}\mathbf{6}\mathcal{T}$, the type theory $\mathcal{J}\mathbf{6}\mathcal{T}$ is essentially reducible to first-order logic.

1. Introduction

The second-order lambda calculus, discovered independently by Girard [12] and Reynolds [30], is an extension of the usual simple typed lambda calculus. Like other kinds of lambda calculus, the simple parameter-binding mechanism of this language corresponds closely to parameter binding in many program languages (cf. [17, 31, 37]). In contrast to other versions of lambda calculus, the type structure of the second-order system corresponds to the type structures of programming languages with polymorphism and data abstraction (such as Ada and CLU [1, 20]). Further discussion of polymorphism in second-order lambda calculus may be found in [10, 28, 18]. The correspondence between elements of existential types and concrete representations of abstract data types will be motivated and developed in a forthcoming joint paper with Gordon Plotkin. Some information about existential types is given in the final section of [32].

1. The author is supported by an IBM fellowship.

The *second order* in second-order lambda calculus comes about because the language is essentially a notational variant of the natural-deduction proof system for a second-order intuitionistic logic [10, 16]. Since there are several choices of basic connectives in logic, there are several proof systems for second-order logic, and hence several versions of second-order lambda calculus. Most of the previous semantic work has focused on the second-order lambda calculus based on the intuitionistic logic with implication ($\rightarrow$) and universal quantification ($\forall$). The other connectives, conjunction ($\wedge$) and disjunction ($\vee$), as well as existential quantification ($\exists$) are often omitted. In programming language parlance, function types and polymorphism have been studied, but pairing, tagged sums and abstract data types have been left out. The language $\mathcal{J}\mathbf{6}\Lambda$ will be defined with the set of type connectives as a parameter, so that we may study all variants of the language simultaneously.

Several definitions for models of second-order lambda calculus have been proposed [6, 19, 23] and a few specific model constructions have been presented [14, 25]. A companion paper to this one is [6], in which a general model definition is proposed and a completeness theorem proved. A major advantage of [6] over previous work is that all previously proposed model constructions (based on Scott's models of untyped lambda calculus) may be viewed as special cases of their general definition. The present paper broadens the "environment model" definition and completeness theorem of [6] to a more general language. Since all natural variants of second-order lambda calculus arise by choosing differents sets of constants for $\mathcal{J}\mathbf{6}\Lambda$, completeness theorems for each version of the language follow from the single theorem in Section 5.

The drawback of the environment model definition is that it can only be stated by referring to the meanings of terms. We would like to be able to study such notions as homomorphisms and submodels without constantly returning to arguments by induction on the structure of terms. To this end, we introduce a logical system $\mathcal{J}\mathbf{6}\mathcal{T}$ for axiomatizing models of $\mathcal{J}\mathbf{6}\Lambda$, without reference to definition by abstraction.

0272-5428/84/0000/0289$01.00 © 1984 IEEE

The axiomatization provides an "algebraic" characterization of second-order models similar to the characterization of untyped lambda calculus models based on combinatory algebra [26]. It is important to note that the semantics of $\mathcal{HOT}$ is straightforward. The fact that the model definition can be formalized in $\mathcal{HOT}$ should be taken as evidence of the simplicity of the definition, not the obscurity of $\mathcal{HOT}$.

By analogy with other forms of lambda calculus, [3, 26, 36], we do not expect the class of models of $\mathcal{H}\Lambda$ to be closed under homomorphisms. Instead, we expect the homomorphic images of models of $\mathcal{H}\Lambda$ to be similar to untyped lambda algebras [3, 26]. The higher-order type theory $\mathcal{HOT}$ provides a general framework for studying classes of typed structures which are analogous to untyped combinatory algebras or untyped lambda algebras. An intriguing research direction is to develop a deductive system for $\mathcal{HOT}$. We expect a completeness theorem for $\mathcal{HOT}$ along the lines of [15] to hold (see also Section 4 of [34]). The reason for this belief is that validity for $\mathcal{HOT}$ formulas can be reduced to first-order validity by essentially the method used for ordinary type theory (cf. [27], Chapter 30).

2. Higher-Order Lambda Calculus

There are three classes of expressions in the language $\mathcal{H}\Lambda$ of higher-order lambda calculus. We are primarily interested in the terms of $\mathcal{H}\Lambda$ that correspond to programming language expressions. Since each of these terms has a type, we also have type expressions. In building a general framework for studying various type structures, we also want to allow operations on types in the language. As in [22, 25], we associate a *kind* with each symbol that might appear in a type expression, and so we have the language of kinds. We begin by defining kinds.

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

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

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

$$c^\kappa \in \kappa, \quad v^\kappa \in \kappa$$

$$\mu \in \kappa_1 \rightarrow \kappa_2,\ \nu \in \kappa_1 \vdash \mu\nu \in \kappa_2$$

$$\mu \in \kappa_2 \vdash \lambda v^\kappa.\mu \in \kappa_1 \rightarrow \kappa_2$$

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

The pure equational theory of constructionals is the familiar equational theory of the simple typed lambda calculus (cf. [3, 11, 35]). In the absence of additional axioms, equality is decidable and every constructional has a β,η-normal form. Although it seems likely that the completeness theorem in Section 5 can be extended to allow arbitrary equations between constructionals as axioms, there are some syntactic difficulties with constructional axioms. In particular, the set of well-typed terms will depend on the set of equations between constructionals. To avoid this complication, we will only consider the pure theory of β, η-equality between constructionals.

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

r, s, t, ... denote type variables

ρ, σ, τ, ... denote type expressions.

As in the definition above, we will generally use μ and ν for constructionals. Some important constructional constants are

$$F \in T \rightarrow T \rightarrow T, \text{ and } \Pi, \Sigma \in [T \rightarrow T] \rightarrow T.$$

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

$$\sigma \rightarrow \tau ::= F\sigma\tau$$

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

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

We will always assume that $\mathcal{C}_{kind}$ includes the constants F and Π above.

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

$$Type_A(M) = \sigma, \ldots \vdash Type_A(N) = \tau.$$

meaning that if the antecedents hold, then the value of $Type_A$ at N is defined to be τ. The language $\mathcal{H}\Lambda_A$ of terms that are well-typed in context A is taken to be the domain of $Type_A$.

The typing functions $Type_A$ are defined by the following rules. If x is a variable, σ a type expression and A a type assignment, then $A[\sigma/x]$ is a

type assignment with $(\Lambda[\sigma/x])(y) = \Lambda(y)$ for any variable y different from x, and $(\Lambda[\sigma/x])(x) = \sigma$.

$Type_\Lambda(c^\tau) = \tau$

$Type_\Lambda(x) = \Lambda(x)$ if x is in the domain of Λ

$Type_\Lambda(M) = \sigma\to\tau$, $Type_\Lambda(N) = \sigma \vdash Type_\Lambda(\text{app } M\ N) = \tau$,

$Type_{\Lambda[\sigma/x]}(M) = \tau \vdash Type_\Lambda(\lambda x{\in}\sigma.M) = \sigma\to\tau$,

$Type_\Lambda(M) = \forall t.\sigma(t) \vdash Type_\Lambda(\text{proj } \tau\ M) = \sigma(\tau)$,

$Type_\Lambda(M) = \tau \vdash Type_\Lambda(\Pi t.M) = \forall t.\tau$,
provided t is not free in $\Lambda(x)$ for x free in M

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

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

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

Lemma 1: If t is bindable in $M{\in}\mathcal{GA}_\Lambda$ then $Type_\Lambda([\tau/t]M) = [\tau/t](Type_\Lambda(M))$.

Since it is cumbersome to write **app** and **proj**, we adopt the abbreviations

$MN ::= \textbf{app } M\ N$

$M\tau ::= \textbf{proj } \tau\ M$

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

2.1. Second-Order Lambda Calculus and Existential Types

We use $\mathcal{GA}(\to, \forall)$ to denote the basic language with constructional constants F, Π and no typed constants. The pure second order lambda calculus $\mathcal{PA}(\to, \forall)$ defined by Reynolds in [30] and studied in [6, 9, 10, 33], consists of all terms M of $\mathcal{GA}(\to, \forall)$ with the property that all free variables of M are either type variables or ordinary variables. The second order lambda calculus of [12] with product types and existential types as well is also a sublanguage of some $\mathcal{GA}$. We discuss existential types below.

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

$Type(\textbf{inj}_{\exists t.\sigma}) = \forall t[\sigma\to\exists t.\sigma]$,

$Type(\textbf{sum}_{\exists t.\sigma}) = \forall s[\forall t[\sigma\to s]\to\exists t.\sigma\to s]$

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

Instead of associating open types with constants, we index constants **inj** and **sum** by closed constructional of kinds $T^i\to T$, $i>0$, where $T^i\to T$ abbreviates the expression $T\to T\to\ldots\to T$ with i occurrences of $\to$. Note that for any type expression σ with all free variables of kind T, there is a closed constructional expression $\lambda\vec{r}.\ \lambda t.\sigma$ with kind $T^i\to T$ for some i. We add $\exists$-types to the language by adding constants $\textbf{inj}_f$ and $\textbf{sum}_f$ for each $f{\in}T^i\to T$ with types

$Type(\textbf{inj}_f) = \forall\vec{r}\{\forall t[f(\vec{r}, t)\to\exists t.f(\vec{r}, t)]\}$,

$Type(\textbf{sum}_f) = \forall\vec{r}\{\forall s[\forall t(f(\vec{r}, t)\to s)\to\exists t.f(\vec{r}, t)\to s]\}$

We use $\mathcal{GA}(\to, \forall, \exists)$ to denote the language with constructional constant Σ and typed constants $\textbf{inj}_f$ and $\textbf{sum}_f$. It is often convenient to write $\textbf{inj}_{\exists t.\sigma}$ as a meta-linguistic abbreviation for the term $\textbf{inj}_{\lambda\vec{r}.\sigma}\ \vec{r}$, and similarly for $\textbf{sum}_{\exists t.\sigma}$. Note that we only have constants $\textbf{inj}_{\exists t.\sigma}$ and $\textbf{sum}_{\exists t.\sigma}$ for each $\exists t.\sigma$ without free variables of higher kinds.

The additional constants of $\mathcal{GA}(\to, \forall, \exists)$ give us the following derived term formation rules.

$Type_\Lambda(M) = \sigma(\tau) \vdash Type_\Lambda(\textbf{inj}_{\exists t.\sigma}\ \tau\ M) = \exists t.\sigma(t)$

$Type_\Lambda(M) = \forall t.\sigma(t)\to\rho \vdash$
$Type_\Lambda(\textbf{sum}_{\exists t.\sigma}\ \rho\ M) = [\exists t.\sigma(t)]\to\rho$,
provided t is not free in ρ

Other second-order types such as $\wedge$ (pairing) and $\vee$ (disjoint sum) may be added to $\mathcal{GA}$ by making suitable choices of $\mathcal{C}_{kind}$ and $\mathcal{C}_{typed}$.

Some useful abbreviations in the language with existential types are

$\Sigma t.M ::= \textbf{sum}_{\exists t.\sigma}\ \rho\ \Pi t.M$,

abstype t with $x \in \sigma$ is M in N ::= $(\Sigma t\, \lambda x \in \sigma\ N)\ M$.

The operator Σ binds type vairables and is dual to Π. The **abstype** construct is a very general form of abstract data type declaration. A term of the form **inj** τ M may be considered an implementation or concrete representation of an abstract data type. Thus it is often a helpful mnemonic to write **rep** τ M for **inj** τ M.

3. Axioms and Inference Rules

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

(α_κ) $\lambda v^\kappa.\mu = \lambda v^\kappa.[u/v]\mu$, u not free in μ

(β_κ) $(\lambda v^\kappa.\mu)\nu = [\nu/v]\mu$

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

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

The pure equational theory of terms of $\mathcal{JC}\Lambda$ is based on α-conversion, renaming of bound variables, and β-conversion, a rule for evaluating expressions using substitution.

Axioms

(α_λ) $\lambda x \in \sigma.M = \lambda y \in \sigma.[y/x]M$, y not free in $\lambda x \in \sigma.M$,

(α_Π) $\Pi t.M = \Pi s.[s/t]M$, s not free in $\Pi t.M$,

(β_λ) $(\lambda x \in \sigma.M)N = [N/x]M$,

(β_Π) $(\Pi t.M)\tau = [\tau/t]M$,

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

Equivalence Rules

$M = N \vdash N = M$

$M = N, N = P \vdash M = P$

Congruence Rules

$M = N \vdash \lambda x \in \sigma.M = \lambda x \in \sigma.N$,

$\Pi t.M = \Pi t.N$, $M\tau = N\tau$

$M = N, P = Q \vdash MP = NQ$,

$\mu^k = \nu^k \vdash M = N$, where N is obtained from M by substituting ν for one or more occurrences of μ

An instance P=Q of an axiom is called an Λ-*instance* if $Type_\Lambda(P) = Type_\Lambda(Q)$. An equation M=N between terms of $\mathcal{JC}\Lambda_\Lambda$ is Λ-*provable*, written $\vdash_\Lambda M = N$, if the equation follows from Λ-instances of axioms other than the extensionality axioms, and the above inference rules. We often write $\vdash$ instead of $\vdash_\Lambda$ when the type assignment Λ is clear from context. It follows easily from the definition that if $Type_\Lambda(M) = \sigma$ and $\vdash_\Lambda M = N$, then $Type_\Lambda(N) = \sigma$.

A set Γ of $\mathcal{JC}\Lambda_\Lambda$-*equations* is a set of equations between terms of $\mathcal{JC}\Lambda_\Lambda$ such that $Type_\Lambda(M) = Type_\Lambda(N)$ for all $M = N \in \Gamma$. A *theory Th for* $\mathcal{JC}\Lambda_\Lambda$ is a set of $\mathcal{JC}\Lambda_\Lambda$ equations which is closed under $\vdash_\Lambda$. The theory of the axioms

Extensionality Axioms

(η_λ) $\lambda x \in \sigma.Mx = M$, x not free in M

(η_Π) $\Pi t(\mathbf{proj}\ t\ M) = M$, t not free in M

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

One important theory is the theory $Th_\exists$ for the language $\mathcal{JC}\Lambda(\rightarrow, \forall, \exists)$ with existential types. The axioms for this theory are motivated by the intended properties of existential types. Intuitively, existential types are similar to infinite sums in category theory [2, 21]. The general notion of sum involves sum types and sum functions. If Q(t) is a term or type expression, possibly with t free, then we use $Q(\tau)$ to denote the result of replacing free occurrences of t by τ. By definition, an infinite sum $\exists t.\sigma(t)$ of a family of types

$\{\ \sigma(\tau) \mid \tau$ a type expression$\}$,

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

$(\mathbf{inj}_{\exists t.\sigma}\ \tau) \in \sigma(\tau) \rightarrow \exists t.\sigma(t)$.

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

$(\Sigma t.N(t))\ (\mathbf{inj}_{\exists t.\sigma}\ \tau\ M) = N(\tau)\ M$.

We associate the set of functions $\{N(\tau)\}$ with the term $\Pi t.N$ and use sum $\Pi t.N$. for $\Sigma t.N$.

The theory $Th_\exists$ is the theory of the axiom

(β_Σ) $(\mathbf{sum}_{\exists t.\sigma}\ \rho\ M)(\mathbf{inj}_{\exists t.\sigma}\ \tau\ N) = (M\ \tau)N$,

based on the sum property above. The extensional theory for $\mathcal{JC}\Lambda(\rightarrow, \forall, \exists)$ is obtained by adding the axiom

(η_Σ) $(\Sigma t)(\lambda x \in \sigma(t).M(\mathbf{inj}_{\exists t.\sigma}\ t\ x) = M$,

provided t and x are not free in M,

to $Th_\exists$, along with (η_λ) and (η_Π).

4. Environment Models

Environment models will be defined in this section. We will later show that the environment model definition is equivalent to a combinatory model definition. In each form of model, we need to interpret both constructionals and terms. Since constructionals are essentially terms of the simple typed lambda calculus, their semantics are routine (see, e.g., [3, 11, 36]). The same straightforward, conventional model for

constructionals will be used in both environment models and combinatory models. We discuss the semantics of constructionals briefly before going onto the semantics of terms.

4.1. Models for Constructionals

A *type structure* $\mathcal{T}$ for a set $\mathcal{C}_{kind}$ of constructional constants is a tuple $\langle \mathcal{U}, c_1^{\mathcal{T}}, c_2^{\mathcal{T}}, \ldots \rangle$, where $\mathcal{U}$ is a family of sets $\{U_\kappa\}$ indexed by kinds with $U_{\kappa_1 \to \kappa_2}$ a set of functions from U_{κ_1} to U_{κ_2}, and the interpretation $c_i^{\mathcal{T}}$ of a constant $c_i \in \mathcal{C}_{kind}$ is an element of the appropriate U_κ. Since constructionals include all typed lambda expressions, $\mathcal{U}$ must be a model of the simple typed lambda calculus. This requires that for all kinds all κ_1, κ_2 and κ_3, there must be elements

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

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

with the familiar properties

$$K\ u\ v = u$$

$$S\ u\ v\ w = (u\ w)\ (v\ w)$$

where u, v and w are variables of the appropriate kinds. The meaning of a constructional μ in environment η for type structure $\mathcal{T}$ is defined as in [3, 11, 15, 36].

Note that we have defined type structures to be extensional models. This is not necessary, but follows most previous work on typed lambda calculus. It is interesting to note that type structures (models of propositionally-typed lambda calculus) are easily described by the above axiomatization of families of combinators K and S. This description is an axiomatization in the usual "simple" type theory of [11]; see also [15, 36].

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

4.2. Higher-Order Lambda Frames

The definition of environment model for $\mathcal{JC}\Lambda$ is based on the environment model definition for untyped lambda calculus, which we review briefly. The terminology below follows [26]. Untyped lambda calculus involves untyped applications MN and function expressions $\lambda x.M$. Thus we need to be able to treat elements as functions and functions as elements. This is accomplished using a pair of maps, the *element-to-function* map Φ and the *function-to-element* map Ψ. An *environment model for untyped lambda calculus* $\langle D, \Phi, \Psi \rangle$ consists of a set D together with functions Φ and Ψ. For some subset $[D \to D] \subseteq D^D$, we require that

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

$$\Phi \circ \Psi = id_{[D \to D]}.$$

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

In environment models of $\mathcal{JC}\Lambda$, we will need typed version of Φ and Ψ to interpret typed application and typed lambda abstraction. In addition, we will use "polymorphic" versions of Φ and Ψ for products $\Pi t.M$ and projection $\mathbf{proj}\ \tau\ M$.

A *frame* $\mathcal{F}$ *for* $\mathcal{JC}\Lambda(\mathcal{C}_{kind}, \mathcal{C}_{type})$ is a tuple

$$\mathcal{F} = \langle \mathcal{T}, \mathcal{D}, \{\Phi, \Psi, c^{\mathcal{F}}\}_{c \in \mathcal{C}_{typed}} \rangle$$

such that

(i) $\mathcal{T} = \langle \mathcal{U}, c^{\mathcal{T}}, \ldots \rangle$ is a type structure for $\mathcal{C}_{kind}$

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

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

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

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

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

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

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

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

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

An $\mathcal{JC}\Lambda$-*environment model* is an $\mathcal{JC}\Lambda$-frame which is sufficiently rich to allow all terms of $\mathcal{JC}\Lambda$ to be interpreted. More precisely, an $\mathcal{JC}\Lambda$-frame $\mathcal{F}$ satisfies (env) if for all environments η satisfying Λ (as defined below), the meaning function $[\![\bullet]\!]\eta$ specified below is a *total* function from $\mathcal{JC}\Lambda_\Lambda$ to elements of the domain of $\mathcal{F}$. An $\mathcal{JC}\Lambda$-environment model is an $\mathcal{JC}\Lambda$-frame satisfying (env).

4.3. Meanings of Terms

Let Λ be a type assignment. Let η an environment mapping $\mathcal{V}_{kind}$ to elements of the appropriate kinds, and $\mathcal{V}$ to elements of $\cup\mathcal{D}$. We say that η *satisfies* Λ, written $\eta \models \Lambda$, if

$$\eta(x) \in [\![\Lambda(x)]\!]\eta$$

for each variable $x \in \mathrm{dom}(\Lambda) \cap \mathcal{V}$.

The following lemma gives some straightforward, useful facts.

Lemma 2: Suppose $\eta \models \Lambda$. Then

(i) If $d \in D_{[\![\sigma]\!]\eta}$, then $\eta[d/x] \models \Lambda[\sigma/x]$

(ii) If t is bindable in M, then for any $a \in T$, we have $\eta[a/t] \models \Lambda|_{FV(M)}$.

If $\eta \models \Lambda$, then the meanings of expressions of $\mathcal{JC}\Lambda_A$ are defined inductively as follows.

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

$[\![c]\!]\eta = c^{\mathcal{F}}$

$[\![MN]\!]\eta = (\Phi_{a,b}[\![M]\!]\eta)\,[\![N]\!]\eta$,

where $a \to b$ is the meaning of $Type_A(M)$ in η,

$[\![\lambda x \in \sigma.M]\!]\eta = \Psi_{a,b}\, g$, where

$g(d) = [\![M]\!]\eta[d/x]$ for all $d \in D_a$ and

a, b are the meanings of σ and $Type_A(M)$ in η

$[\![M\tau]\!]\eta = (\Phi_f[\![M]\!]\eta)\,[\![\tau]\!]\eta$,

where Πf is the meaning of $Type_A(M)$ in η,

$[\![\Pi t.M]\!]\eta = \Psi_f\, g$, where

$g(a) = [\![M]\!]\eta[a/t]$ for all $a \in T$ and

$f \in [\Gamma \to T]$ is the function $[\![\lambda t.Type_A(M)]\!]\eta$

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

Lemma 3: Let η be an environment for a model $\langle \mathcal{T}, \mathcal{D}, ... \rangle$. If $M \in \mathcal{JC}\Lambda_A$ and $\eta \models \Lambda$, then $[\![M]\!]\eta \in D_{[\![Type_A(M)]\!]\eta}$.

A very useful fact is the following Substitution Lemma.

Lemma 4: (Substitution) (i) Let M and N be terms of $\mathcal{JC}\Lambda_A$ and x an ordinary variable. Suppose $\eta \models \Lambda$ and $Type_A(N) = \Lambda(x)$. Then

$[\![[N/x]M]\!]\eta = [\![M]\!]\eta[[\![N]\!]\eta/x]$.

(ii) Furthermore, if t is bindable in M and then

$[\![[\sigma/t]M]\!]\eta = [\![M]\!]\eta[[\![\sigma]\!]\eta/t]$.

(iii) If μ, ν^κ are constructionals and $\nu^\kappa \in \mathcal{V}_{kind}$ 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 simply-typed lambda calculus. A special case of the Substitution Lemma is that the meaning $[\![M]\!]\eta$ of a term M in environment η does not depend on $\eta(x)$ or $\eta(t)$ for x or t not free in M.

5. Completeness

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

Lemma 5: (Soundness) Let Γ be a set of $\mathcal{JC}\Lambda_A$ equations and let M, $N \in \mathcal{JC}\Lambda_A$. If $\Gamma \vdash_A M = N$, then $\Gamma \models M = N$.

We also have the following completeness theorem.

Theorem 1: (Completeness) Let Γ be a set of $\mathcal{JC}\Lambda_A$ equations and let M, $N \in \mathcal{JC}\Lambda_A$. If $\Gamma \models_A M = N$, then $\Gamma \vdash M = N$.

The proof is omitted due to limitations of space.

6. Higher-Order Type Theory and Combinatory Algebras

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

Typed $\mathcal{JC}$-combinatory algebras and $\mathcal{JC}\Lambda$-combinatory models are analogous to untyped combinatory algebras and combinatory models. We review the untyped definitions briefly using terminology similar to [26]. An untyped applicative structure $\mathfrak{D} = \langle D, \bullet \rangle$ consists of a set D with a binary operation $\bullet$. As in group theory, for example, it is customary to omit the operation $\bullet$ from expressions. An untyped applicative structure is *combinatorially complete* if, for every polynomial $p(x_1, ..., x_n)$ over D with indeterminates among $x_1, ..., x_n$, there is a constant $d \in D$ such that

$\mathfrak{D} \models p(x_1, ..., x_n) = dx_1...x_n$.

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

$K\,x\,y = x$

$S\,x\,y\,z = (x\,z)(y\,z)$;

see [3, 26]. Thus a combinatory algebra is an applicative frame satisfying two equational axioms.

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

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

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

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

Untyped lambda abstraction can now be interpreted

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

$de = [\![M]\!]\eta[e/x]$ for all $e \in D$.

Since $\langle D, \bullet\rangle$ is a combinatory algebra, such an $e \in D$ will exist for any M. Furthermore, weak extensionality follows from the properties of ε. A comprehensive discussion of the equivalence between the environment and combinatory model definitions for untyped lambda calculus is given in [26]. Note that since the combinatory algebra axioms are equational, combinatory algebras form an algebraic variety [13]. In contrast, the axioms for ε are not equational.

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

6.1. Higher-Order Type Theory

An applicative term is a term without any occurrences of λ or Π. The formulas of $\mathcal{HT}_A$ are defined by

$$G ::= M = N \mid G \wedge H \mid \neg G \mid \forall v^{\kappa}.G \mid \forall x \in \sigma.G,$$

where M and N are applicative terms of $\mathcal{HC}\Lambda_A$ with $Type_A(M) = Type_A(N)$. Since only applicative terms appear in formulas of $\mathcal{HT}$, we do not need the Ψ functions of $\mathcal{HC}\Lambda$-frames to interpret formulas. Instead, we use the simpler applicative frames described below. Formulas of $\mathcal{HT}$ are interpreted by giving the logical connectives $\wedge$ and $\neg$ their usual meaning, and by interpreting quantifiers as ranging over the appropriate sets of the frame.

6.2. Applicative Frames

An *applicative frame for* $\mathcal{HC}\Lambda_A(\mathcal{C}_{kind}, \mathcal{C}_{typed})$ is a tuple

$$\mathcal{F} = \langle \mathcal{T}, \mathcal{D}, \{\bullet, c^{\mathcal{F}}, ...\}\rangle$$

with

(i) $\mathcal{T} = \langle \mathcal{U}, c^{\mathcal{T}}, ...\rangle$ a type structure for $\mathcal{C}_{kind}$

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

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

$\bullet_{a,b} : D_{Fab} \to D_a \to D_b$,

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

$\bullet_f : D_{\Pi f} \to \Pi_{a \in T} D_{fa}$

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

Both $\bullet_{a,b}$ and $\bullet_f$ will be written as infix operations. When the types are clear from context, we will omit the "$\bullet$" and write, e.g., de for $d \bullet_{a,b} e$.

Every second-order frame $\langle \mathcal{T}, \mathcal{D}, \{\Phi, \Psi, c^{\mathcal{F}}, ...\}\rangle$ has an *associated applicative frame* $\langle \mathcal{T}, \mathcal{D}, \{\bullet, c^{\mathcal{F}}, ...\}\rangle$ defined by taking

$d \bullet_{a,b} e = (\Phi_{a,b}\, d)\, e$

$d \bullet_f a = (\Phi_f d)\, a$

and preserving the interpretations of constants.

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

6.3. Combinatory Completeness

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

$$\mathcal{F} \models M = N\, \vec{s}\, \vec{x}.$$

This definition is similar to the usual definition of combinatory completeness for untyped lambda calculus [3, 26], but with the added consideration of type variables. We do consider variables of higher kinds since terms of $\mathcal{HC}\Lambda$ only include λ-binding of ordinary variables and Π-binding of type variables, and cannot be abstracted with respect to free variables of higher kinds.

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

$I \in \forall t.t \to t$

$K \in \forall s,t.s \to t \to s$

$S \in \forall r,s,t.(r \to s \to t) \to (r \to s) \to r \to s$

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

$A_f \in \forall r, \vec{s}.[(r \to \forall t.f(\vec{s}, t)) \to \forall t.(r \to f(\vec{s}, t))$

$B \in \forall r[r \to \forall t.r]$

$C_{f,g} \in \forall \vec{r}, \vec{s}\ [\forall t(f(\vec{r}, t) \to g(\vec{s}, t)) \to$

$\forall t.f(\vec{r}, t) \to \forall t.g(\vec{s}, t)]$

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

satisfying the $\mathcal{HT}$ axioms below. Note that each combinator has a closed type.

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

$I\ t\ x = x$

$K\ s\ t\ x\ y = x$

$S\ r\ s\ t\ x\ y\ z = x\ z\ (y\ z)$

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

$\forall t \in T\ \forall x \in t.\ I\ t\ x = x$

Each A, B, C and D must satisfy the typed universal closure of the appropriate equational axiom below.

$(A_f\ r\ \vec{s})\ x\ t\ y = (x\ y)\ t$

$(B\ r)\ x\ t = x$

$(C_{f,g}\ \vec{r}, \vec{s})\ x\ y\ t = x\ t\ (y\ t)$

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

A second-order frame *has combinators* if the associated applicative frame has combinators. It is worth noting that this set of combinators has been chosen for ease in proofs and is not intended to be a minimal set of combinators.

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

$\langle x \in \sigma \rangle x = I\ \sigma$

$\langle x \in \sigma \rangle y = K\ \tau\ \sigma\ y$, where $\tau = Type_A(y)$,

$\langle x \in \sigma \rangle c = K\ \tau\ \sigma\ c$, where τ is the type of constant c

$\langle x \in \sigma \rangle (MN) = S\ \sigma\ \tau\ \rho\ (\langle x \in \sigma \rangle M)\ (\langle x \in \sigma \rangle N)$

where $\tau = Type_A(M)$ and $\rho = Type_A(N)$,

$\langle x \in \sigma \rangle (M\rho) = (\Lambda_f\ \sigma\ \vec{s})\ (\langle x \in \sigma \rangle M)\ \rho$

where $\forall t.\tau = Type_A(M)$, all variables free in $\forall t.\tau$ are among $\vec{s}$, and $f = \lambda \vec{s}, t.\ \tau$.

If t is bindable in the applicative term M of $\mathcal{H}\Lambda(\mathcal{F})_A$, we define $\langle t \rangle M$ by

$\langle t \rangle y = B\ \tau\ y$, where $\tau = Type_A(y)$,

$\langle t \rangle c = B\ \tau\ c$, where τ 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 $f = \lambda \vec{r}, t.\ \sigma$ and $g = \lambda \vec{s}, t.\ \tau$ are closed and $\sigma \to \tau = Type_A(M)$ and $\sigma = Type_A(N)$,

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

where $h = \lambda \vec{r}, t, u.\ \sigma$ and $f = \lambda \vec{s}, t.\ \tau$ are closed constructionals and $\forall u.\sigma = Type_A(M)$.

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

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

$(\langle x \in \sigma \rangle M)\ N = [N/x]M$

$(\langle t \rangle M)\ \sigma = [\sigma/t]M$.

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

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

7. Combinatory Models

Just as untyped combinatory algebras are not suitable models of untyped lambda calculus, $\mathcal{H}$-combinatory algebras are not suitable for interpreting terms of $\mathcal{H}\Lambda$. In order to satisfy the weak extensionality property of $\mathcal{H}\Lambda$, we need a family of "choice elements" ε analogous to the single untyped ε of [26]. The typed ε's are similar to the κ and θ of [19].

7.1. Combinatory Model Definition

A *second-order combinatory model for* $\mathcal{H}\Lambda$ is a second-order combinatory algebra for $\mathcal{H}\Lambda$ such that

(vi) For each a,b ∈ T, there is an element $\varepsilon_{a,b} \in D_{(a \to b) \to (a \to b)}$ such that

$\forall x \in a \to b, y \in a.\ (\varepsilon_{a,b} x)y = xy$

$\forall x, y \in a \to b.\ \forall z \in a(xz = yz) \supset \varepsilon_{a,b} x = \varepsilon_{a,b} y$

$\varepsilon_{a \to b,\, a \to b}\ \varepsilon_{a,b} = \varepsilon_{a,b}$

(vii) For every $f \in [T \to T]$, there is an element $\varepsilon_f \in D_{\Pi f \to \Pi f}$ such that

$\forall x \in \Pi f, \forall t.\ (\varepsilon_f x)t = xt$

$\forall x, y \in \Pi f.\ \forall t(xt = yt) \supset \varepsilon_f x = \varepsilon_f y$

$\varepsilon_{\Pi f, \Pi f}\, \varepsilon_f = \varepsilon_f$

Note that the axioms for $\varepsilon_{a,b}$ and ε_f, as well as the axioms for combinators, are formulas of $\mathcal{HT}$.

If $\eta \models A$, then the meanings of expressions of $\mathcal{H}\Lambda_A$ in a combinatory model $\langle \mathcal{T}, \mathcal{D}, ... \rangle$ are defined inductively as follows.

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

$[\![c]\!]\eta = c^{\mathcal{T}}$,

$[\![M\ N]\!]\eta = [\![M]\!]\eta \bullet_{a,b} [\![N]\!]\eta$,

where $a \to b$ is the meanings of $Type_A(M)$ in η,

$[\![\lambda x \in \sigma.M]\!]\eta = \varepsilon_{a,b} \bullet d$, where d satisfies

$d \bullet e = [\![M]\!]\eta[e/x]$ for all $e \in D_a$ and

a, b are the meanings of σ and $Type_A(M)$ in η

$[\![M\ \tau]\!]\eta = [\![M]\!]\eta \bullet_f [\![\tau]\!]\eta$,

where Πf is the meaning of $Type_A(M)$ in η,

$[\![\Pi t.M]\!]\eta = \varepsilon_f \bullet d$, where d satisfies

$d \bullet_f a = [\![M]\!]\eta[a/t]$ for all $a \in T$ and

$f \in [T \to T]$ is the function $[\![\lambda t.Type_A(M)]\!]\eta$

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

Theorem 3: If $\mathcal{T}$ is a combinatory model for $\mathcal{H}\Lambda$, M is a term of $\mathcal{H}\Lambda_A$ and η respects Λ, then $[\![M]\!]\eta$ is a uniquely defined element of $\mathcal{T}$ of the appropriate semantic type.

Thus combinatory models are models.

7.2. Combinatory Model Theorem

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

If $\mathcal{A} = \langle \mathcal{T}, \mathcal{D}, \{\Phi, \Psi, c^{\mathcal{A}}, ...\} \rangle$ is an environment model, we define an applicative frame with ε's

$\mathcal{A}_{comb} = \langle \mathcal{T}, \mathcal{D}, \{\bullet, \varepsilon, c^{\mathcal{A}}, ...\} \rangle$

interpreting the same constants as follows.

For $a,b \in T$, define $\bullet_{a,b}$ and $\varepsilon_{a,b}$ by

$d \bullet_{a,b} e = (\Phi_{a,b}\, d)\, e$

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

For $f \in [T \to T]$, define $\bullet_f$ and ε_f by

$d \bullet_f a = (\Phi_f\, d)\, a$

$\varepsilon_f = \lambda x \in \forall t.f(t)\ \Pi s.\ x\ s$

It is easy to verify that $\bullet$ and ε have the required properties. Note that the above ε definitions use terms of $\mathcal{H}\Lambda(\mathcal{A})$ to name elements of $\mathcal{A}$.

Conversely, if $\mathcal{A} = \langle \mathcal{T}, \mathcal{D}, \{\bullet, \varepsilon, c^{\mathcal{A}}, ...\} \rangle$ is a combinatory model, we define an $\mathcal{H}\Lambda$-frame

$\mathcal{A}_{env} = \langle \mathcal{T}, \mathcal{D}, \{\Phi, \Psi, c^{\mathcal{A}}, ...\} \rangle$

interpreting the same constants as follows.

For each a, $b \in T$, let

$[D_a \to D_b] = \{g_d : D_a \to D_b \mid d \in D_{Fab}\}$,
where $g_d(e) = d \bullet_{a,b} e$ for all $e \in D_a$

$\Phi d = g_d \in [D_a \to D_b]$ as above, and

$\Psi g_d = \varepsilon d$.

For each $f \in [T \to T]$, let

$[\Pi_{a \in T} D_{fa}] = \{h_d : \Pi_{a \in T} D_{fa} \mid d \in D_{\Pi f}\}$,
where $h_d(a) = d \bullet_f a$ for all $a \in T$

$\Phi d = h_d \in [\Pi_{a \in T} D_{fa}]$ as above, and

$\Psi h_d = \varepsilon d$.

The properties of $\bullet$ and ε may be used to show that Φ and Ψ have the required properties.

We show that $\mathcal{A}_{comb}$ has combinators. Consider the following terms of $\mathcal{H}\Lambda$, written using variables f, g and h of higher kinds.

$I^{\Lambda} = \Pi t\, \lambda x \in t.\ x$

$K^{\Lambda} = \Pi s,t\, \lambda x \in s, y \in t.\ x$

$S^{\Lambda} = \Pi r,s,t\, \lambda x \in r \to s \to t, y \in r \to s, z \in r.\ x\ z\ (y\ z)$

$A_f^{\Lambda} = \Pi r, \vec{s}\,[\lambda x \in (r \to \forall t.f(\vec{s}, t).\ \Pi t.\ \lambda y \in r.\ x\ y\ t]$,

$B^{\Lambda} = \Pi r.\ \lambda x \in r.\ \Pi t.\ x$

$C_{f,g}^{\Lambda} = \Pi \vec{r}, \vec{s}\,[\lambda x \in \forall t(f(\vec{r}, t) \to g(\vec{s}, t)).$

$\lambda y \in \forall t.f(\vec{r}, t).\ \Pi t.(x\ t)(y\ t)]$

$D_{h,f}^{\Lambda} = \Pi \vec{r}, \vec{s}\,[\lambda x \in \forall t, u.\ h(\vec{r}, t, u).\ \Pi t.\ x\ t\ f(\vec{s}, t)]$

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

To see that $\mathcal{A}_{env}$ satisfies condition (env), note that by Theorem 6, every term has a meaning in combinatory model $\mathcal{A}$. A straightforward induction on terms shows that that for any term $M \in \mathcal{H}\Lambda_A$ and environment $\eta \models A$, the meaning $\mathcal{A}_{env}[\![M]\!]\eta$ exists and is equal to $\mathcal{A}[\![M]\!]\eta$. Thus we have the following combinatory model theorem, analogous to the combinatory model theorem of [26].

> **Theorem 4:** If $\mathcal{A}$ is an environment model, then $\mathcal{A}_{comb}$ is a combinatory model with the same equational theory as $\mathcal{A}$ and $(\mathcal{A}_{comb})_{env} = \mathcal{A}$. Similarly, if $\mathcal{A}$ is a combinatory model, then $\mathcal{A}_{env}$ is an environment model with the same equational theory as $\mathcal{A}$ and $(\mathcal{A}_{env})_{comb} = \mathcal{A}$.

8. Definability of Existential Types

In formal logic, it is common to adopt a small number of logical constants as basic, introducing the remaining constructs by definition. In intuitionistic second-order logic, for example, we may take $\rightarrow$ and $\forall$ as basic, since Prawitz [29] has shown that these are sufficient to define all others. As noted in the Introduction, there is a close connection between the types of second-order lambda calculus (or $\mathcal{H}\Lambda$) and second-order formulas. This suggests that we may be able to define some type connectives from others, a suggestion implicit in [32].

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

9. Directions for Further Investigation

The main open problem in the semantics of second-order lambda calculus is to determine whether there exist natural mathematics models that are not constructed using universal domains. Recent work by Reynolds [33] suggests that certain unanticapated isomorphisms between types may be forced in all models. Reynolds shows that in every "set-theoretic" model, there is some type S which is isomorphic to $(S \rightarrow B) \rightarrow B$ for some nontrivial B. This is a set-theoretic contradiction. However, it is not clear whether some form of Reynolds' argument may be applied to all models. If some form of Reynolds' argument applies generally, then elementary models may not exist.

An important problem related to the problem of constructing models is to characterize the isomorphisms between types (or retracts of types) that must hold in all models. In particular, is there a type τ such that $(\tau \rightarrow B) \rightarrow B$ must be isomorphic to τ in all models (where B is some nontrivial type such as $\forall t.(t \rightarrow t) \rightarrow t \rightarrow t$)? Recently, Kim Bruce and G. Longo [5] have characterized a class of "definable" isomorphisms that must hold in every model, but other kinds of isomorphism may also be important, Some other important directions for futher research are listed below.

1. Investigate model constructions and relationships between models.
2. Is there a nontrivial finite model (i.e. a model in which every D_a is finite)?
3. Investigate theories involving equations between constructionals.
4. The language $\mathcal{H}\Lambda$ only allows binding of ordinary variables and type variables. Investigate the extension of $\mathcal{H}\Lambda$ which allows any term to be treated as a function of any variable v^κ of any kind.
5. Do models of $\mathcal{H}\Lambda$ shed any light on models for Martin-Löf's constructive theory [24]? Automath [4, 8] or the related calculus developed by Huet and Coquand [7]?
6. Develop a deductive system for $\mathcal{H}\mathcal{T}$. A general investigation of $\mathcal{H}\mathcal{T}$ may prove useful in further research into $\mathcal{H}\Lambda$ and similar systems.

Acknowledgments: This paper draws on previous work by Kim Bruce and Albert Meyer. I thank both of them for many helpful discussions.

References

1. US Dept. of Defense. *Reference Manual for the Ada Programming Language.* GPO 008-000-00354-8, 1980.

2. Arbib, M.A. and E.G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, 1975.

3. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1981.

4. Barendregt, H. and Rezus, A. "Semantics for Classical AUTOMATH and Related Systems". *Information and Control* (1984). to appear..

5. Bruce, K. and Longo, G. Definable Isomorphisms and Domain Equations in the Second-Order Lambda Calculus. Unpublished Manuscript.

6. Bruce, K. and Meyer, A. A Completeness Theorem for Second-Order Polymorphic Lambda Calculus. Proc. Int. Symp. on Semantics of Data Types, Sophia-Antipolis (France), 1984, pp. 131-144..

7. Coquand, T. and Huet, G. A Theory of Constructions. Proc. Int'l Symp. on Semantics of Data Types, June, 1984. Paper does not appear in proceedings..

8. De Bruijn, N.G. A survey of the project AUTOMATH. In *To H.B. Curry: Essays on Combinatory Logic*, Seldin, J.P. and J.R. Hindley, Eds., Academic Press, 1980, pp. 579-607.

9. Donahue, J. "On the semantics of data type". *SIAM J. Computing 8* (1979), 546-560.

10. Fortune, S., Leivant, D. and O'Donnel, M. "The Expressiveness of Simple and Second Order Type Structures". *JACM 30*, 1 (1983). pp 151-185.

11. Friedman, H. Equality Between Functionals. In R. Parikh, Ed., *Logic Colloquium*, Springer-Verlag, 1975, pp. 22-37.

12. Girard, J.-Y. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *2nd Scandinavian Logic Symp.*, Fenstad, J.E., Ed., North-Holland, 1971, pp. 63-92.

13. Gratzer, G.. *Universal Algebra.* Van Nostrand, 1968.

14. Haynes, C.T. *A Theory of Data Type Representation Independence.* Ph.D. Th., Univ. of Iowa, Dept. of Computer Science, 1982. Technical Report 82-04..

15. Henkin, L. "Completeness in the Theory of Types". *Journal of Symbolic Logic 15*, 2 (June 1950). pp 81-91.

16. Howard, W. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Seldin, J.P. and J.R. Hindley, Eds., Academic Press, 1980, pp. 479-490.

17. Landin, P.J. "A Correspondence Between ALGOL 60 and Church's Lambda Notation". *CACM 8* (1965). pp 89-101; 158-165..

18. Leivant, D. The Complexity of Parameter Passing in Polymorphic Procedures. 13-th ACM Symposium on Theory of Computing, May, 1981. pp 38-45..

19. Leivant, D. Structural Semantics for Polymorphic Types. Proc. 10-th ACM Symp. on Principles of Programming Languages, 1983, pp. 155-166.

20. Liskov, B. et. al.. *Lecture Notes in Computer Science.* Volume 114: *CLU Reference Manual.* Springer-Verlag, 1981.

21. Mac Lane, S.. *Graduate Texts in Mathematics.* Volume 5: *Categories for the Working Mathematician.* Springer-Verlag, 1971.

22. MacQueen, D. and Sethi, R. A Semantic Model of Types for Applicative Languages. ACM Symp. on LISP and Functional Programming, 1982, pp. 243-252..

23. Martin-Löf, P. About models for intuitionistic type theories and the notion of definitional equality. 3rd Scandinavian Logic Symp., 1975, pp. 81-109.

24. Martin-Löf, P. Constructive mathematics and computer programming. Paper presented at 6th International Congress for Logic, Methodology and Philosophy of Science, Preprint, Univ. of Stockholm, Dept. of Math. 1979.

25. McCracken, N. *An Investigation of a Programming Language with a Polymorphic Type Structure.* Ph.D. Th., Syracuse Univ., 1979.

26. Meyer, A.R. "What Is A Model of the Lambda Calculus ?". *Information and Control 52*, 1 (1982). pp 87-122..

27. Monk, J. D.. *Graduate Texts in Mathematics.* Volume 37: *Mathematical Logic.* Springer-Verlag, 1976.

28. O'Donnell, M. A practical programming theorem which is independent of Peano arithmetic. 11th ACM Symp. on the Theory of Computation, 1979, pp. 176-188.

29. Prawitz, D.. *Natural Deduction.* Almquist and Wiksell, Stockholm, 1965.

30. Reynolds, J.C. Towards a Theory of Type Structure. Paris Colloq. on Programming, 1974, pp. 408-425.

31. Reynolds, J.C. The Essence of ALGOL. Inter. Symp. on Algorithmic Languages, 1981, pp. 345-372..

32. Reynolds, J.C. Types, Abstraction, and Parametric Polymorphism. IFIP Congress, 1983, pp. .

33. Reynolds, J.C. Polymorphism is not Set-Theoretic. Int. Symp. on Semantics of Data Types, 1984, pp. 145-156.

34. Scott, D.S. Relating theories of the lambda calculus. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Seldin, J.P. and J.R. Hindley, Eds., Academic Press, 1980, pp. 403-450.

35. Statman, R. "The Typed Lambda Calculus is Not Elementary Recursive". *Theoretical Computer Science 9* (1979). pp 73-81..

36. Statman, R. Equality Between Functionals, Revisited. Friedman Volume, to appear.

37. Trakhtenbrot, B.A., J.Y. Halpern and A.R. Meyer. From Denotational to Operational and Axiomatic Semantics for ALGOL-like Languages: An Overview. Logics of Programs, Proceedings, Lecture Notes in Computer Science, Springer, 1983.