MIT/LCS/TR-325

USING UNTYPED LAMBDA CALCULUS TO COMPUTE WITH ATOMS

Paul G. Weiss

Using Untyped Lambda Calculus To Compute With Atoms

by

Paul G. Weiss
B.S., Massachusetts Institute of Technology
(1981)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE
DEGREE OF

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 1984

Signature of Author Paul Weiss
Department of Electrical Engineering and Computer Science
February 28, 1984

Certified by [illegible]
Albert R. Meyer
Thesis Supervisor

Accepted by ____________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Keywords: Lambda calculus, typeless, untyped, atoms, completeness

Abstract: Axioms and verification rules are given for typeless λ-calculus with a conditional test for equality between atoms. A semantic completeness theorem is proved and a deterministic evaluator is proposed.

1. Introduction

The λ-calculus of Church (treated throughly in [BAR80]) is a system for denoting functions. For example, the identity function is represented in this system as $\lambda x.x$, and the function which adds 1 to its argument is represented as $\lambda x.x + 1$. A more complicated example is the "double application" functional, whose arguments are themselves functions, and which acts by composing a function with itself. This is represented in the λ-calculus as $\lambda f.\lambda x.f(fx)$.

For many years a model for the λ-calculus could not be found, due to set theoretical difficulties. Finally, Scott was able to construct a structure which was generally agreed to be a model, using complete lattices [SCOTT76]. Many attempts were then made to give a clean characterization of what a model of the λ-calculus was, these are detailed in [BAR80] and in [MEYER82].

These systems all have the property that any term can be interpreted as a function. This is necessary, since a model of the untyped λ-calculus must make sense out of the application of any term to any other term. In fact any term can also be interpreted as a functional, that is, a function which maps functions to functions, and so on up through the type hierarchy. But this is not the behavior we want when we are using λ-calculus to compute with integers.

The problem is that there is nothing to distinguish the integers from the other terms. Suppose we use a λ-calculus with constants for the integers and successor, suitably axiomatized. Then in any model, while it is true that the values of integers will behave correctly with respect to the value of successor, it is also true that the value of 3 applied to the value of 4 will be some value, and there is nothing in the language or the model that tells us that this is any different than successor applied to 3. This is not what we want. We want constants such as 3 to denote *atomic values* in all models. These are values that cannot be applied to anything without yielding an error. The constants that are used to denote atomic values will be called *numerals*.

One method for computing with atomic values in the λ-calculus is to add type information to the terms, to tell what kind of datum each subterm represents. This is approach taken in the typed λ-calculus. In order for one term to be applied to a second, the type of the first term must be functional, with the argument type matching the type of the second term. Terms representing atomic values do not have functional type, and therefore cannot be applied to anything. Typed λ-calculus is dealt with thoroughly in an appendix to [BAR80].

In this treatment, we use a different approach to type errors. We will allow arbitrary applications in the language, however, certain terms will lead to *run-time type errors* when evaluated. Our λ-calculus will be untyped, and we will provide semantics so that the terms which lead to run-time type errors are precisely those terms which semantically denote an error value. We are motivated throughout by the language LISP, which has a λ-calculus like syntax, but expresses computation on objects which do not necessarily denote functions (atoms and lists). See [WAND84] for a discussion of LISP.

Since we are using untyped λ-calculus, we will be able to draw on the results of [MEYER82], to provide a model. A system with error values for run-time type errors was also considered in [MILNER78], with a complete partial order semantics.

In order to do useful computation with numerals, we will find that a conditional statement is needed. This will take the form: "if <term1>=<term2> then <term3> else <term4>." Without this construct, the expressive power is greatly reduced. However, there are many choices to be made in the behavior of this construct. Do we evaluate <term1> and <term2> sequentially, or in parallel? What happens if the evaluation of <term1> or <term2> leads to a run-time type error? Also, what notion of equality between terms do we use? The most strict notion is identity. Another notion is provable equality (under some suitable axioms and proof rules). We will try to make choices that will result in a recursive evaluator and simple axioms for the proof system, while still giving us enough expressive power for programming.

The language considered is an untyped λ-calculus, with a conditional statement and error terms. The proof system is that of the classical λ-calculus enriched with axioms to handle these new constructs, and to handle the properties of numerals.

The class of models for this language is a special case of combinatory models [MEYER82]. A completeness theorem for the language is derived from the completeness theorem for the classical λ-calculus[MEYER82].

2. Syntax

We will define an untyped λ-calculus for computing with atoms. Our language will be an extension of the classical λ-calculus of Church. Since we have occasion to refer to the classical λ-calculus, we define it here.

Definition: Let **Var** be an infinite set of variables. Let C be a set of constants. We define the set of terms $\Lambda(C)$ by the following grammar, where t denotes an element of $\Lambda(C)$, x denotes an element of **Var**, and c denotes an element of C:

$$t ::= x \mid c \mid t_1 t_2 \mid \lambda x.t\,.$$

We omit parentheses in the usual fashion. In particular, uvw abbreviates $(uv)w$, and $\lambda xy.u$ abbreviates $\lambda x\,\lambda y.u$.

We now extend the language to express computations with atoms. An *atom* is a semantic object, which cannot be applied to anything else without yielding an error. An example of an atom might be the number 3 or 17, if we are talking about integers, or perhaps the list nil if we are talking about lists. In order to represent atoms in our language we introduce *atomic constants.* These are a special type of constant whose meaning can only be an atom.

These are the base syntactic sets:

Let **Var** be a set of variables. Typical elements are x, y, z.

Let **Con** be a set of constants. Typical elements are $c_1, c_2, \ldots$

Let **ACon** be another set of constants (the atomic constants). Typical elements are $a_1, a_2, \ldots.$

The three sets **Var**, **Con**, and **ACon**, must be pairwise disjoint.

Out of these basic sets we build the "λ-terms with atoms," called **AT** (for Atomic Terms).

Definition: Let **cond** and $*$ be new symbols. Then given **ACon** and **Con** we define the set **AT**(**ACon**, **Con**) as as $\Lambda(\mathbf{Con} \cup \mathbf{ACon} \cup \{\mathbf{cond}, *\})$. When there is no confusion, we will write simply **AT**.

We define an equational calculus over **AT** by specifying axioms and rules of proof.

Definition: (Substitution) Free and bound variables are defined inductively, in the usual way. The expression $[v/x]u$, where $u, v \in \mathbf{AT}$, $x \in \mathbf{Var}$ denotes the result of substituting v for all free occurrences of x in u, with the usual proviso about renaming bound variables to avoid capture, i.e. before we substitute v for x in u, we change all the bound variables in u to be different from the free variables in v and then we replace every free occurrence of x with v.

Definition: Two terms u and v are *α-equivalent*, if v results from u by renaming the bound variables in some subterm of u (avoiding capture). Following Barendregt [BAR80], we consider two terms that are α-equivalent equal on a syntactic level, that is, terms are considered modulo α-equivalence. For example, $\lambda x.yx$ and $\lambda z.yz$ are the *same* term.

Here are the axiom schemes:

(β)	$(\lambda x.u)v = [v/x]u$	
(E)	$uv = *$,	for $u \in \mathbf{ACon} \bigcup \{*\}$.
(C1)	$\mathbf{cond}\, aavw = v$,	for $a \in \mathbf{ACon}$.
(C2)	$\mathbf{cond}\, a_1a_2vw = w$,	if $a_1, a_2 \in \mathbf{ACon}$, a_1 and a_2 different.
(C3)	$\mathbf{cond}\, u_1u_2vw = *$,	if either u_1 or u_2 is $*$.
(C4)	$\mathbf{cond}\, u_1u_2vw = *$,	if either u_1 or u_2 is of the form $\lambda x.u'$.

And here are the rules:

(trans & sym) $$\frac{\begin{matrix} u = v \\ u = v' \end{matrix}}{v = v'}$$

(cong) $$\frac{\begin{matrix} u = u' \\ v = v' \end{matrix}}{(uv) = (u'v')}$$

(ξ) $$\frac{u = v}{\lambda x.u = \lambda x.v}$$

This proof systems requires a bit of discussion. The rules are just the usual rules taken from the classical λ-calculus. Since we are committing to axiom scheme (β), it follows that the language has a call-by-name parameter passing mechanism (as does classical λ-calculus). This is to be contrasted with the usual LISP evaluator, which evaluates the arguments to a function first (call-by-value). The two strategies differ on a term such as $(\lambda xy.x)uv$, where v is a term whose evaluation doesn't terminate. In the call-by-value evaluator, the evaluation of the whole term doesn't terminate, since the evaluator never gets done evaluating the arguments. But in a call-by-name evaluator, the term v is never evaluated, and the result of evaluating the term will be the result of evaluating u.

This leaves axiom schemes (E) and (C1) through (C4), which are connected with the behavior of type errors, and of cond. So what behavior do we want? This depends on our intended use of the language AT. In this treatment, we view AT as a programming language for writing programs "about" atoms. That is, when a program is given to the evaluator, there are three interesting things that might happen:

(i) The evaluation of the program might terminate, resulting in a numeral.

(ii) The evaluation of the program might lead to a run-time type error.

(iii) The evaluation of the program might not terminate.

This is not to say that a term such as $\lambda xy.x$ is not interesting, rather, that its utility lies in its ability to be included in programs that will produce numerals. If we take this view, then the job of the evaluator is: "given a term, if it is equal to a numeral, find that numeral." In particular, if a term is not equal to a numeral, we don't care what the evaluator does, however, it would be nice if the evaluator terminates on as many terms as possible. More on this, when we discuss (C4) below. In the rest of this section, we will have need to discuss the properties of the intended evaluator. Later we will formally define an evaluator with these properties. (We are faced here with an expositional difficulty. I am reminded of a remark I heard at a philosophy seminar about Kant's *Critique of Pure Reason* [KANT29], namely, that he had many interesting things to say, and he said them all first. We might have defined the evaluator before the proof system, and equality in terms of the evaluator, and then defined a proof system which captures it. In fact, neither idea, that of the proof system nor the evaluator is really prior to the other. We want the axiom schemes to allow for a reasonable evaluator, i.e. one that is effective, and on the other hand, we want axiom schemes that make it relatively easy to reason about equality.)

Now to the rest of the axioms.

The purpose of having $*$ in the language, is so we can have a notation for run-time type errors. Our hope is to define an evaluator and a notion of run-time type error, so that a term not containing $*$ will be provably equal to $*$ if and only if it causes a run-type type error when evaluated. There are two kinds of type errors that can occur, and they correspond to those axioms schemes, that viewed as reductions have the effect of producing an $*$. These are (E), (C3), and (C4). (We could have introduced two symbols $*_1$ and $*_2$ in order to distinguish between them, at the cost of complicating the axioms a little bit).

First let us see what (E) says. Actually, it is two axioms schemes combined. The first says that $av = *$ if a is a numeral. This is one way a type error is created. It corresponds to an attempt by the evaluator to apply a numeral to a term. The second part, i.e. $*u = *$ for any term u, corresponds to "leftmost" evaluation, and is needed to insure that type-errors propagate correctly. This is best illustrated by the two following examples.

Consider the term auv, where a is a numeral. Recall that this is an abbreviation for $(au)v$. This is the sort of term that will cause a run-type type error, since the first operation of the evaluator will be to try to apply a to u. Therefore our proof system should prove this term equal to $*$. By the first part of rule (E), we know that it is equal to $*v$. We need the second part to show that it is equal to $*$.

Now consider the term $(\lambda xy.x)a(bu)$, where a and b are atoms. This illustrates that a term might not cause a run-time type error even though it has a subterm which is equal to $*$. The reason is that our evaluator will use (β) to turn this into $(\lambda y.a)(bu)$, and then use (β) again to turn it into a, which is the value of the term. The evaluator never "sees" that we are applying a numeral to a term, so there is no run-time type error. Note that in a call-by-value evaluator, since the arguments would have been evaluated first, the evaluator would indeed have encountered the type error. This illustrates our choice of the term "run-time type error" since this term would have a static type error in a language such as typed λ-calculus.

Now for the axioms about cond. The first two, (C1) and (C2), are relatively uncontroversial. They correspond to our intuition that cond $u_1u_2v_1v_2$ is a notation for "if $u_1 = u_2$ then v_1 else v_2."

Axiom scheme (C4) deals with the second kind of type error in the language. The first type error can be thought of as "trying to use an atom, where a function was expected." The type error corresponding to (C4) is, in a sense the opposite. Actually, our intuition in the preceding paragraph is a bit wrong. The problem is that it is not clear that our proof system can tell for sure when two arbitrary terms are not equal. Indeed, this relation for the classical λ-calculus is Π^0_1-complete. So the intuition for cond expressed above is a bit ambitious. Here is a second try: cond $u_1u_2v_1v_2$ means "if u_1 and u_2 are equal to the same numeral then v_1, if they are equal to different numerals then v_2."

But what about when one or both of them are not equal to numerals? The behavior we intend is that if the evaluator can determine that this situation exists, then a type error occurs. This brings up the question of when the evaluator can be sure that a term is not equal to a numeral. The answer we propose is when it is a λ-abstraction, i.e. of the form $\lambda x.u$. The purpose of (C4) is to produce such type errors. Why can't λ-abstractions be equal to numerals? It is not due to semantic problems that we disallow it. Instead we disallow it for two reasons: first, it is not clear that we could get a well behaved reduction system (one with the Church-Rosser property, as defined in chapter 4), if we did allow it; second, it would go against our intuition of what is meant by a numeral. That is, a numeral is something that should not be applied to a term, while λ-abstractions can be applied to terms by means of (β). Once we have made this decision, we can structure our evaluator, so that if it tries to evaluate a λ-abstraction at top level, it stops, since it knows that the term cannot be equal to a numeral. This allows evaluation to terminate on more terms than otherwise.

Finally, the purpose of (C3) is to make sure that if the evaluator encounters a type error while evaluating one of the two terms to be compared, then the result of the whole thing is a type error. It is analogous to the $*u = *$ part of (E) above.

Note that these axioms require parallel evaluation of the terms to be compared in a cond. That is, if we have cond $u_1u_2v_1v_2$, and the evaluation of u_1 does not terminate, if the evaluation of u_2 leads to a type error, then we want the whole term to be $*$. The same is true if we reverse the roles of u_1 and u_2. Thus, we cannot evaluate either u_1 or u_2 before the other. If we simplify our evaluator to do sequential evaluation of u_1 and u_2, then the axioms might be slightly modified: we must essentially provide an axiom for each possible outcome of the result of evaluating u_1. For a sequential evaluator, (C3) and (C4) would be replaced by the following:

(C3′)	$\mathsf{cond} * uvw = *$	
(C4′)	$\mathsf{cond}\,(\lambda x.u)u_1v_1v_2 = *$	
(C3″)	$\mathsf{cond}\,a * uv = *$	if a is a numeral
(C4″)	$\mathsf{cond}\,a(\lambda x.u)v_1v_2 = *$	if a is a numeral

So let us summarize what $\mathsf{cond}\,u_1u_2v_1v_2$ means: "Evaluate u_1 and u_2 in parallel. If they evaluate to equal numerals, then v_1. If they evaluate to unequal numerals then v_2. If either one of them evaluates to a λ-abstractions, then this is a run-time type error. If the evaluation of either one of them causes a run-time type error then we preserve that run-time type error." Notice that we leave unspecified what happens if the evaluation of both u_1 and u_2 result in terms that are neither numerals nor λ-abstractions, and do not cause type errors. Different models will do different things in this case.

Definition: Let T_{AX} be all instances of all the above axiom schemes except (β). Let T be a set of equations between terms, and let u and v be terms. Let $T \vdash u = v$ be the proof relation in classical λ-calculus, i.e. $u = v$ follows from T using just (β) and the rules. Then we say T *proves* $u = v$ if $T \bigcup T_{AX} \vdash u = v$. A set of equations T, between terms of AT is a *theory* if the set $T \bigcup T_{AX}$ is a classical λ-theory (i.e. contains all instances of (β) and is closed under application of the rules). A set of equations T between terms of AT is *inconsistent* if for every equation $u = v$, $T \bigcup T_{AX} \vdash u = v$ (which is to say that $T \bigcup T_{AX}$ is inconsistent in classical λ-calculus). Otherwise, T is consistent.

Note that a necessary condition for T to be consistent, is that for all $a_1, a_2 \in$ ACon, when a_1 and a_2 are different symbols, we do not have that T proves $a_1 = a_2$. For if so, then if $u = v$ is an arbitrary equation, we can show that T proves $u = v$. First, by (C2) we have T proves $\mathsf{cond}\,a_1a_2uv = v$. Next by repeated applications of (cong) we can show that

$$T \text{ proves } \mathbf{cond}\, a_1 a_2 uv = \mathbf{cond}\, a_1 a_1 uv\,.$$

But by (C1) we have T proves $\mathbf{cond}\, a_1 a_1 uv = u$, hence by repeated applications of (trans & sym) the result follows.

3. Semantics

We now define what a model for this language is, along with a denotational semantics [STOY77]. The model is a combinatory model as in [MEYER82], with extra structure added to take care of the behavior of atoms. Combinatory models are models of classical λ-calculus. Our semantics is also taken from the usual semantics of λ-calculus. This approach is somewhat similar to defining a group as a first order structure satisfying some nonlogical axioms. Completeness of these axioms with respect to groups then follows from completeness of first order logic. In our case, the classical λ-calculus and combinatory models are in the same relation to each other as logic would be to a first-order definable structure. The axiom schemes (E), (C1), (C2), (C3), and (C4) correspond to the group axioms.

First we recall the definition of combinatory model from [MEYER82]. These serve as models for the classical λ-calculus.

Definition: A *combinatory model* $\mathcal{D}$ is a tuple $(D, \cdot, \epsilon)$ where $\cdot$ is a binary operation on D, and there exists $K, S \in D$ such that

(CM.1) For all $d_1, d_2 \in D$, $(K \cdot d_1) \cdot d_2 = d_1$.

(CM.2) For all $d_1, d_2, d_3 \in D$, $((S \cdot d_1) \cdot d_2) \cdot d_3 = (d_1 \cdot d_3) \cdot (d_2 \cdot d_3)$

(CM.3) For all $d_1, d_2 \in D$, $(\epsilon \cdot d_1) \cdot d_2 = d_1 \cdot d_2$.

(CM.4) If for all $d \in D$, $d_1 \cdot d = d_2 \cdot d$ then $\epsilon \cdot d_1 = \epsilon \cdot d_2$.

In what follows, we write $d_1 d_2$ for $d_1 \cdot d_2$ and $d_1 d_2 \cdots d_n$ for $(\cdots(d_1 \cdot d_2) \cdot \cdots \cdot d_n)$.

Given a combinatory model $\mathcal{D} = (D, \cdot, \epsilon)$, let ι be an interpretation of constants, i.e. a map from C to D. Let $\mathbf{Env} = \mathbf{Var} \to D$. For $\rho \in \mathbf{Env}$, $x \in \mathbf{Var}$, and $d \in D$ let $\rho\{d/x\} \in \mathbf{Env}$ be that function such that

$\rho\{d/x\}(x) = d$, and

$\rho\{d/x\}(y) = \rho(y)$, for $y \neq x$.

The function $\mathcal{E}_{\mathcal{D},\iota} : \Lambda(C) \rightarrow \mathbf{Env} \rightarrow D$ is the semantic function for λ-terms in a combinatory model from [MEYER82]. As a notational convenience we write $\mathcal{E}_{\mathcal{D},\iota}[\![u]\!]\rho$ as simply $[\![u]\!]\rho$, when no confusion results.

Definition: The denotational semantics for λ-terms.

(DS.1) $[\![c]\!]\rho = \iota(c)$, for $c \in C$.

(DS.2) $[\![x]\!]\rho = \rho(x)$, for $x \in \mathbf{Var}$.

(DS.3) $[\![uv]\!]\rho = ([\![u]\!]\rho)([\![v]\!]\rho)$.

(DS.4) $[\![\lambda x.u]\!]\rho = \epsilon\delta$, where $\delta \in D$ is such that for all $d \in D$, $\delta d = [\![u]\!]\rho\{d/x\}$. (By definition of ϵ in a combinatory model, $\epsilon\delta$ is independent of the choice of such a δ. Furthermore, it shown in [MEYER82] that such a δ must exist if $\mathcal{D}$ is a combinatory model.)

To serve as models for AT we allow only certain types of combinatory models and certain types of constant mappings, ι:

Definition: An *atomic combinatory model* (acm) $\mathcal{A}$ is a tuple: $(D, \cdot, \epsilon, D^A, *^D, \gamma)$, where $*^D, \gamma \in D$ and:

(ACM.1) $(D, \cdot, \epsilon)$ is a combinatory model.

(ACM.2) $D^A \subseteq D$ is a set whose elements are called atoms.

(ACM.3) For all $d \in D$, all $a \in D^A \bigcup \{*^D\}$, $a \cdot d = *^D$.

(ACM.4) For all $a \in D^A$, all $d_1, d_2 \in D$, $\gamma a a d_1 d_2 = d_1$.

(ACM.5) For all $a_1, a_2 \in D^A$, $a_1 \neq a_2$, all $d_1, d_2 \in D$, $\gamma a_1 a_2 d_1 d_2 = d_2$.

(ACM.6) For all $d_1, d_2, d_3 \in D$, $\gamma *^D d_1 d_2 d_3 = \gamma d_1 *^D d_2 d = *^D$.

(ACM.7) For all $d_1, d_2, d_3, d_4 \in D$, $\gamma(\epsilon d_1) d_2 d_3 d_4 = \gamma d_1 (\epsilon d_2) d_3 d_4 = *^D$.

The subset D^A of D will serve as values for the atomic constants, that is, they are the atoms of D. An acm is simply a combinatory model that satisfies the axiom schemes (E) and (C1) through (C4), if $*^D = [\![*]\!]\rho$ and $\gamma = [\![\mathbf{cond}]\!]\rho$, for all ρ. That this happens is guaranteed by our choice of constant mapping functions ι:

Definition: Let $\mathcal{A} = (D, \cdot, \epsilon, D^A, *^D, \gamma)$ be an acm. A function

$$\iota : \mathbf{Con} \bigcup \mathbf{ACon} \bigcup \{\mathbf{cond}, *\} \rightarrow D$$

is called an *interpretation* if

(I.1) $\iota(\text{cond}) = \gamma$.

(I.2) $\iota(*) = *^{\prime\prime}$.

(I.3) $\iota(a) \in D^A$, for every $a \in \Lambda\text{Con}$.

(I.4) $\iota(a_1) \neq \iota(a_2)$ if a_1 and a_2 are different.

Definition: Let $\mathcal{D}$ be a combinatory model and $\iota : C \rightarrow D$, a constant mapping. Let $u, v \in \Lambda(C)$. Recall that $\models_{\mathcal{D},\iota} u = v$ if for all $\rho \in \text{Env}$, $[\![u]\!]\rho = [\![v]\!]\rho$. If T is a set of equations between terms of $\Lambda(C)$, we write $\models_{\mathcal{D},\iota} T$ if $\models_{\mathcal{D},\iota} t$ for all $t \in T$. If T is a set of equations between terms of $\Lambda(C)$, write $T \models u = v$ if for all $\mathcal{D}$ and ι, whenever $\models_{\mathcal{D},\iota} T$ then $\models_{\mathcal{D},\iota} u = v$. If T is a set of equations between terms of AT, and $u, v \in \text{AT}$ then we say T *semantically implies* $u = v$ if $T \bigcup T_{AX} \models u = v$.

Definition: Let $\mathcal{D}$ be a combinatory model, and $\iota : C \rightarrow D$ a constant mapping. Then define

$$\text{Th}(\mathcal{D}, \iota) = \{u = v : u, v \in \Lambda(C), [\![u]\!]\rho = [\![v]\!]\rho, \text{ for all } \rho\}.$$

The two theorems below are from Meyer [MEYER82].

Theorem: (Soundness Theorem for $\Lambda(C)$) If $T \vdash u = v$ then $T \models u = v$. (From which it follows that for any combinatory model $\mathcal{D}$ and any constant mapping function ι, $\text{Th}(\mathcal{D}, \iota)$ is a λ-theory.)

Theorem: (Completeness Theorem for $\Lambda(C)$) For any λ-theory $\mathcal{T}$, there is a combinatory model $\mathcal{D}$ and a constant mapping function ι, such that $\mathcal{T} = \text{Th}(\mathcal{D}, \iota)$. (From which it follows that for any set of equations T, if $T \models u = v$ then $T \vdash u = v$.)

That our proof system is complete now follows directly from Meyer's results, just as in group theory we know that the axioms for groups are complete for the class of group by virtue of the fact that first order logic is complete. The axioms T_{AX} correspond to the axioms for groups.

Theorem 3.1: (Soundness Theorem for AT) If T proves $u = v$ then T semantically implies $u = v$. (From which it follows that for any acm $\mathcal{A}$ and any interpretation ι, Th$(\mathcal{A}, \iota)$ is a theory.)

Theorem 3.2: (Completeness Theorem for AT) For any consistent theory $\mathcal{T}$, there is an acm $\mathcal{A}$ and an interpretation ι, such that $\mathcal{T} =$ Th$(\mathcal{A}, \iota)$. (From which it follows that for any set of equations T, if T semantically implies $u = v$ then T proves $u = v$.)

4. Reduction

In the two preceding sections, we have presented a proof system and a notion of model, and shown that the proof system is complete for that notion of model. We now turn to reduction, which comes closer to the computational aspect of terms.

What are the terms to be used for? We want to use the terms to write programs. In this section, we explore an interpreter for those programs. All that the interpreter cares about a term, is whether it is provably equal to a numeral. If so, its job is to find that numeral.

With this is mind we introduce a notion of reduction. First, we define the notion of a *context*.

Definition: A *context* is a term of AT with a "hole" in its parse tree. Formally, let Θ be a new symbol. Then, a context, $C[\,]$ is a term of

$$\Lambda(\mathbf{ACon} \bigcup \mathbf{Con} \bigcup \{\mathbf{cond}, *, \Theta\}).$$

If u is a term of AT, and $C[\,]$ a context, then $C[u]$ denotes the result of replacing *without* renaming bound variables, every occurrence of the symbol Θ in $C[\,]$ with u. For example, if $C[\,] = \lambda x.\Theta$, then $C[x] = \lambda x.x$. This is in contrast to substitution: $[x/\Theta]\lambda x.\Theta = \lambda x'.x$, where x' is a fresh variable different from x.

Definition: A notion of reduction R is a binary relation between terms of AT. Given R, define the relation $\rightarrow_R$ as

$$\{(C[u], C[v]) : C[\,] \text{ is a context and } (u,v) \in R\}\,.$$

The relation is written in infix notation. If $u \to_R v$ we say u *reduces in one step* to v. The relation $\to_R^*$ is the reflexive, transitive closure of $\to_R$. If $u \to_R^* v$ then we say that u *reduces to* v, or v is *a reduction* of u.

Lemma 4.1: Let $C[\,]$ be a context. If $u \to_R v$ then $C[u] \to_R C[v]$. Also if $u \to_R^* v$ then $C[u] \to_R^* C[v]$.

Proof: If $u \to_R v$ then there exists $(u', v') \in R$ and a context $C'[\,]$, such that $u = C'[u']$ and $v = C'[v']$. But then $C[u] = C[C'[u']]$ and $C[v] = C[C'[v']]$. But then as $C[C'[\,]]$ is also a context and by definition of $\to_R$, we have $C[u] \to_R C[v]$. The other statement follows by induction on the number of steps it takes to reach v from u. ∎

When u is reduced to v we can think of this as a computation step. If the notion of reduction is reasonable, then we are never lead down any "blind alleys," that is, if a term is reduced in two different ways to yield two different terms, then it is possible to reduce each of these terms to the same term. This is the definition of the Church-Rosser property, as defined in Barendregt [BAR80].

Definition: A notion of reduction R is *Church-Rosser* if whenever a term u reduces to both v_1 and v_2, then there exists a term u' that is a reduction of both v_1 and v_2.

We will choose our notion of reduction so that it captures the proof system presented above (in a way that will be made precise) for a given set of equations T, and is Church Rosser. A set of equations T is called *simple* if they are of the form:

(i) $c_1c_2 = c_3$, where $c_i \in \text{ACon} \cup \text{Con}$ and $c_1 \notin \text{ACon}$, or

(ii) $c_1* = *$, where $c_1 \in \text{Con}$.

We also require that for every equation $c_1c_2 = c_3$ in T, the equation $c_1* = *$ is also in T. If $c_1c_2 = c_3 \in T$, we say that c_1 is an *active* constant, since then the reduction system has rules for applying it to arguments. A set of equations of this form, can be thought of as specifying the behavior of builtin functions on the numerals and on each other. Requirement (ii) says that builtin functions cannot ignore type errors, i.e. if we get a type error while evaluating the argument to a builtin function, then the whole term is equal to $*$.

Definition: Our notion of reduction R is

$$R_\beta \bigcup R_E \bigcup R_{C1} \bigcup R_{C2} \bigcup R_{C3} \bigcup R_{C4} \bigcup R_T ,$$

where

$$R_\beta = \{(u,v) : u = v \text{ is an instance of axiom scheme } \beta\} ,$$

similarly for all the other axiom schemes and

$$R_T = \{(u,v) : u = v \in T\} .$$

We will abbreviate $\rightarrow_{R_\beta}$ as $\rightarrow_\beta$ and similarly for the other notions of reduction.

We are working toward the following result:

Theorem 4.2: (Church-Rosser Theorem for R) The notion of reduction R defined above is Church-Rosser.

The following definition and two results are taken from Barendregt [BAR80].

Theorem 4.3: The notion of reduction R_β is Church-Rosser.

Definition: Let R_1 and R_2 be two notions of reduction. We say R_1 *commutes* with R_2 if whenever there exist terms u, v_1, and v_2 such that $u \rightarrow^*_{R_1} v_1$ and $u \rightarrow^*_{R_2} v_2$, then there is a term u' such that $v_1 \rightarrow^*_{R_2} u'$ and $v_2 \rightarrow^*_{R_1} u'$.

Lemma 4.4: (Lemma of Hindley-Rosen): If R_1 and R_2 are two Church-Rosser notions of reduction, and R_1 commutes with R_2, then the notion of reduction $R_1 \bigcup R_2$ is Church-Rosser.

The Lemma of Hindley-Rosen can be generalized to work for any number of notions of reduction:

Lemma 4.5: If R commutes with R_i for $1 \leq i \leq n$, then R commutes with $\bigcup_{i=1}^n R_i$.

Proof: We abbreviate $\bigcup_{i=1}^{n} R_i$ by $\bigcup R_i$. We must show that if u R-reduces to v_1 and u $(\bigcup R_i)$-reduces to v_2, then there exists u' which is an R-reduction of v_2 and a $(\bigcup R_i)$-reduction of v_1. The proof is by induction on the number of steps it takes to reduce u to v_2.

The base case is when it takes 0 steps, i.e. $v_2 = u$. Then the desired u' is just v_1: by assumption it is an R-reduction of v_2, and since it is equal to v_1 it is certainly a $(\bigcup R_i)$-reduction of v_1.

Suppose now that the lemma is true when u reduces to v_2 in k steps, we prove it for $k+1$. Then we must have a term v_2' that is a $(\bigcup R_i)$-reduction of u in k steps, and without loss of generality we can assume that v_2' R_1-reduces in one step to v_2 (otherwise interchange the names of the R_i). Then by induction there is a term u'' that is an R-reduction of v_2' and is a $(\bigcup R_i)$-reduction of v_1. But then as R commutes with R_1, there is a term u' which is an R-reduction of v_2 and an R_1-reduction of u''. But since u'' is a $(\bigcup R_i)$-reduction of v_1 and u' is an R_1-reduction of u'', we have that u' is a $(\bigcup R_i)$-reduction of v_1, so it is the desired u'. ∎

Lemma 4.6: Let $R_1, \ldots, R_n$ be a sequence of Church-Rosser notions of reduction, where R_i commutes with R_j for $1 \leq i < j \leq n$. Then the notion of reduction $R_1 \cup \cdots \cup R_n$ is Church-Rosser.

Proof: Induction on n. For $n = 2$ this is the Lemma of Hindley-Rosen. Suppose the lemma is true for $n < k$. Consider now $n = k$. Then by the induction hypothesis, $R_1 \cup \cdots \cup R_{k-1}$ is Church-Rosser. However by the previous lemma R_k commutes with $R_1 \cup \cdots \cup R_{k-1}$. Hence by Hindley-Rosen, $(R_1 \cup \cdots \cup R_{k-1}) \cup R_k$ is Church-Rosser, which completes the proof of the lemma. ∎

Definition: A reduction relation has the *diamond property*, if whenever u reduces in one step to both v_1 and v_2, there is a term u' which is reducible in at most one step from both v_1 and v_2.

The next Lemma is from Barendregt [BAR80].

Lemma 4.7: If R has the diamond property then R is Church-Rosser.

Definition: Let R be a notion of reduction. If $(u, v) \in R$ then the term u is called a *redex* and the term v is called its *reduct*. When we refer to a redex r of a term u, we are referring to a particular occurrence of a redex r as a subterm of u.

Let us consider now

$$R_E \bigcup R_{C1} \bigcup R_{C2} \bigcup R_{C3} \bigcup R_{C4} \bigcup R_T \, .$$

This notion of reduction has the following reduction properties:

1. A reduct is either a constant, or a subterm of the redex.
2. If u is a redex of $C[u]$, with reduct v, and $C[u]$ is a redex with reduct w, then:
   2.1. If w does not contain u, then $C[v]$ is also redex whose reduct is w.
   2.2. If w does contain u, i.e. $w = C'[u]$, then $C[v]$ is a redex whose reduct is $C'[v]$.

This is enough to show that the above notion of reduction has the diamond property, i.e.:

Lemma 4.8: $R_E \cup R_{C1} \cup R_{C2} \cup R_{C3} \cup R_{C4} \cup R_T$ is Church-Rosser.

Proof: We show that it has the diamond property. Suppose a term u has two redexes, r_1 and r_2. Then there are two cases to consider:

1. The redexes r_1 and r_2 are disjoint. In this case the redexes can be reduced in either order, yielding the same term.
2. One redex occurs inside another. Without loss of generality, assume that r_2 occurs inside r_1. Then there are two subcases:
   2.1. The reduct of r_1 does not contain r_2. Then by the above reduction properties, if we first reduce r_2 and then reduce the resulting term, we get the same term as if we simply reduced r_1.
   2.2. The reduct of r_1 contains r_2. Then r_1 is $C[r_2]$, and the reduct of r_1 is $C'[r_2]$. Suppose the reduct of r_2 is r. Then if we first reduce r_1 we get $C'[r_2]$. If we first reduce r_2 we get $C[r]$. But we can reduce $C'[r_2]$ to get $C'[r]$ and by the above reduction propeties, we can reduce $C[r]$ to $C'[r]$.

Since this notion of reduction has the diamond property it is Church-Rosser. ∎

At this point we know that R_β is Church-Rosser, and the rest of the notions of reduction, taken together, are Church-Rosser. We will show that all the notions of reduction, taken together, are Church-Rosser, using the lemma of Hindly-Rosen. So we must establish that R_β commutes with the rest of the reduction notions. By Lemma 4.5, it suffices to show that R_β commutes with all the other notions of reduction.

Just as we used the diamond property to show that a notion of reduction is Church-Rosser, we define a property of two notions of reduction that will insure that they commute. The definition and the following lemma are taken from Barendregt [BAR80].

Definition: Two notions of reduction, R_1 and R_2, have the ***cross diamond property***, if whenever there are terms u, v_1, and v_2, such that u R_1-reduces in one step to v_1 and R_2-reduces in one step to v_2, then there is a term u' that is R_1-reducible from v_2 in at most one step, and is R_2-reducible from v_1 (in any number of steps).

Lemma 4.9: If two notions of reduction have the cross diamond property then they commute.

Now we can show that R_β commutes with all the other notions of reduction by showing that R_β and each of the other notions enjoy the cross diamond property. Unfortunately, to show this is rather tedious, it being a case by case analysis of how redexes can overlap. Therefore, we will show one case, the rest are similar.

Lemma 4.10: R_β and R_E commute.

Proof: We show that they have the cross diamond property. There are two cases.

1. A β-redex occurs inside an E-redex. Then the E-redex is of the form $uC[(\lambda x.v)w]$, where $u \in \mathrm{ACon} \bigcup \{*\}$. If we do the E-reduction first we get $*$. If we do the β-reduction first, we get $uC[([v/x]u)]$, which is an E-redex with reduct $*$.
2. An E-redex occurs inside a β-redex. Then there are two subcases:
   2.1. The β-redex is of the form $(\lambda x.w)C[uv]$, where $u \in \mathrm{ACon} \bigcup \{*\}$. Then if we first do the E-reduction we get $(\lambda x.w)C[*]$, and we can then do a β-reduction to get $[C[*]/x]w$. On the other hand if we first do the β-

reduction, we get $[C[uv]/x]w$, and we can then do a series of E-reductions, one for every free x in w, to ultimately yield $[C[*]/x]w$.

2.2. The β-redex is of the form $(\lambda x.C[uv])w$, where $u \in \mathrm{ACon} \bigcup \{*\}$. Then if we E-reduce first we get $(\lambda x.C[*])w$, and we can then β-reduce to get $[w/x](C[*])$, which is $C'[*]$, where $C'[\,]$ is the result of renaming the bound variables in $C[\,]$ and substituting w for free occurrences of x. On the other hand if we β-reduce first we get $[w/x](C[uv])$. Now this is equal to $C'[uv']$, where v' is the result of substituting w for all occurrence of x in v that are free in $C[uv]$. But as uv' is an E-redex, we may reduce it to get $C'[*]$, as before.

This shows that R_E and R_β have the cross diamond property, and therefore commute. ∎

Theorem 4.2 now follows from Lemma 4.3, Lemma 4.8, Lemma 4.4, and Lemma 4.10 (and the other omitted cases).

Theorem 4.11: If T is a simple set of equations, then $T \vdash u = v$ if and only if there is a term w that is reducible from both u and v.

Proof: Suppose w is reducible from both u and v. Since all the notions of reductions are instance of axiom schemes or equations in T, by rules (cong) and (ξ), if u reduces to u' in one step then $T \vdash u = u'$, hence by rule (trans & sym) if u reduces to w then $T \vdash u = w$. Then if w is reducible from both u and v then $T \vdash u = w$ and $T \vdash v = w$ and then by rule (trans & sym), $T \vdash u = v$.

Conversely, suppose $T \vdash u = v$. We use induction on the length of proof. If the length is 0, then $u = v$ is either an instance of an axiom scheme, or an equation in T. In either case u reduces to v in one step, so the desired term w is just v. Otherwise, $u = v$ follows via a rule, from equations that have shorter proofs. We consider one rule at a time.

(trans & sym) Then $T \vdash r = u$ and $T \vdash r = v$ for some term r. By induction, then, there are terms w_1 reducible from r and u and w_2 reducible from r and v. But since w_1 and w_2 are both reducible from r, by the Church-Rosser property there is a term w reducible from both w_1 and w_2. But this term is then reducible from both u and v.

(ξ) Then u is of the form $\lambda x.u'$ and v is of the form $\lambda x.v'$, where $T \vdash u' = v'$. By induction, then, there is a term w', which is reducible from both u' and v'. But then the term $w = \lambda x.w'$ is reducible from both u and v, by Lemma 4.1, using context $\lambda x.\Theta$.

(cong) Then u is of the form u_1u_2 and v is of the form v_1v_2, where $T \vdash u_1 = v_1$ and $T \vdash u_2 = v_2$. Then there exists terms w_1 and w_2 such that w_i is reducible from both u_i and v_i. Then from Lemma 4.1, using context $u_1\Theta$, we get that u_1w_2 is reducible from u_1u_2. Again using Lemma 4.1, with context Θw_2, we get that w_1w_2 is reducible from u_1w_2. Hence w_1w_2 is reducible from u_1u_2. Similarly, we can show that w_1w_2 is reducible from v_1v_2, so $w = w_1w_2$ is the desired term. ∎

5. Evaluation

If, as remarked above, we view reduction of a term as a computational step, the results of the preceding chapter tell us how to build a naive evaluator for our language. Namely, start with a term and try all possible reduction sequences. If we arrive at a term that can no longer be reduced, then stop. The Church-Rosser theorem guarantees that this term will be unique.

However, this evaluator is a bit unsatisfying. First of all, since we must remember the state of several reduction sequences at once, its demands on memory are great. Secondly, it will be slow, since it is doing breadth-first search of a tree, without using any heuristics to narrow down to the goal. And lastly, it gives us no insight into what a run-time type error is, since it might do several E-reductions, and ultimately arrive at a term which is not $*$.

All that we require of an evaluator is that if a term is provably equal to a numeral from T (by Church-Rosser theorem, it must therefore reduce to that numeral) then the evaluator will find that numeral. We don't care what the evaluator does with a term that is not equal to a numeral, just so long as it doesn't return a numeral. That is all that we require. However, there are certain things that we desire. One is that the evaluator terminate on as many terms as possible. Second is a notion of type error that coincides with the use of * in the axioms (of course, this is the chicken and egg phenomenon).

As was remarked in Chapter 2, the parallel nature of **cond** will complicate things, since the evaluator cannot simply evaluate one arm of the **cond** before the other. In fact, if we were using the sequential axioms, (C3′), (C4′), (C3″), and (C4″), then the evaluator which always reduces the leftmost redex would be *normalizing*, i.e. if a term u was equal to a term v which had no redexes, then this evaluator would reduce u to v.

Unfortunately, life is not so simple, and we cannot get away with such a simple evaluator. Due to the parallel nature of cond, we are forced to consider a parallel evaluator, that is, an evaluator which at every step reduces a set of disjoint redexes (since the redexes are disjoint, the order in which they are reduced does not matter, indeed, they may reduced at the same time, which is why the evaluator is called parallel). Parallel evaluators were considered by [LÉVY80].

Definition: A term of the form $\mathbf{cond}\, u_1u_2u_3u_4$ is called a cond-expression.

We now describe the evaluator EVAL : AT → AT. If u is a term of AT, EVAL(u) is a term which is reducible from u. If EVAL(u) = u then the evaluator is said to *halt* on u. The evaluator is repeatedly applied until a term is reached where it halts. This process is called EVAL-uation.

Definition: The evaluator EVAL:

1. If u is a redex, then EVAL(u) is its reduct.
2. If u is a cond-expression $\mathbf{cond}\, u_1u_2v_1v_2$ then

$$\mathrm{EVAL}(u) = \mathbf{cond}\, \mathrm{EVAL}(u_1)\mathrm{EVAL}(u_2)v_1v_2\,.$$

3. If u is $\lambda x.v$ then $\mathrm{EVAL}(u) = \lambda x.\mathrm{EVAL}(v)$.
4. If u is u_1u_2, where $u_1 \in \mathrm{Var} \bigcup \mathrm{Con}$ then $\mathrm{EVAL}(u) = u_1\mathrm{EVAL}(u_2)$.
5. If $u = (u_1u_2)u_2$ then $\mathrm{EVAL}(u) = \mathrm{EVAL}(u_1u_2)u_3$.
6. Otherwise $\mathrm{EVAL}(u) = u$.

In English, EVAL works as follows: it looks for the leftmost redex or cond-expression; if it is a redex, it reduces it, if it is a cond-expression, it calls itself recursively on the two "arms" of the cond-expression.

Normalization Claim: EVAL is a normalizing evaluator. That is, if $u = v$ is provable from T and v has no redexes (is in normal form), then the EVAL-uation of u yields v, and if $\text{EVAL}(u) = u$ then u is in normal form.

It is hoped that this can be shown using some notion of standard reduction, in the same way that the Standardization Theorem is proved for classical λ-calculus [BAR80]. At present, there was not time to prove this claim.

Even though EVAL is normalizing, it is still not the evaluator we want for AT. Recall that all we required of an evaluator is that if a term was equal to a numeral, it found that numeral. Since numerals are normal forms, and EVAL is normalizing, it accomplishes that goal. But it will not terminate on lots of term which we can be sure are not numerals, for example

$$u = \lambda x.(\lambda y.yy)(\lambda y.yy)$$

has no normal form, so the EVAL-uation of u will never stop, yet since u is a λ-abstraction, it can never be a numeral. To fix this problem, we modify EVAL so that it never looks inside a λ-abstraction.

Definition: In a λ-abstraction $\lambda x.u$ the term u is said to be the *scope* of the λ.

Definition: The evaluator EVAL' is defined as follows:

1. If u is a redex, then $\text{EVAL}'(u)$ is its reduct.
2. If u is a cond-expression $\text{cond}\, u_1u_2v_1v_2$ then

$$\text{EVAL}'(u) = \text{cond}\,\text{EVAL}'(u_1)\text{EVAL}'(u_2)v_1v_2\,.$$

3. If u is u_1u_2, where u_1 is an active constant then $\text{EVAL}'(u) = u_1\text{EVAL}'(u_2)$.
4. If $u = (u_1u_2)u_2$ then $\text{EVAL}'(u) = \text{EVAL}'(u_1u_2)u_3$.
5. Otherwise $\text{EVAL}'(u) = u$.

The difference between EVAL and EVAL$'$ is that EVAL$'$ does not reduce inside λ-abstractions and it only evaluates arguments of live constants, since otherwise it knows that it has no rules for reducing the application. Although EVAL$'$ is now no longer normalizing (since it halts on $\lambda x.(\lambda y.y)x$) it still has all that we required of an evaluator:

Theorem 5.1: If $u = v$ is provable from T and v is a numeral, then the EVAL$'$-uation of u yields v.

Proof: We know by the Normalization Claim that the EVAL-uation of u yields v. Now if clause 3. is used in the EVAL-uation, then on the next pass it must be used again, since no new redexes or cond-expressions will be created outside the λ. So a numeral cannot result. Similarly, clause 4. will never be used when u_1 is not a live constant, since that would result in clause 4. being used again on the next pass, as nothing new will be created to trigger clauses 1., 2., or 3. Hence the EVAL-uation of u is also an EVAL$'$-uation and hence the EVAL$'$-uation u yields v. ∎

We now can explain what a run-time type error is, in terms of the evaluator EVAL$'$. We say that EVAL$'$ encounters a run-time type error on term u, if in the EVAL$'$-uation of u, rule 1. is applied to an (E)-redex or to a (C4)-redex.

Theorem 5.2: Let u be a term which does not contain $*$. Then EVAL$'$ encounters a run-time type error on term u, if and only if $u = *$ is provable from T.

Proof: Certainly if $u = *$ then by Church-Rosser it is possible to reduce u to $*$. However, since (E) and (C4) are the only reduction rules which *create* an $*$, one of these must be used. Also, by the same reasoning as Theorem 5.1, the evaluation of u will result in $*$. Hence, one of the above redexes must be contracted.

For the converse, it suffices to show that if EVAL$'(u)$ results in a type error then $u = *$. We argue by cases, on what clause is used to handle EVAL$'(u)$.

1. If a type-error results then the redex is either an (E)-redex or a (C4) redex. Then the reduct is $*$ so $u = *$.
2. Then $u = \mathbf{cond}\, u_1 u_2 v_1 v_2$ and either EVAL$'(u_1)$ or EVAL$'(u_2)$ results in a type error. By induction, then either u_1 or u_2 is equal to $*$. Hence $u = *$, by (C3).

3. Then $u = u_1 u_2$ and $\text{EVAL}'(u_2)$ results in a type error. Then by induction $u_2 = *$. So by the restrictions on T, $u_1 * = * \in T$, so $u = *$.
4. Then $u = (u_1 u_2) u_3$ and $\text{EVAL}'(u_1 u_2)$ results in a type error. By induction, then, $u_1 u_2 = *$ so by axiom scheme (E), $u = *$.
5. This cannot cause a type error.

6. Expressive Power

In this chapter, we study the expressive power of a particular language of the type we have been discussing. In particular, we fix the constants and the set of equations T, and ask what functions we can represent. Let the language **LAM** be the language defined in chapter 2, with the following choice of constants:

ACon $= \{\underline{n} : n = 0, 1, 2 \ldots\}$.

Con $= \{\underline{\text{Succ}}\}$.

Let the language **LAM**$_0$ be the language **LAM**, without **cond**.

For both **LAM** and **LAM**$_0$, the set of equations T will be

$$\{\underline{\text{Succ}}\, \underline{n} = \underline{n+1} : n = 0, 1, 2, \ldots\}.$$

Definition: Let f be an n-ary partial function over the natural numbers. We say that f is *numeral represented* by a term u, if

$$\text{whenever } f(i_1, \ldots, i_n) = j \text{ then } T \vdash u \underline{i_1} \cdots \underline{i_n} = \underline{j}\,,$$

and

$$\text{whenever } f(i_1, \ldots, i_n) \text{ is undefined then } T \not\vdash u \underline{i_1} \cdots \underline{i_n} = \underline{j}, \text{ for any } j.$$

Definition: The *Church numeral* $\underline{\underline{n}}$ is defined as follows:

$\underline{\underline{0}} = \lambda f\, \lambda x . x$

$\underline{\underline{n}} = \lambda f\, \lambda x . f^{(n)} x$, for $n > 0$.

We also define what it means for a term to *Church-represent* a partial function: simply replace $\underline{i}$ by $\underline{\underline{i}}$ in the above definition.

Theorem 6.1: The Church-representable partial functions are exactly the partial recursive functions.

Proof: See Barendregt [BAR80].

We show that we can translate between $\underline{n}$ and $\underset{\sim}{n}$ using terms, and therefore:

Theorem 6.2: The numeral representable partial functions are exactly the partial recursive functions.

This follows after a few lemmas.

Lemma 6.3: ([BAR80]) There is a term $\underline{\mathrm{Succ}}$ such that for all n, $\underline{\mathrm{Succ}}\,\underline{n} = \underline{\underline{n+1}}$.

Proof: An immediate corollary of Theorem 6.1. In fact the term

$$\lambda y\,\lambda f\,\lambda x.f(yfx)\,,$$

will serve as $\underline{\mathrm{Succ}}$ as is easily shown by induction. ∎

Lemma 6.4: ([BAR80]) There is a term Y (Curry's Paradoxical Combinator) such that for all u, $Yu = u(Yu)$.

Proof: $Y = \lambda f(\lambda x.f(xx))(\lambda x.f(xx))$, since

$$Yu = (\lambda x.u(xx))(\lambda x.u(xx)) = u((\lambda x.u(xx))(\lambda x.u(xx))) = u(Yu)\,. \quad ∎$$

Lemma 6.5: There is a term $\underline{\mathrm{Pred}}$ such that for all $m > n \geq 0$,

$$T \vdash \underline{\mathrm{Pred}}\,\underline{m}\,\underline{n} = \underline{m-1}\,.$$

Proof: We can write a recursive definition for $\underline{\mathrm{Pred}}$ as follows:

$$\underline{\mathrm{Pred}}\,xy = \mathrm{cond}\,x(\underline{\mathrm{Succ}}\,y)y(\underline{\mathrm{Pred}}\,x(\underline{\mathrm{Succ}}\,y))\,.$$

In "programming" terms, we check if x is the successor of y, if it is we return y, if not we increment y and try again. The program must halt if $x > y$. Writing the above equation another way, we get

$$\underline{\mathrm{Pred}} = (\lambda f\,\lambda x\,\lambda y.\mathrm{cond}\,x(\underline{\mathrm{Succ}}\,y)y(fx(\underline{\mathrm{Succ}}\,y)))\underline{\mathrm{Pred}}\,.$$

Then by the previous lemma, the term

$$\underline{\mathrm{Pred}} = Y(\lambda f\,\lambda x\,\lambda y.\mathrm{cond}\,x(\underline{\mathrm{Succ}}\,y)y(fx(\underline{\mathrm{Succ}}\,y)))$$

will behave as desired, as can be checked by induction. ∎

Lemma 6.6: There are LAM terms u and v such that for all n

$$T \vdash u\underline{n} = \underline{\underline{n}} \text{ and } T \vdash v\underline{\underline{n}} = \underline{n}\,.$$

Proof: The term v is simply $\lambda x.x\underline{\mathrm{Succ}}\,\underline{0}$, since then

$$v\underline{\underline{n}} = \underline{\underline{n}}\,\underline{\mathrm{Succ}}\,\underline{0} = \underline{\mathrm{Succ}}^{(n)}\underline{0} = \underline{n}\,.$$

The term u is more complicated. Again, we write a recursive definition:

$$ux = \mathbf{cond}\, x\underline{0}(\underline{\underline{0}})(\underline{\underline{\mathrm{Succ}}}(u(\underline{\mathrm{Pred}}x)))\,,$$

or equivalently,

$$u = (\lambda f\, \lambda x.\mathbf{cond}\, x\underline{0}(\underline{\underline{0}})(\underline{\underline{\mathrm{Succ}}}(u(\underline{\mathrm{Pred}}x))))u\,,$$

so again we see that

$$u = Y(\lambda f\, \lambda x.\mathbf{cond}\, x\underline{0}(\underline{\underline{0}})(\underline{\underline{\mathrm{Succ}}}(u(\underline{\mathrm{Pred}}x))))$$

will work, as can be verified by induction. ∎

Proof of Theorem 6.2: Let f be an n-ary partial recursive function. Then by Theorem 6.1 there is a term h which Church-represents f. Let u and v be as in the preceding lemma. The the following term will represent f:

$$\lambda x_1 \cdots x_n.v(h(ux_1)\cdots(ux_n))\,.$$

By switching the roles of u and v we can show that every representable function is Church-representable. ∎

So using LAM, we can represent all the numeric functions that we can hope the represent. We explore now, what the situation is if **cond** is not allowed, that is, what functions are representable by terms of $\mathbf{LAM}_0$.

Definition: Let π_i^n be the function of n arguments whose value is the ith argument, i.e.

$$\pi_i^n(x_1, \ldots, x_n) = x_i\,.$$

For all natural numbers n, let a_n be the function of one argument that adds n to its argument, and let k_n be the function of one argument who value in n, i.e.

$$a_n(x) = x + n$$
$$k_n(x) = n \, .$$

Let ω be the unary function that is undefined everywhere.

Lemma 6.7: The functions $\omega \circ \pi_i^n$, $k_m \circ \pi_i^n$, and $a_m \circ \pi_i^n$ are representable by $\mathbf{LAM}_0$ terms, for all natural numbers m, n, and i with $1 \leq i \leq n$.

Proof: First note that if an n-ary function f is represented by a term F, and a unary function g is represented by a term G, then the n-ary function $g \circ f$ is represented by $\lambda x.G(Fx)$. Hence it suffices to show that the functions π_i^n, ω, k_n, and a_n are all representable. But π_i^n is represented by $\lambda x_1 \cdots x_n.x_i$, k_n is represented by $\lambda x.\underline{n}$, a_n is represented by $\lambda x.\underline{\mathrm{Succ}}^{(n)}x$, and ω is represented by $((\lambda x.xx)(\lambda x.xx))$.

We will show that these simple functions are *all* the functions that can be represented by terms of $\mathbf{LAM}_0$. To do this we must analyze the nature of reductions. Let R' be the notion of reduction R above, restricted to terms of $\mathbf{LAM}_0$, i.e., $R' = R_\beta \cup R_E \cup R_T$.

Lemma 6.8: Let $R'_{\neg T} = R_\beta \cup R_E$. If there are term u and v of $\mathbf{LAM}_0$ such that $u \to_{R'}^* v$, then there exists a term w, such that $u \to_{\neg T}^* w$ and $w \to_T^* v$, in other words, we may postpose T-reduction to the very last.

Proof: We will show that T-reduction can be "moved past" the other two types of reduction, i.e., if

$$w_1 \to_T w_2 \to_\beta w_3$$

then there is a term w_4 such that

$$w_1 \to_\beta w_4 \to_T w_3 \, ,$$

and similarly for R_E. In other words if a T-reduction occurs before either a β-reduction (or an E-reduction), then we can replace those two reductions by a β-reduction (E-reduction respectively) followed by a T-reduction. To see this, note that a T-reduct is a single constant, therefore cannot contain a β-redex or an E-redex. Therefore any

β-redexes or E-redexes in the reduced term must be disjoint with the original T-redex, so the reductions could have been carried out in reverse order. ∎

Definition: Let $c_1, c_2, \ldots$ be new constants of ACon. A term which includes these constants is said to be a *generalized term.* If u is a generalized term and f is any total unary function on the natural numbers then we write $f(u)$ to mean the term of $\mathbf{LAM}_0$ which results from u by replacing each constant c_i by $\underline{f(i)}$. If $f(u') = u$ for some f then we say that u' *generalizes* u.

Lemma 6.9: Let f be a total unary function on natural numbers.

(i) If $(u, v) \in R_{\neg T}$ then $(f(u), f(v)) \in R_{\neg T}$.

(ii) If $u \rightarrow_{\neg T} v$ then $f(u) \rightarrow_{\neg T} f(v)$.

(iii) If $u \rightarrow^*_{\neg T} v$ then $f(u) \rightarrow^*_{\neg T} f(v)$.

Proof: It suffices to show (i). For then, if $u \rightarrow_{\neg T} v$ then there is a context $C[\,]$ such that $u = C[u_0]$, $v = C[v_0]$, and $(u_0, v_0) \in R_{\neg T}$. But then by (i), $(f(u_0), f(v_0)) \in R_{\neg T}$ and since $f(u) = f(C)[f(u_0)]$ and $f(v) = f(C)[f(v_0)]$ we have that $f(u) \rightarrow_{\neg T} f(v)$, showing (ii). To show (iii), we proceed by induction using (ii).

To show (i), we proceed by cases. If $(u, v) \in R_E$, then u is of the form cu_0 where $c \in \mathrm{ACon} \cup \{*\}$, and $v = *$. Then $f(u)$ is of the form $c'f(u_0)$, where c' is either c or a new constant c_i, and $f(v) = *$. But as $c_i \in \mathrm{ACon}$, we again have $(f(u), f(v)) \in R_E$.

If $(u, v) \in R_\beta$, then $u = (\lambda x.u_0)v_0$, $v = [v_0/x]u_0$. But then $f(u) = (\lambda x.f(u_0))f(v_0)$ and $f(v) = [f(v_0)/x]f(u_0)$. Hence, $(f(u), f(v)) \in R_\beta$. ∎

Lemma 6.10: Suppose u and v are $\mathbf{LAM}_0$ terms, f is a total unary function on natural numbers and $f(u') = u$. Then:

(i) If $(u, v) \in R_{\neg T}$ then there exists a term v' such that $f(v') = u$ and $(u', v') \in R_{\neg T}$.

(ii) If $u \rightarrow_{\neg T} v$ then there exists a term v' such that $f(v') = u$ and $u' \rightarrow_{\neg T} v'$.

(iii) If $u \rightarrow^*_{\neg T} v$ then there exists a term v' such that $f(v') = u$ and $u' \rightarrow^*_{\neg T} v'$.

Proof: Again, by a similar argument it suffices to prove (i). We show (i) by cases.

If $(u, v) \in R_E$ then $v = *$ and u is of the form cu_0, where $c \in \mathrm{ACon} \cup \{*\}$. Then u' must be of the form $c'u'_0$, where $f(u'_0) = u_0$ and c' is either c or some new constant c_i. Let $v' = *$. Since $c_i \in \mathrm{ACon}$, in either case we have $(u', v') \in R_E$, and $f(v') = v$.

If $(u,v) \in R_\beta$, then $u = (\lambda x.u_0)v_0$, and $v = [v_0/x]u_0$. Then u' must be of the form $(\lambda x.u'_0)v'_0$, where $f(u'_0) = u_0$ and $f(v'_0) = v_0$. Let $v' = [v'_0/x]u'_0$. Then $(u', v') \in R_\beta$ and $f(v') = v$. ∎

Theorem 6.11: The functions $\omega \circ \pi^n_i$, $k_m \circ \pi^n_i$, and $a_m \circ \pi^n_i$, for all natural numbers m, n, and i with $1 \leq i \leq n$, are the only functions representable by $\mathbf{LAM}_0$ terms.

Proof: Suppose an n-ary function g is represented by a $\mathbf{LAM}_0$ term G.

Case 1: $g(0, \ldots, 0)$ is undefined. Then $G\underline{0}\cdots\underline{0}$ does not reduce to a numeral. Suppose $g(i_1, \ldots, i_n) = m$ for some $(i_1, \ldots, i_n)$. Then $G\underline{i_1}\cdots\underline{i_n}$ reduces to $\underline{m}$. Then by Lemma 6.8 there is a term w such that

$$G\underline{i_1}\cdots\underline{i_n} \rightarrow^*_{\neg T} w \text{ and}$$
$$w \rightarrow_T \underline{m}\,.$$

But since the only T-reduction is of the form $\underline{\mathrm{Succ}}\,\underline{n} = \underline{n+1}$, the term w must be of the form $\underline{\mathrm{Succ}}^{(p)}\underline{i}$ for some natural numbers p and i, such that $p + i = m$. Define functions f_1 and f_2 on natural numbers by $f_1(x) = x$, $f_2(x) = 0$. Consider now the term $G' = Gc_{i_1}\cdots c_{i_n}$. Then $f_1(G') = G\underline{i_1}\cdots\underline{i_n}$. By Lemma 6.10 there is a term w' such that $G' \rightarrow^*_{\neg T} w'$ and $f_1(w') = \underline{\mathrm{Succ}}^{(p)}\underline{i}$. Then w' is either $\underline{\mathrm{Succ}}^{(p)}\underline{i}$ or $\underline{\mathrm{Succ}}^{(p)}c_i$. By Lemma 6.9, $f_2(G') \rightarrow_{\neg T} f_2(w')$. But $f_2(G') = G\underline{0}\cdots\underline{0}$, and $f_2(w')$ is either $\underline{\mathrm{Succ}}^{(p)}\underline{i}$ or $\underline{\mathrm{Succ}}^{(p)}\underline{0}$, contradicting the fact that $G\underline{0}\cdots\underline{0}$ does not reduce to a numeral. Hence g is undefined at all argument, so is equal to $\omega \circ \pi^n_i$ for any i.

Case 2: $g(0, \ldots, 0) = m$. Then $G\underline{0}\cdots\underline{0}$ reduces to $\underline{m}$. As before, this means that there is a term w such that

$$G\underline{0}\cdots\underline{0} \rightarrow^*_{\neg T} w \text{ and}$$
$$w \rightarrow_T \underline{m}\,,$$

which means that w is $\underline{\mathrm{Succ}}^{(p)}\underline{i}$. Let $G' = Gc_1\cdots c_n$. Then $f_2(G') = G\underline{0}\cdots\underline{0}$. By Lemma 6.10, there is a term w' such that $f_2(w') = w$ and $Gc_1\cdots c_n \rightarrow^*_{\neg T} w'$. But w' is then either $\underline{\mathrm{Succ}}^{(p)}\underline{i}$ or if $i = 0$, w' can be $\underline{\mathrm{Succ}}^{(p)}c_j$, for some j, where $1 \leq j \leq n$.

Now consider $g(i_1, \ldots, i_n)$. Let f_3 be defined by $f_3(x) = i_x$, for $x = 1, \ldots, n$, otherwise anything at all. Then $f_3(G') = G\underline{i_1}\cdots\underline{i_n}$. Then be Lemma 6.9, we must have $G\underline{i_1}\cdots\underline{i_n} \rightarrow^*_{\neg T} f_3(w')$.

If $w' = \underline{\text{Succ}}^{(p)}i$, then $f_3(w') = \underline{\text{Succ}}^{(p)}\underline{i}$, so $g(i_1, \ldots, i_n) = i$. Hence we have shown that g is the function $k_i \circ \pi^n_j$ for any j.

Otherwise $w' = \underline{\text{Succ}}^{(p)}c_j$. But then $f_3(w') = \underline{\text{Succ}}^{(p)}\underline{i_j}$. Hence $g(i_1, \ldots, i_n) = i_j + p$, so g is the function $a_p \circ \pi^n_j$. ▮

7. Conclusion

The result of all of the above is that we have achieved a harmonious match between a proof system for equality, a denotational semantics and an evaluator. The completeness theorem tells us that a match exists between syntax and semantics: our proof system proves exactly those equations which are valid in all models. Also, the axioms match the evaluator: the proof system proves equations $u = v$, where v is a numeral iff the evaluator can drive u to v, also, a $*$-free term u is provably equal to $*$, iff the evaluator encounters a run-time time error during the evaluation of u. Thus, the intuituion of $*$ as a notation for run-time type errors is justified.

One would like, at this point, to begin to make extensions to the language, while trying to keep this match intact. There are several ways to extend. Of course, the Normalization Claim needs to be proved, and beyond that, there is the question of how to lift the restrictions on T (i.e. the simpleness restrictions) in such a way that leads to a Church-Rosser reduction system, and an evaluator which behaves properly with respect to $*$. For instance we might want to allow equations of the form $c_1c_2 \ldots c_n = c$ into T, to better model functions that take more than one argument.

Another extension is to examine systems where the atomic elements have some structure. For example, in LISP, lists of atoms, such as (3 4 5) are terms which should behave like numerals with respect to application. Another structure construct that would be useful is Cartesian product. However, it is a result of Klop [BAR80] that the usual axioms for surjective pairing:

$$\begin{aligned} \text{left pair } x\,y &= x \\ \text{right pair } x\,y &= y \\ \text{pair (left } x)(\text{right } x) &= x \end{aligned}$$

are not Church-Rosser, when combined with (β). It is not completely clear however, whether or not it is possible to devise a Church-Rosser reduction system whose theory of equality is the same as that of (β) plus the surjective pairing axioms.

Another direction is to look at systems that have some machinery to tell atoms from non-atoms. The cond construct *almost* does the trick, but not quite. Let $u = \lambda x.\mathbf{cond}\, xxyy$; if we apply u to a numeral we will get back y, while if we apply u to a λ-abstraction we will get $*$. If we apply it something that is neither a numeral nor a λ-abstraction, the result will depend on how strong the T-axioms are, i.e. how few applications are normal forms. Still, if we apply it to something whose evaluation doesn't terminate then we get no information.

Another construct that we might consider is

$$\mathbf{case}\, uv_1v_2\,.$$

This construct comes up when we are considering models that are disjoint sums, i.e. if we are given a domain A of atoms, we seek a domain D such that $D = A + (D \rightarrow D)$. The intended meaning of $\mathbf{case}\, uv_1v_2$ is

$v_1(a)$, if $u = \text{inl}(a)$, for some $a \in A$,
$v_2(f)$, if $u = \text{inr}(f)$, for some $f \in (D \rightarrow D)$,

where inl and inr are the injections into D from A and $(D \rightarrow D)$, respectively. However, we may also run into Church-Rosser difficulties here, since the desired axioms for **case**:

$$\begin{aligned} \mathbf{case}\,(\text{inl } x)\, fg &= fx \\ \mathbf{case}\,(\text{inr } x)\, fg &= gx \\ \mathbf{case}\, x\, (h \circ \text{inl})\, (h \circ \text{inr}) &= hx \end{aligned}$$

are very similar to those for surjective pairing, if fact, they are the category theoretic dual.

If in fact the surjective pairing axioms, and the **case** axioms cannot be captured by a Church-Rosser reduction system in the untyped λ-calculus, work needs to be done on how these axioms can be weakened to yield Church-Rosser systems that still capture the "intuition" of pairing and **case**.

References

[BAR80] Barendregt, H. P.
"The Lambda Calculus — Its Syntax and Semantics"
Studies in Logic 103, North-Holland, 1981

[BCD] Barendregt, H. P., Coppo, M., and Dezani-Ciancaglini, M.
A filter lambda model and the completeness of type assignment
Journal of Symbolic Logic, to appear

[KANT29] Kant, I.
"Critique of Pure Reason"
MacMillan & Co., 1929

[LÉVY80] Lévy, J. J.
Optimal Reductions in the Lambda-Calculus
To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism
J.P. Seldin, J.R. Hindley, ed., 159–191, Academic Press, 1980

[MEYER82] Meyer, A. R.
What is a model of the lambda calculus?
Information and Control, 52, 87–122,1982

[MILNER77] Milner, R.
Fully abstract models of typed λ-calculi
Theoretical Computer Science, 4, 1–22, 1977

[MILNER78] Milner, R.
A theory of type polymorphism in programming
Journal of Computer and System Sciences, 17, 348–375, 1978

[PLOTKIN75] Plotkin, G. D.
Call-by-name, call-by-value, and the lambda calculus
Theoretical Computer Science, 1, 125–159, 1975

[PLOTKIN77] Plotkin, G. D.
LCF considered as a programming language
Theoretical Computer Science, 5, 223–255, 1977

[SCOTT76] Scott, D. S.
Data types as lattices
SIAM Journal on Computing, 5, 522–587, 1976

SCOTT81] Scott, D. S.
Lectures on a mathematical theory of computation
Oxford Univ. Computing Lab., Tech. Mono. PRG-19, 1981

[STOY77] Stoy, J. E.
"Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory"
MIT Press, Cambridge MA, 1977

[WAND84] Wand. M.
What is LISP?
American Mathematical Monthly, 91, 9, 1984