

Recursion-theoretic Techniques for Denotational Semantics

Ryan J. Wisnesky
Philosophy 351A

March 2006

Introduction

Many areas of computer science directly sprang from recursion theory proper; for instance, the study of complexity classes P and NP [11]. Other areas, such as domain-theory, developed more independently. In this paper we consider recursion-theoretic techniques in the heart of language theory: the denotational semantics of lambda calculus.

My reasons for choosing this topic are twofold. First, as a computer science student, I have used this paper as an opportunity to both solidify my understanding of concepts I learned earlier in CS258 and to further explore topics in that class that we did not cover. Second, as a recursion theory student, I have used this paper to explore extremely interesting but tangential topics, i.e. realizability, that I do not believe I would have seen otherwise. As such, this paper is essentially a walk-through of other papers, with additional commentary by me and a few avenues explored in detail greater than that given in the texts. At the very least, writing this paper has forced me to examine in detail a new area, and I also believe this paper provides a more straightforward explanation of *why* the development of recursion-theoretic models proceeded as it did; in [1], the motivational threads tend to become frayed during the intensive development of the theory.

That there is such a natural connection between computer science and recursion theory within language theory that enables me to do this is one of the reasons I enjoy the field so much.

1 Denotational Semantics

Simply put, *denotational semantics* is the study of what programs denote. It is one of the three main mechanisms for understanding programs, the other two being *operational semantics* – the study of how programs compute – and *axiomatic semantics* – the study of equality of programs. The three mechanisms are intimately related, and denotational

semantics is often used to reason about operational semantics, which in turn is useful for program optimization [1, p78].

Lambda calculus is a useful tool in studying programming languages because of the following facts:

1. Most programming languages are Turing-complete; that is, all partial recursive functions on natural numbers may be defined in them,
2. Lambda calculus, in its many forms, both typed and untyped, and with many reduction systems (strict, lazy, left-most, etc.) also defines all partial recursive functions on natural numbers,
3. For many languages, especially functional ones, there are very natural ways of converting programs to lambda terms, and vice versa.

Therefore, rather than studying many different languages, it is useful to instead to study lambda calculus. In fact, many programming languages have actually been created based on lambda calculus (e.g. LISP), and ideas from the study of lambda calculus often percolate back into language design (e.g. the compile-time checking of ML typing) [1, p607].

Of course, the situation is slightly more complicated than the rosy picture just painted. For instance, translating programs that take input and output take special consideration, and parallelism is not readily expressible in the lambda calculus. However, both of those concerns have led to further interesting research areas, on monads [7] and actors [8], respectively.

In short, to study denotational semantics is to study what terms in lambda calculus denote. To do so, we must construct a *model* for lambda calculus, where every lambda term has a denotation in the model, and where provably equal lambda terms have the same denotation. (The idea of provably equal terms having the same denotation raises subtle issues, which we will not address here [1, p355].)

2 Domain Theory

Domain theory has applications in the denotational semantics of both typed and untyped lambda calculus. The field grew out of the study of models for untyped lambda calculus by Dana Scott in the 1970s [2, p9], but has since become an independent sub-field of mathematics in its own right [9]. These domains have interesting connections to other areas of mathematics, most notably topology [10].

In this section, we give a brief overview of the problems that motivated the discovery of domain theory and we describe the general properties that any model of lambda calculus must have. Although in our examination of recursion theoretic techniques we will be considering only typed lambda calculus, we do not limit ourselves to only typed lambda calculus here, as concepts from domain theory for both calculi will be useful later.

Consider first untyped lambda calculus. By untyped lambda calculus we mean any lambda calculus without types and with reasonable reduction rules and axioms; the particular system studied does not matter, so long as it is not pathological. We wish to find a denotation for every lambda term; it seems reasonable to associate some kind of function with each term. (Natural numbers are represented as lambda terms in an ingenious way, discovered by Kleene [2, p136], that allows for functions like `if then else` to be defined. Hence, we may regard natural numbers as functions in this model. The key point is that we have only one “type” to consider.) Such a model would have to possess the property that its domain D is isomorphic to $D \rightarrow D$. (The notation $D \rightarrow D$ is taken to mean the set of all functions from D to D .) Consider first why elements with a denotation in D may also have a denotation in $D \rightarrow D$. This direction of the isomorphism is required because lambda terms operate on other lambda terms; in other words, a term can be viewed as a function from a term to another term. For instance, the term

$$(\lambda x. \lambda y. xy)(\lambda k. k)$$

must clearly denote something in D , since all terms must have a denotation there, but it must also denote something in $D \rightarrow D$: this is because $(\lambda k. k)$ denotes something in D , and so does the result of the application, $\lambda y. (\lambda k. k)y$. For the other direction of the isomorphism, simply note that any term that denotes something in $D \rightarrow D$ must also denote something in D simply because it is a term and all terms denote something in D .

In short, our model must contain its own function space. This is an extremely interesting property and at first glance it seems like such a model is impossible because of cardinality concerns: if we have some set D , then if we let $D \rightarrow D$ contain all functions from D to D , the cardinality of $D \rightarrow D$ will be strictly greater than the cardinality of D . This violates our required isomorphism, and so we must conclude that $D \rightarrow D$ must not contain all functions from D to D . However, it is not obvious which functions are to be included and excluded. (A recursion theorist would undoubtedly say that we should include only partially computable functions, thus necessitating the study of computability on arbitrary domains. This is in fact the approach developed in the sources this paper follows.) Classical Domain theory provides a mechanism – continuity on Scott’s topology – for limiting the set of functions included in $D \rightarrow D$ so that every term has a denotation and the isomorphism holds.

For typed lambda calculus, the above situation is not a problem because typing rules do not allow the construction of terms with denotations in both D and $D \rightarrow D$. In other words, every term must have exactly one type. Let the notation \mathcal{A}^σ mean the domain of type σ in a model \mathcal{A} (a type is simply a set of objects from \mathcal{A}); then the typing rules prohibit $\mathcal{A}^\sigma \cap \mathcal{A}^{\sigma \rightarrow \sigma} \neq \emptyset$.

However, we still run into trouble when interpreting the *fixed point operator*. One of the axioms of typed lambda calculus is

$$fix_\sigma(f) = \lambda f : \sigma \rightarrow \sigma. f(fix_\sigma f)$$

or, written more informally,

$$f(fix_\sigma(f)) = fix_\sigma(f)$$

for each type σ . This axiom is required to introduce recursion into typed lambda calculus; in fact, without the presence of this operator, it is impossible to write terms without a normal form. In other words, without *fix*, typed lambda calculus cannot define any partial functions at all [1, p60].

Blurring the lines between function symbols and their denotations, we have that because f is an arbitrary term of type $\sigma \rightarrow \sigma$, $fix_\sigma(f)$ must have a denotation in \mathcal{A}^σ for every $f \in \mathcal{A}^{\sigma \rightarrow \sigma}$. However, if we let $\mathcal{A}^{\sigma \rightarrow \sigma}$ contain all functions from \mathcal{A}^σ to \mathcal{A}^σ , then there will be functions in $\mathcal{A}^{\sigma \rightarrow \sigma}$ that do not contain fixed points, and hence the axiom fails in the model. Thus, we must restrict $\mathcal{A}^{\sigma \rightarrow \sigma}$ to contain only functions with fixed points.

One final note about typed lambda calculus is that we would like non-terminating computations to have a type. That is, the term

$$\lambda x. \text{ if } x = 3 \text{ then } 4 \text{ else } \textit{no - normal - form}$$

should have type integer even though during reduction it may never produce a normal form. In general, the way this is accomplished is that for every type σ we must have, in each type domain, a distinguished element \perp_σ . It is also possible to work with partial function spaces, but this is not the approach taken by Mitchell [1, p373].

The basic idea of domain theory is to develop techniques to allow these challenges in constructing models to be overcome. Conveniently, the same techniques developed to help untyped lambda calculus are also useful in typed lambda calculus, and vice versa.

3 Preliminaries

Domain theory has been studied to such an extent that a brief introduction to the subject, with proofs and definitions, comprises only a few pages in the foundational text of Barendregt [2]. It is my personal belief that among all branches of mathematics, domain theory has the highest ratio of time to develop an intuition of the mechanisms to complexity of definitions. Because many of the concepts developed in domain theory will be used later, we give their definitions here. These definitions are taken from Mitchell [1]; equivalent definitions may be found in Barendregt [2]. Most of the differences are trivial; for instance, Barendregt considers all CPOs to be pointed. One non-trivial difference is that Barendregt directly considers the Scott topology of a CPO and uses category theory to show that projective limits of sequences of CPOs exist; this limit process, along with the concept of retraction, is used to directly construct domains with the property $D \cong D \rightarrow D$. Mitchell, in contrast, does not consider the formal topology and instead directly constructs a Henkin model from CPOs for every type. The two approaches are related but are indicative of the differences required when modeling untyped versus typed lambda calculus.

A *partial order* $\langle D, \leq \rangle$ is a set D with a reflexive, anti-symmetric and transitive relation \leq . An *upper bound* of a subset $S \subseteq D$ is an element $x \in D$ with $y \leq x$ for every $y \in S$. A *least upper bound* of S is an upper bound which is \leq every upper bound of S . Least upper

bounds are unique, by symmetry of \leq . A subset $S \subseteq D$ is directed if every finite $S_0 \subseteq S$ has an upper bound in S . A *complete partial order (CPO)* is a partial order $\langle D, \leq \rangle$ such that every directed $S \subseteq D$ has a least upper bound, denoted $\vee S$. A CPO with a least element is called *pointed*. (This name derives from the fact that when drawing a CPO, the least element sticks out of the bottom of the graph.) A set D is *lifted* by adding a *bottom* element \perp so that $x \leq y$ iff $x = y$ or $x = \perp$. The CPO $\langle D \cup \{\perp\}, \leq \rangle$ is called D lifted and is denoted D_\perp .

Let $\mathcal{D} = \langle D, \leq_D \rangle$ and $\mathcal{E} = \langle E, \leq_E \rangle$ be CPOs, and let $f : D \rightarrow E$. When $S \subseteq D$, we write $f(S)$ for the subset of E given by $f(S) = \{f(d) \mid d \in S\}$. f is *monotonic* if $d \leq d'$ implies $f(d) \leq f(d')$. A monotonic function f is continuous if for every directed $S \subseteq D$, $f(\vee S) = \vee f(S)$. (Cutland [6, p184] gives a definition of continuity of recursive operators that is more intuitive than this definition, but is more limited in scope, in the sense that Cutland's definition cannot apply to arbitrary CPOs.)

Mitchell models typed lambda calculus by associating a pointed CPO with each type and showing that continuous functions from one type CPO to another also form a pointed CPO; in these type CPOs, every function has a fixed point. The construction is straightforward; for instance, the type of integers is modeled by \mathcal{N}_\perp . In contrast, Barendregt's approach is far more abstract, although it does provide a closer connection with topology.

Although the construction of a typed lambda calculus model from these definitions is straightforward, these models are somewhat far removed from our intuitive notions of computability. Interpreting bottom elements \perp as representing "undefined computations" is relatively intuitive, but passing from there to identifying continuity as the key idea is not. That is, continuity on partial orders does not mesh well with our intuitive understanding of lambda terms as defining computable functions. We would like to interpret $\mathcal{A}^{\sigma \rightarrow \sigma}$ as the set of all *computable* functions from \mathcal{A}^σ to \mathcal{A}^σ , for some appropriate and hopefully intuitive sense of the word computable, rather than as continuous functions. However, we do not have a notion of computability on arbitrary domains. It is this area in which recursion theory can be helpful. In developing a recursion-theoretic model of typed lambda calculus, we will both develop this notion of computability and find a connection between continuity and computability.

4 Outline

The rest of this paper follows the development of recursion-theoretic models found in Mitchell [1, ch5.5]. As such, this section borrows heavily from his work; some definitions are taken verbatim. I have tried to focus only on the core of the argument, omitting ancillary results and proofs, and adding commentary about how the material is considered recursion theoretic, and exploring some directions in greater detail than in his treatment. An outline of the material is as follows:

1. *Computability on Other Domains.* A model for typed lambda calculus intuitively con-

sists of functions, pairs, and natural numbers. (This is because these are taken to be the primitives in the definition of the particular lambda calculus studied, PCF.) Recursion theory provides a method for coding such concepts into natural numbers. So, to begin, we must establish how to represent these concepts, in a way that is useful for modeling typed lambda calculus, inside the natural numbers. This leads to the study of modest sets and computability on these sets. The history of these sets is also discussed; this digression is not found in Mitchell.

2. *Partial Equivalence Relations.* In many ways, we are more interested in how our objects/arbitrary domains are represented as natural numbers than we are with the objects themselves. For instance, while we are constructing our model, it is more convenient and often necessary to speak of a number n rather than ϕ_n . The use of partial equivalence relations (pers) is a way of defining modest sets that places more emphasis on the codings than on the set being studied. Pers are the fundamental tool in the following sections, and they also model typed lambda calculus without *fix* in a straightforward way.
3. *Partial Combinatory Algebras.* Pcas are a generalization of pers where the domain of the enumeration function is not restricted to \mathcal{N} . In addition, every pca defines a partial function \cdot which we will use for function application. Pcas have interesting connections to the theory of combinators and other areas of computer science, in addition to their usefulness in modeling. They are considered here because the operation \cdot is useful in modeling recursion.
4. *Domain theory returns.* Pers and pcas provide almost enough machinery to create a model. However, arbitrary functions on pers do not necessarily have fixed points. Therefore, we are forced to use concepts from domain theory to identify a specific class of pers where the fixed point property holds. This convergence of concepts provides the link between continuity and computability, and allows us to create our recursion-theoretic model of typed lambda calculus.

On a final note, I do not know why CPO is abbreviated in all capitals, whereas per and pca are not.

5 Computability on Other Domains

As a first step in constructing our model we need to develop a notion of computability on domains other than \mathcal{N} . The standard way to do this, taken in both Cutland [6, p66] and Mitchell, is through the use of a coding function that links \mathcal{N} and the other domain. A *modest set* is a pair $\langle A, e_A \rangle$ where A is a set and $e_A : \mathcal{N} \rightarrow A$ is a surjective partial function. e_A is the partial enumeration function for A ; when $e_A(a) = n$ we say that n codes a . e_A must be surjective because every element in A must have a code. Because e_A will simply be

used to define a class of recursive functions on A , procedures for determining if two numbers code the same element in A are not required. The recursive functions on modest sets are the functions that may be effectively computed from the underlying codes.

Various results about modest sets may be established easily; among these, that for modest sets A and B , both $A \times B$ and $A \rightarrow B$ are modest sets, with corresponding enumeration functions $e_{A \times B}$ and $e_{A \rightarrow B}$. The details of these constructions are given in Mitchell and are not repeated here.

5.1 Realizability

There is a slight complication with the definition of modest sets, at least when we are trying to code functions on partial recursive functions on integers. That is, it is possible to choose an enumeration function e such that the usual computable functions on partial recursive functions may not be recursive functions $\langle \mathcal{PR}, e \rangle \rightarrow \langle \mathcal{PR}, e \rangle$. Mitchell says only that the reason is exceedingly complex; and he is definitely correct; an account of my investigation of this topic follows.

There is an interesting connection between modest sets, realizability, category theory, and constructive mathematics. Realizability was originally defined by Kleene [5] in his investigations of constructive mathematics in the 1940s. To a constructivist, a formula $\exists x \phi(x)$ is only regarded as true when there is an “effective procedure” for determining what x is; likewise, formulas such as $\alpha \rightarrow \beta$ may be regarded as true when there is an effective procedure for transforming a proof of α into a proof of β . A number n realizes a formula ϕ when it provides the mechanism or witness or code for this effective procedure; Kleene formalizes this in his paper. Intuitively, it seems like there should be some connection between this notion of “effective realizability” and computability. However, the connection was not made until many years later.

In [4], Hyland formulates the notion of realizability in terms of category theory. (Interestingly, around this time there seems to be a major shift in programming language research toward more category-theoretic techniques. I speculate that this is because modeling more expressive type systems, for instance, type systems allowing polymorphism, requires the more general framework given by category theory, instead of conventional Henkin models.) In this paper, he discovers the “effective topos” in which the realizability constraints he is modeling force operations on the objects in the topos to be “effective” in a computable-like way. (Unfortunately, understanding the full details requires far more knowledge of all four fields than I possess.)

In [3], Hyland discovers that the effective topos contains a small complete category, namely the category of modest sets. He also examines the category of pers. It is from these definitions that the notion of an effective enumeration may be examined; however, there is simply no way to detail these discoveries without depending on a significant amount of category theory. In a way, this is to be expected, as even thinking about Gödel numbering formulas requires a fair amount of mathematics; enumerating modest sets so as to obtain an

exact correspondence between recursive functions on partial recursive functions and recursive functions on partial recursive functions as modest sets cannot be trivial. At the very least, it requires a detailed understanding of the properties of the class of partial recursive functions on \mathcal{N} and an understanding of the category of modest sets.

5.2 Direct applications of Recursion Theory

It is interesting to note that it is possible to work many of the problems in Mitchell's text using only references from Cutland. Here is one example.

Exercise 5.5.9. Since application is lambda definable, and we want the meaning of every lambda definable function to be computable, it is essential that given a code n for a function f and a code m for an element a in the domain of f that we can compute a code for $f(a)$. Given this, show that if \mathcal{N} is the modest set of natural numbers, enumerated by the identity function, there is no total function $e_{\mathcal{N} \rightarrow \mathcal{N}}$ for the modest set of total recursive functions from \mathcal{N} to \mathcal{N} .

Solution The set $S = \{f : \mathcal{N} \rightarrow \mathcal{N} \mid f \text{ is total}\}$ is not recursively enumerable. This is because to be so, the predicate “ f is total” would have to be partially decidable. By Cutland p119 ex7, we know that it is not. So, if we had a total surjective function $e : \mathcal{N} \rightarrow S$, then S could be enumerated as $S = \{e(0), e(1), \dots\}$. We could then write a partial recursive characteristic function $g = \mu k (e(k) = \text{code for } f)$ which halts exactly when f is total, thus showing that S is r.e.

5.3 A Modest Set Model Without *fix*

Modest sets provide a straightforward way to model typed lambda calculus without *fix*.

This construction makes use of the concept of an applicative structure. Formally, a great deal of machinery is involved in defining what exactly such a structure is [1, p280]. However, the intuition is captured by thinking that such a structure simply gives a meaning to every constant symbol and specifies how application and pairing is interpreted. Applicative structures are Henkin models, and because the intuition is so apparent, we will not give the full definition here.

The full construction of our model is straightforward. Formally, let $\Sigma = \langle B, C \rangle$ be a signature for typed lambda calculus with pairing and function application. That is, $B = \{b_0, b_1, \dots\}$ is a countable collection of type symbols and C is a collection of constant symbols. Let $\langle A^{b_i}, e_i \rangle$ be a modest set for each $b_i \in B$. A *full recursive hierarchy* \mathcal{A} for Σ over modest sets $\langle A^{b_0}, e_0 \rangle, \dots, \langle A^{b_k}, e_k \rangle$ is an applicative structure with

$$A^{\sigma \times \tau} = A^\sigma \times A^\tau$$

$$A^{\sigma \rightarrow \tau} = \text{all recursive } f : \langle A^\sigma, e_\sigma \rangle \rightarrow \langle A^\tau, e_\tau \rangle$$

where these sets are enumerated by $e_{A^\sigma \times A^\tau}$ and $e_{A^\sigma \rightarrow A^\tau}$. Application and pairing work exactly as expected: the denotation of fx is $f(x)$, $p_1 \langle x, y \rangle$ denotes x and $p_2 \langle x, y \rangle$ denotes

y. Denotations for constant symbols may be chosen arbitrarily, provided typing restrictions are obeyed.

Such a model is exactly what we were looking for: a recursion-theoretic model of typed lambda calculus (albeit without *fix*). In this model, every function is interpreted in a straightforward way using a generalized concepts of computability over modest sets. Unfortunately, more machinery is required to interpret *fix*. In defining this machinery it is more convenient to work with a concept equivalent to modest sets, but with fewer technical details: partial equivalence relations.

6 Partial Equivalence Relations

Modest sets provide a way to think about computability on arbitrary sets. However, we are more interested in how the coding function e_A works than we are in the set underlying the modest set, A . In other words, it makes sense to try to move away from thinking about modest sets to reasoning directly about the coding function. Notice that the enumeration function $e : \mathcal{N} \rightarrow A$ divides \mathcal{N} into equivalence classes based on the element in a mapped to; for instance, if $A = \{red, blue\}$ and $e_A(0) = e_A(1) = red$ and $e_A(3) = blue$, we would say that $0, 1$ form an equivalence class and that 3 forms an equivalence class. In other words, $\{(0, 0), (0, 1), (1, 1), (1, 0)\} \subset \mathcal{N} \times \mathcal{N}$ is an equivalence class and so is $\{(3, 3)\} \subset \mathcal{N} \times \mathcal{N}$. Because e is partial, some elements in \mathcal{N} are not part of any equivalence class; thus, the enumeration function defines a set of *partial equivalence relations* (*pers*). Our motivation for thinking about pers is simply that much of the later development is easier using pers rather than modest sets.

Mitchell spends some time describing how given a per, we may construct a modest set, and vice-versa. For our purposes, we will take this for granted and instead examine how pairing and function application work in pers. Let $R, S \subseteq \mathcal{N} \times \mathcal{N}$ be pers. We define the pers $R \times S$ and $R \rightarrow S$ as

$$n(R \times S)m \text{ iff } (n)_1 R (m)_1 \text{ and } (n)_2 R (m)_2$$

(Intuitively, two pairs $((n)_1, (n)_2)$ and $((m)_1, (m)_2)$ are equivalent in $R \times S$ only when their corresponding components are equivalent in R and S .)

$$m_1(R \rightarrow S)m_2 \text{ iff } \forall n_1, n_2 \in \mathcal{N}. n_1 R n_2 \text{ implies } \phi_{m_1}(n_1) S \phi_{m_2}(n_2)$$

(Intuitively, two functions m_1 and m_2 are equivalent if when their inputs are related in R , their outputs are related in S . This assumes the function application halts.) We define the per $R \multimap S$ later.

We may begin to see a correspondence between pers and types with the following observation: each per defines a type; the type is the equivalence class on \mathcal{N} . This connection is suggested by the notation

$$n : R \text{ iff } n R n$$

Where we can read $n : R$ as saying that n has type R , just as we would for a typing assertion.

This is all of the development of pers that we require. However, pers themselves are just modest sets in disguise; they cannot model typed lambda calculus with *fix* any more than modest sets can. But before we examine what extra machinery is required to model *fix*, we first generalize the concept of pers to partial combinatory algebras.

It is worthwhile to note that the connection between types and pers is not only useful and insightful, but is more natural in the setting of pers than of modest sets.

Finally, we will often move from thinking about modest sets to thinking about pers, and vice-versa, whenever one concept is more convenient than the other.

7 Partial Combinatory Algebras

The definition of modest sets (and hence pers) may be generalized to use partial enumeration functions from any set D instead of \mathcal{N} , as long as we are able to code functions and pairs as elements of D . In so doing, we are generalizing modest sets to structures called *partial combinatory algebras (pcas)*. A pca is a structure $\mathcal{D} = \langle D, \cdot, K, S \rangle$ where D is a set, \cdot is a partial binary operation, and $K, S \in D$ have the properties that for every $x, y, z \in D$,

$$(K \cdot x) \cdot y = x$$

$$(S \cdot x) \cdot y \downarrow$$

$$((S \cdot x) \cdot y) \cdot z = (x \cdot y) \cdot (y \cdot z), \text{ if } (x \cdot y) \cdot (y \cdot z) \downarrow, \text{ undefined otherwise}$$

These axioms for K and S are dear to the computer scientist's heart. If we let D be the set of all untyped lambda terms and \cdot denote application, then K and S may be defined as

$$K = \lambda x. \lambda y. x$$

$$S = \lambda x. \lambda y. \lambda z. (xz(yz))$$

Here, K is used to define constant functions and S is used as a sort of “super-application.” It is a well-known fact that every untyped lambda term can be written in terms of K and S .

Returning to generalizing modest sets, for any pca \mathcal{D} , a \mathcal{D} -modest set is a pair $\langle A, e_A \rangle$ where $e_A : D \rightarrow A$ is a surjective “enumeration” function on D .

Let R, S be \mathcal{D} -modest sets. If we wish to interpret $R \times S$ and $R \rightarrow S$ as modest sets, then we must be able to define pairing in terms of elements of D . Fortunately, K and S have precisely the properties required to define pairing; a detailed proof is given in Mitchell.

Note that when we start “working inside” a pca, we take \cdot from the pca and then slice up the pca into pers, using \cdot for function application. Often we use \mathcal{N} and the pca \mathcal{N} interchangeably; the elements S and K for the pca of natural numbers are not used in the development below but are defined in the text.

8 Domain Theory Returns

Our final goal is to use the pca of natural numbers with $n \cdot m$ defined as $\phi_n(m)$ to construct a model of typed lambda calculus with *fix*. However, the failure of the fixed point theorem on certain classes of pers requires concepts from domain theory to be used in restricting what functions are allowed in our sets of function types. Thus, our model will be an interesting mix of recursion theory and classical domain theory. It is also at this point that our work with pers will pay off, as this development is easier with pers than with modest sets.

To begin, we define partial function pers. That is, let $R, S \subseteq \mathcal{N} \times \mathcal{N}$ be pers. Then the per $R \rightarrow S$ of computable partial functions from R to S is given by

$$m_1(R \rightarrow S)m_2 \text{ iff } \forall n_1, n_2 \in \mathcal{N}. \text{ if } n_1 R n_2 \text{ then} \\ \text{if } m_1 \cdot n_1 \downarrow \text{ or } m_2 \cdot n_2 \downarrow \text{ then } m_1 \cdot n_1 S m_2 \cdot n_2$$

Intuitively, this reads as “ m_1 and m_2 represent the same partial function from R to S if, whenever n_1 and n_2 represent the same element of R , if either $m_1 \cdot n_1$ or $m_2 \cdot n_2$ represent an element of S , then both must be defined and represent the same element. A slightly weaker but perhaps more intuitive condition is

$$m : R \rightarrow S \text{ implies } \forall n \in \mathcal{N}. \text{ if } n : A \text{ then if } m \cdot n \downarrow \text{ then } m \cdot n : S$$

The proof that for all pers $R, S \subseteq \mathcal{N} \times \mathcal{N}$, the relation $R \rightarrow S$ is a per is given in Mitchell. It is straightforward and quite important, but we omit it here.

We are almost ready to define *fix*, but we have one detail to take care of first: we need the ability to represent non-termination; we wish to represent partial functions as total functions with an “undefined” element. That is, we desire that

$$R \rightarrow S \cong R \rightarrow S_{\perp}$$

To that end we introduce the one element per *unit*:

$$unit = \{(0, 0)\}$$

There are alternative definitions of *unit* because there are many one unit pers; this definition yields the simplest proofs in this treatment. (A brief aside: the type *unit* often plays a part in functional programming languages where “void” operations are required. For instance, in ML, the type of “writing a file” is of type *unit*.)

We make use of *unit* to help with non-termination as follows: if $S \subseteq \mathcal{N} \times \mathcal{N}$ is a per, define

$$S_{\perp} = unit \rightarrow S$$

Intuitively, because *unit* has exactly one element, there are $|S|$ total functions from *unit* to S , and one partial function; the one partial function becomes identified with \perp – this is exactly the property we need for S_{\perp} .

8.1 The Failure of the Recursion Theorem

Before we proceed, we must define a simple partial function per $N \rightarrow N$, where $N \subseteq \mathcal{N} \times \mathcal{N}$ is the per $\{\langle n, n \rangle \mid n \in \mathcal{N}\}$. Here n codes the n -th partial recursive function, and so we have that $n_1(N \rightarrow N)n_2$ exactly when $n_1 \cdot m = n_2 \cdot m$ for all $m \in \mathcal{N}$, that is, when n_1 and n_2 are indices for the same partial recursive function. In the following discussion, it is essential to differentiate between $N \rightarrow N$ and $\mathcal{N} \rightarrow \mathcal{N}$.

We are ready to investigate how to define *fix*. We know that every total function from $N \rightarrow N$ to $N \rightarrow N$ has a fixed point; this is Kleene’s recursion theorem, which is one of the fundamental results of recursion theory. Consider n , where n codes a function in $(N \rightarrow N) \rightarrow (N \rightarrow N)$. By the fixed point theorem we know that there is some $k \in \mathcal{N}$ such that $n \cdot k \downarrow$ and $\phi_k = \phi_{n \cdot k}$. Therefore the equivalence class $[k]_{N \rightarrow N} = [n \cdot k]_{N \rightarrow N}$ is a fixed point of the function given by n . At last, it seems like we have discovered how to define *fix*.

However, any celebration would be premature. Recall that we are actually operating on the natural numbers lifted, not the natural numbers. Unfortunately, the recursion theorem does not hold on arbitrary domains. One way to construct a counterexample is to consider pers with finite domains and use that that domain is “incomplete” to show that a specific function’s fixed point must lie outside of the finite domain. The end result of this development is the understanding that fixed-points of total recursive functionals on all lifted pers don’t necessarily exist. But because lifting is essential to understanding recursion, we are forced to limit ourselves to a special type of per where the recursion theorem holds. This is where domain theory returns in earnest.

8.2 Passing The Test

It is not enough to simply order a given per as we would in domain theory; domain theory allows for finite types but above we discussed an example of a finite per where the fixed point theorem fails. In addition, simply ordering the per does not guarantee a CPO. Thus, rather than start with an arbitrary per and order it, we instead investigate a class of pers that are already CPO-like.

To do this, the notion of *passing the test* is introduced. This notion is supposed to be a generalization of the concept in domain theory that one function is greater than another (in the sense of the ordering) if it is defined for more arguments and matches on the others. We say that an element a of a CPO “passes the test” given by a continuous function $f : A \rightarrow \{\top, \perp\}$ if $f(a) = \top$ and “fails the test” if $f(a) = \perp$. It can be shown that for an algebraic CPO, $a \leq b$ iff for all continuous $f : D \rightarrow \{\top, \perp\}$, if $f(a) = \top$ then $f(b) = \top$. This is the generalization: b is greater than a if it passes all the same tests that a does.

With this notion of tests introduced, we attempt to order by tests. To that end, define a *computable test* on a per $A \subseteq \mathcal{N} \times \mathcal{N}$ as any partial recursive function $A \rightarrow \text{unit}$. Because $\{\top, \perp\}$ is the result of lifting a one element CPO, and $A \rightarrow \text{unit} \cong A \rightarrow \text{unit}_\perp$, and unit_\perp is the result of lifting a one element CPO, this definition of test is equivalent to the previous

one. In this view of tests, an element $n : A$ “passes the test” $f : A \rightarrow \text{unit}$ if $f \cdot n \downarrow$. We then have a kind of intrinsic order on pers: if $A \subseteq \mathcal{N} \times \mathcal{N}$ is any per and $f : A \rightarrow \text{unit}$, then the *intrinsic preorder* on A is defined by

$$n \leq_A m \text{ iff } \forall f : A \rightarrow \text{unit}. f \cdot n \downarrow \text{ implies } f \cdot m \downarrow$$

This relation is a preorder that is not necessarily antisymmetric. With a preorder on every per, we are able to identify a class of CPO-like pers, namely, pers where the intrinsic preorder is a complete partial order. However, we must take a restricted view of completeness: a per on the natural numbers can have only countably many elements. However, a countable partial order may have uncountably many directed sets, which leads to cardinality trouble. An example illustrates this best: the per $N \rightarrow N$ contains all functions with finite domain, since every partial function with finite domain is computable. Every total function on the natural numbers is the least upper bound of some directed set of partial functions with finite domain. There are uncountably many total functions, so no per can be the full CPO of all partial functions on the natural numbers.

Luckily, we do not require least upper bounds of all directed sets: for interpreting a countable language, we only need least upper bounds of computable increasing sequences. Thus we may define “effective” or “computable” CPOs. Let A be any per. A *computable increasing sequence* in A is a computable function $s : N \rightarrow A$ with $s \cdot n \leq_A s \cdot (n+1)$ for each $n \in N$. A least upper bound of a computable increasing sequence s is an element a whose equivalence class is the least upper bound of $s \cdot 0, s \cdot 1, s \cdot 2, \dots$. A per A is an *effective CPO* if the intrinsic preorder on A is antisymmetric and there is a computable supremum function with index sup_A such that whenever $s : N \rightarrow A$ is a computable increasing sequence in A , then $\text{sup}_A \cdot s$ is a least upper bound of s in A . In other words, we must have a computable function that finds the least upper bound of every computable increasing sequence.

Every effective CPO is a per over the natural numbers, and so there are two classes of function from one effective CPO to another: the computable functions on pers and the continuous functions on CPOs. With effective CPOs, the computable functions on the underlying pers are continuous: this establishes a necessary link between the recursion-theoretic concept of computability and the domain-theoretic concept of continuity. Intuitively, this means that every computable function on a pointed effective CPO has a least fixed point. This is exactly what we wanted.

However, the proof that every computable function on a pointed CPO has a fixed point is long and somewhat tedious. The main theorem is that when a per $A \subseteq \mathcal{N} \times \mathcal{N}$ is a pointed effective CPO, then every computable $f : A \rightarrow A$ has a least fixed point $\text{fix}_A f$, with $\text{fix}_A : (A \rightarrow A) \rightarrow A$ computable. Thus, we have domains within which we may interpret *fix*. The final details required, namely to show that for pers A and B that are effective CPOs, pers $A \times B$, $A \rightarrow B$ and $A \rightarrow B$ are effective CPOs, are routine.

Mitchell’s treatment ends here, without taking the final step of showing how pointed effective CPOs define types in a model for typed lambda calculus. Perhaps he expects that any reader that has made it this far will undoubtedly be able to do this for himself.

Conclusion

The path to developing a recursion-theoretic model of typed lambda calculus is long and draws on many different areas of recursion theory and domain theory. Given that domain-theoretic models are in a certain sense easier to develop, one might wonder why a recursion-theoretic model is useful at all. Given the wide range of topics further explored in the development of the recursion-theoretic model, for instance combinators and realizability, it seems like the study of recursion-theoretic models is useful if only because it leads to interesting connections between other areas.

But the study is also useful in its own right, as it is another step in the direction of discovering fully abstract models of typed lambda calculus; these models are useful in program optimization and are also intrinsically beautiful.

But, perhaps the most important reason that the recursion-theoretic model is useful is because of the connection it provides between continuity and computability, finally allowing our intuition of what lambda calculus does to be understood in an intuitive way.

References

- [1] Mitchell, J. *Foundations for Programming Languages*. MIT Press, 1996.
- [2] Barendregt, H. *The Lambda Calculus*. North-Holland Publishing Company, 1981.
- [3] Hyland, J. *A Small Complete Category*. In *Annals of Pure and Applied Logic* 40, 1988.
- [4] Hyland, J. *The Effective Topos*. In *Studies in Logic and the Foundations of Mathematics*, volume 100: The L. E. J Brouwer Centenary Symposium, 1981.
- [5] Kleene, S. *Realizability: A Retrospective Survey*. In *Lecture Notes in Mathematics* 337: Cambridge Summer School in Mathematical Logic, 1971.
- [6] Cutland, N. *Computability*. Cambridge University Press, 1980.
- [7] Moggi, E. *Notions of Computation and Monads*. *Information And Computation*, 93(1), 1991.
- [8] Clinger, W. *Foundations of Actor Semantics*. PhD thesis, MIT, May 1981.
- [9] Muller, R. et al. *Domain theory for Nonmonotonic Functions*. Harvard University Center for Research in Computing Technology TR-11-90, 1990.
- [10] Mislove, M. *Topology, Domain Theory and Theoretical Computer Science*. In *Topology and Its Applications*, vol. 89, p.3-59, 1998.
- [11] Cobham, A. *The Intrinsic Computational Difficulty of Functions*. In *Logic, Methodology and Philosophy of Science II*, 1964.