Can LCF Be Topped? Flat Lattice Models of Typed λ-Calculus*

Bard Bloom**

TR 89-1073
December 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*To appear in Information and Computation Special Issue for LICS '88.
**This research was supported by an NSF Fellowship, also NSF Grant No. 8511190-DCR and ONR Grant No. N00014-83-K-0125.

Can LCF Be Topped? Flat Lattice Models of Typed λ-Calculus

To appear in **Information and Computation** Special Issue for LICS '88

Bard Bloom*

Department of Computer Science, Cornell University

December 20, 1989

1 Introduction

Abstract: *Plotkin, [Plo77], examines the denotational semantics of PCF (essentially typed λ-calculus with arithmetic and looping). The standard Scott semantics* V *is computationally adequate but not fully abstract; with the addition of some parallel facilities, it becomes fully abstract, and with the addition of an existential operator, denotationally universal. We consider carrying out the same program for* $\diamondsuit$, *the Scott models built from flat* lattices *rather than flat cpo's.*

Surprisingly, no computable extension of PCF can be denotationally universal; perfectly reasonable semantic values such as supremum and Plotkin's "parallel or" cannot be definable. There is an unenlightening fully abstract extension $\mathcal{L}_A(\texttt{approx})$, *based on Gödel numbering and syntactic analysis. Unfortunately, this is the best we can do; operators defined by PCF-style rules cannot give a fully abstract language. (There is a natural and desirable property,* operational extensionality, *which prevents full abstraction with respect to* $\diamondsuit$.*) However, we show that Plotkin's program can be carried out for a non-confluent evaluator.*

*This research was supported by an NSF Fellowship, also NSF Grant No. 8511190–DCR and ONR grant No. N00014–83–K–0125.

When proposing a denotational semantics for an existing programming language, one should ask how closely the denotations of constructs agree with their computational behavior. Plotkin, in his seminal paper *LCF Considered as a Programming Language* [Plo77,Saz76], gave the paradigmatic treatment. PCF (more precisely called $\mathcal{L}_A$, and less precisely called LCF), is a typed λ-calculus with integers and Booleans, and with enough arithmetic, logical, and fixed-point operators to give it full computing power.

In general in semantics, we are trying to explain the behavior of computer programs. Following the λ-calculus community, we restrict attention to programs which read input and then produce output, such as sorting routines and TEX. We formalize programs as terms of the simply typed λ-calculus. Programs before linking correspond to open terms; complete, executable programs correspond to closed terms. Programs together with their inputs match *closed ground terms*; and outputs correspond to numerals.[1]

A (typed λ-calculus) language $\mathcal{L}$ is a set of typed constants, and a set of rules for evaluating terms built using them; details will be given later as necessary. An $\mathcal{L}$-term is a term of the typed λ-calculus built using the constants of $\mathcal{L}$.

One often wishes to reason about the behavior of terms other than closed ground terms. A basic question to ask is, "when are two pieces of code the same?" The fundamental answer in Computer Science is, "Two pieces of code are the same iff they are interchangeable; *i.e.*, if either can be substituted for the other in any program and no difference can be observed." This definition has two free parameters: the language $\mathcal{L}$ in which the programs are being used, and the aspects of program behavior we consider important.

To build a theory, we choose a set $\mathcal{O}$ of predicates on $\mathcal{L}$-terms, called *observations*, which we consider relevant. Observations should be at worst semidecidable (recursively enumerable), or we will have trouble observing them on our computer system. For our purposes, it suffices to only observe closed ground terms. Two closed ground terms M and M' are *observationally similar*, written $M \equiv_{obs} M'$, if they agree on all observations. Two arbitrary terms M and M' are *observationally congruent* with respect to $\mathcal{L}$, $M \equiv^{\mathcal{L}}_{obs} M'$, if they are observationally similar in all contexts of $\mathcal{L}$ which drive them to closed ground terms.

We, like Plotkin, will use the notions

$$\mathcal{O}_{\mathrm{num}} = \{\text{"Evaluates to } c\text{"} \mid c \text{ is a numeral}\}$$

[1] We call the Boolean values tt and ff numerals for ease of discourse; the numerals are precisely the ground constants.

as observations; we make the observations by running the program until it returns a value, if ever. By the Church-Rosser theorem (which holds for many dialects of the λ-calculus), and the fact that the normal forms of closed ground terms are numerals, it is worthwhile to define the function $\mathsf{Eval}(M)$ which completely evaluates closed ground terms. With $\mathcal{O}_{\mathrm{num}}$ as the notion of observation, $M \equiv_{obs} M'$ iff M and M' evaluate to the same numeral or both diverge; that is, if $\mathsf{Eval}(M) = \mathsf{Eval}(N)$.

The operational theory of PCF and similar languages is quite rich. One important property is *operational extensionality* (also called *the context lemma*): if two functions agree on all *definable* inputs, then they are congruent. Intuitively, in an operationally extensional language, all the interpreter does to functions is to pass them around and apply them to arguments. PCF is operationally extensional; we will see an example of a language which is not operationally extensional below [Mul86,BCL85,Mil77].

This definition is operational: it depends on the way that the computer evaluates programs. Another, more abstract way to describe the behavior of programs and terms is a denotational semantics, a mapping $[\![\cdot]\!]$ into some mathematical space of meanings. Preferably, the space will have some comprehensible structure; *e.g.*, the meaning of a term of functional type ought to be a function, or something with functional behavior. If we are to get anything useful from the denotational semantics, it should have some connection with the operational semantics. A minimal desirable property is computational adequacy, that closed ground terms should evaluate the way that the semantics say they should; we restrict this discussion to adequate semantics.

A computationally adequate semantics is appropriate for reasoning about the behavior of code together with its input. If we can prove that two terms (of any type, not necessarily closed) have the same denotation, then they are necessarily interchangeable. The converse is generally false; it is quite common for programs to have different denotations, yet behave the same under all circumstances. That is, an adequate semantics allows us to prove things about program *equality*, but not *difference*. The semantics are a sound model for the language, but not a complete one. When the converse holds, the language is *fully abstract*.

Denotational semantics allows us to ask other questions as well. When designing a programming language, one would like to be able to call it "universal" in some precise and useful sense. Turing universality is not a particularly useful notion here; most languages compute all partial recursive functions on integers. One must look further to get a useful notion of universality. Scott-style denotational semantics frequently allows one to define the notion of a computable semantic value. A language

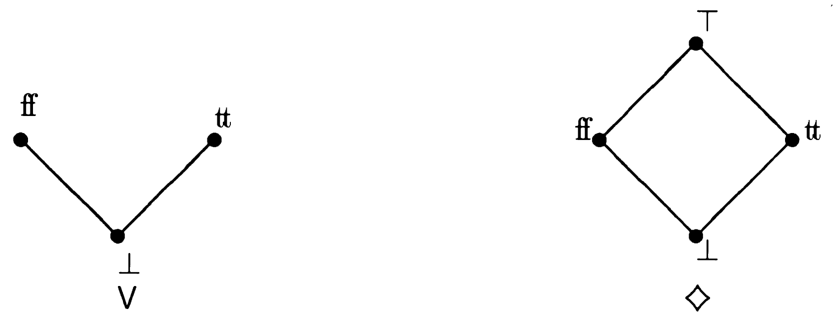

Figure 1: Domains of Booleans

is *denotationally universal* for a model if all computable values are definable.

Turing completeness implies that the language allows us to compute any first order function that we could reasonably expect to compute. Denotational universality implies that we have all functions and control structures that we could reasonably hope to have, relative to the semantics.

Plotkin, [Plo77], investigates the questions of adequacy, full abstraction and universality for PCF. The denotational semantics V is the type frame built with flat cpo's at base type and cpo-continuous function spaces (Figure 1). The base language $\mathcal{L}_A$ is already adequate for V, but not fully abstract. Adjusting the denotational spaces to achieve full abstraction is a difficult open problem [BCL85] or involves a model involving a good deal of syntactic information [Mul85,Mil77]. It is possible to keep the spaces fixed and adjust the language; $\mathcal{L}_A(\mathsf{pcond})$, which is $\mathcal{L}_A$ with a parallel conditional operator added, is fully abstract.[2] The term $\mathsf{pcond}\ B\ \mathsf{phen}\ M\ \mathsf{plse}\ N$ evaluates all its arguments simultaneously. If B returns true or false, $\mathsf{pcond}\ B\ \mathsf{phen}\ M\ \mathsf{plse}\ N$ evaluates to M or N respectively. However, if M and N evaluate to the same numeral c, then $\mathsf{pcond}\ B\ \mathsf{phen}\ M\ \mathsf{plse}\ N$ evaluates to c as well – even if B diverges. Finally, $\mathcal{L}_A(\mathsf{pcond} + \exists)$, which is $\mathcal{L}_A(\mathsf{pcond})$ with the addition of a continuous approximation of the existential quantifier is denotationally universal. These results are summarized in Figure 2.

Plotkin mentions two other models, V_∞ and $\Diamond$. Both of these are computationally adequate for $\mathcal{L}_A$, but neither fully abstract nor universal. The addition of

[2] One use of denotational semantics is to guide language design. In this case, the semantics suggests the addition of programming constructs to the language.

Computational Adequacy	$\forall$termsM, M' $[\![M]\!] = [\![M']\!]$ implies $M \equiv^{\mathcal{L}}_{obs} M'$ **Equivalently,** $\forall$closed ground terms M, M' $[\![M]\!] = [\![M']\!]$ iff $M \equiv_{obs} M'$	all three
Full Abstraction	$\forall$terms M, M' $[\![M]\!] = [\![M']\!]$ iff $M \equiv^{\mathcal{L}}_{obs} M'$	$\mathcal{L}_A(\mathsf{pcond})$
Denotational Universality	$\forall$computable$f$ $\exists F. [\![F]\!] = f$	$\mathcal{L}_A(\mathsf{pcond} + \exists)$
Operational Extensionality	$\left(\forall \vec{N}. M\vec{N} \equiv_{obs} M'\vec{N}\right)$ implies $M \equiv^{\mathcal{L}}_{obs} M'$	all three

Figure 2: Summary of Plotkin's Results

parallel conditional does not change the situation for either model.

V_∞, which is V with an extra integer ∞ incomparable to all proper integers, is not particularly different from V. Adding a constant denoting ∞ and a test for equality to ∞ to $\mathcal{L}_A(\mathsf{pcond})$ gives a language fully abstract for V_∞. Adding $\exists$ to that language makes it denotationally universal. These results are predictable from Plotkin's paper; the proofs are trivial modifications of those in the paper. Similar results should hold routinely for arbitrary effectively presented flat domains with suitable effective predicates and functions.

$\diamondsuit$ is more interesting, mathematically and historically. It has *flat lattices* (Figure 1) at ground type, and cpo-continuous function spaces at higher type. It is mathematically more tractable than V, because the semantic domains at all types are lattices rather than merely cpo's; all sets of values have suprema. This is quite helpful when doing mathematics: our proofs of full abstraction and universality are significantly simpler than Plotkin's proofs for just this reason.

Despite these apparent advantages, the semantics community has moved toward the use of cpo models rather than lattices. This paper gives formal support to this movement, showing that in at least one context the lattice models are less appropriate than cpo models: we will show that Plotkin's program cannot be carried out in $\diamondsuit$ as cleanly as it can in V. First, we define a minimal requirement for typed λ-calculi. Recall that PCF only does simple arithmetic; it may be thought of as a higher-order desk calculator. It would be very disconcerting for PCF programs to have the possibility of returning more than one result. The essence of PCF is single-valued; and we consider multi-valued variants of PCF undesirable.

In certain places, the *evaluation* is not deterministic, in that there are several ways to evaluate a term. However, PCF is confluent (has the Church-Rosser property); if it is possible for a term of any type to reduce in two ways, those ways can be brought back together again. In particular this holds for closed ground terms, and so PCF is single-valued. (Note that single-valuedness is weaker than confluence, as higher-order terms may not conflue.)

Having chosen single-valuedness as an essential criterion for PCF-like languages, we immediately find a flaw with the lattice model (Section 3). No single-valued extension of $\mathcal{L}_A$ is denotationally universal for $\diamondsuit$.

Denotational universality is perhaps the least important of our criteria; we might not be too unhappy with a merely fully abstract language. This is readily available by a very general construction; we call the resulting language $\mathcal{L}_A(\texttt{approx})$. In section 4, we show $\mathcal{L}_A(\texttt{approx})$ is fully abstract with respect to $\diamondsuit$.

On further consideration, $\mathcal{L}_A(\texttt{approx})$ is unpleasant; it is a ruthless combination of Gödel numbering and syntactic analysis which breaks operational extensionality. In fact, we show that any language which is fully abstract is not operationally extensional. It is an instance of a general construction which makes any vaguely pliable language and domain fully abstract. It bears little resemblance to the tasteful structured operational rules which define $\mathcal{L}_A$ and Plotkin's extensions of it. Unfortunately, this cannot be repaired; for a suitable and very general definition of "PCF-like", (Sections 5 and 6): we show that there is no single-valued extension of $\mathcal{L}_A$ by PCF-like operational rules which is fully abstract for $\diamondsuit$. In particular, it is not possible to add a single-valued "parallel conditional" to PCF to make it fully abstract for $\diamondsuit$.

The impossibility results hinge on the fact that the evaluator is forced to return a single final answer in finite time, or to diverge. If we take a more liberal interpretation of how a computation returns answers, we can achieve full abstraction and denotational universality for $\diamondsuit$ even more easily than Plotkin did for V; the parallel conditional and a constant denoting $\top$ are all that is required, and the existential quantifier $\exists$ is not necessary. In Section 7, we give a nondeterminate language $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$, and show that $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ is fully abstract, denotationally universal, and operationally extensional.

The cost is high. $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ is severely non-single-valued; the constant $\top$ reduces to every numeral. It is possible to change the notion of an observation, and to have a confluent language for that notion, but in the resulting language most computations do not terminate even when they produce a value. The languages $\mathcal{L}_A(\mathtt{approx})$ and $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$ answer the theoretical questions, but are qualitatively different from usual functional languages.

In summary, this paper is the converse to [Plo77]. Plotkin showed how a well-chosen denotational semantics can match an operational semantics, and demonstrates that everything can work right. The results of this paper show that a poorly-chosen denotational semantics can fail to match any good operational semantics (although it can be tantalizingly close to matching), and demonstrates that the success of Plotkin's program is far from automatic.

2 Review of PCF and Scott Domains

The material in this section is standard; experienced readers may skip it.

Our core language, called PCF, is essentially Plotkin's core language [Plo77, Mil77]. PCF is simply typed λ-calculus. with integers and Booleans as base types, and enough constants to give it general computing power. The base or ground types are ι and o, which will be used for the integers and Booleans respectively; if σ and τ are types, then $\sigma \to \tau$ is the type of functions from σ to τ. As usual, $\to$ associates to the right: $\sigma \to \tau \to \rho$ is read as $\sigma \to (\tau \to \rho)$.

We fix disjoint, countably infinite sets of variables $\{x_i^\sigma\}$ of each type σ. Let $\mathcal{C} = \{c_i^\sigma\}$ be a set of typed constants. The terms and their typings of $\mathcal{L}(\mathcal{C})$, typed λ-calculus over $\mathcal{C}$, are given by the following rules. The phrase $M : \sigma$ means that M is a term, and it has type σ. In this system, it is straightforward to prove that each term has precisely one typing.

$$x_i^\sigma : \sigma$$

$$c_i^\sigma : \sigma$$

$$\frac{M : \sigma \to \tau, \quad N : \sigma}{(MN) : \tau}$$

$$\frac{M : \tau}{\lambda x_i^\sigma . M : \sigma \to \tau}$$

$$n : \iota \text{ for each integer } n \in \mathbf{N}.$$
$$\mathsf{tt}, \mathsf{ff} : o \qquad \text{truth and falsehood.}$$
$$^{-}\mathbf{1}+, \mathbf{1}+ : \iota \to \iota$$
$$\mathbf{0}= : \iota \to o$$
$$\mathsf{cond}_\iota : o \to \iota \to \iota \to \iota$$
$$\mathsf{cond}_o : o \to o \to o \to o$$
$$\mathbf{Y}_\sigma : (\sigma \to \sigma) \to \sigma$$

Figure 3: Constants of $\mathcal{L}_A$

The constants in the core language of PCF are given in Figure 3. We write $\mathcal{L}_A(\mathcal{C})$ for the typed λ-calculus with the core constants as well as those in $\mathcal{C}$. For convenience of notation, we use the mathematical integers and Booleans as numerals. A ground term is any term of type ι or o. The notion of free and bound variables is completely standard; a *closed term* is a term without free variables. We are, ultimately, concerned only with closed ground terms: only they, in general, can produce output of a form that we are willing to observe.[3] In this paper, equations between terms denote syntactic equality modulo renaming of bound variables; in particular the symbol "=" is not used for β-convertability.

The operational semantics is defined in the usual way. We define a *one-step reduction relation* $M \to N$ between terms, with rules given in Figure 4. We will investigate its transitive reflexive closure $M \twoheadrightarrow N$. $M[x := N]$ is the term M with N substituted for free occurrences of x, with renaming of bound variables as appropriate; see [Bar81]. We will write terms in a functional-programming style, *e.g.*

let $fxy = M(f) + 3$
  in if B then R else S

will abbreviate[4]

$$(\lambda f.(\mathsf{cond} BRS))\Big(\mathbf{Y}(\lambda f.\lambda x.\lambda y.\mathbf{1}{+}\mathbf{1}{+}\mathbf{1}{+}(Mf))\Big).$$

[3] As stated in the introduction, our primitive observations are the printing of integers: we may see that the program M evaluates to 19. Other observations are possible; see [BR89].

[4] This **let** is often written **letrec**.

$$
\begin{array}{rcl}
((\lambda x_i^\sigma.M)N) & \rightarrow & M[x := N] \\
\mathsf{cond}\ \mathsf{tt}\ \mathsf{then}\ M\ \mathsf{else}\ N & \rightarrow & M \\
\mathsf{cond}\ \mathsf{ff}\ \mathsf{then}\ M\ \mathsf{else}\ N & \rightarrow & N \\
\mathbf{1}{+}\ n & \rightarrow & n+1 \\
{}^{-}\mathbf{1}{+}\ n & \rightarrow & \begin{cases} n-1 & n>0 \\ 0 & n=0 \end{cases} \\
\mathbf{0}{=}n & \rightarrow & \begin{cases} \mathsf{tt} & n=0 \\ \mathsf{ff} & n\neq 0 \end{cases} \\
\mathbf{Y}M & \rightarrow & M(\mathbf{Y}M)
\end{array}
$$

$$
\frac{M \rightarrow M'}{(MN) \rightarrow (M'N)}
$$

$$
\frac{N \rightarrow N'}{(MN) \rightarrow (MN')}
$$

$$
\begin{array}{rcl}
\mathsf{pcond}\ \mathsf{tt}\ \mathsf{phen}\ M\ \mathsf{plse}\ N & \rightarrow & M \\
\mathsf{pcond}\ \mathsf{ff}\ \mathsf{phen}\ M\ \mathsf{plse}\ N & \rightarrow & N \\
\mathsf{pcond}\ B\ \mathsf{phen}\ c\ \mathsf{plse}\ c & \rightarrow & c
\end{array}
$$

Figure 4: Operational Rules of $\mathcal{L}_A$

Plotkin's interpreter for $\mathcal{L}_A$ is different in two ways. The first is not substantial: we define the predecessor of 0 to be 0; Plotkin does not define it. The advantage of defining it is that the only closed ground normal forms are numerals. The more important difference is that Plotkin's interpreter is determinate, while ours is not: that is, there is at most one way to reduce any given term in Plotkin's system. This difference is irrelevant for the ultimate computational behavior of terms, because our system is confluent: if $M \twoheadrightarrow M_1$ and $M \twoheadrightarrow M_2$, then there is some N such that $M_1 \twoheadrightarrow N$ and $M_2 \twoheadrightarrow N$. In fact, our system could be made determinate in such a way that all of our theorems hold, at the cost of some added complexity.

We define the *evaluator* Eval, a partial function from closed ground terms to numerals. $\mathsf{Eval}(M)$ is the unique numeral c such that $M \twoheadrightarrow c$, if it exists. By confluence, PCF is single-valued: if $M \twoheadrightarrow c$ and $M \twoheadrightarrow d$, then $c = d$. We define $M \equiv_{obs} N$ iff $\mathsf{Eval}(M) \cong \mathsf{Eval}(N)$. ($x \cong y$ if both x and y are defined and equal, or neither is defined.)

The reduction rules give behavior to terms of all types. As the problem of equality for general terms is tricky to define and undecidable to execute, we only observe *numerals*: constants of type ι and o.

Deciding when two terms, or even normal forms, of arbitrary type ought to be identified is nontrivial, and usually undecidable. Even choosing the criterion for identifying them is nontrivial. There are several common criteria.

The first notion of identity we consider is the purely operational notion of *congruence*. We first define $\mathcal{L}$-*contexts* $C = C[X_1, \ldots, X_n]$, which are simply $\mathcal{L}$-terms (which may contain the variables X_i; we capitalize these variables for visual distinctiveness). Contexts are places in which code may be inserted. $C[P_1, \ldots, P_n]$ is $C[X_1, \ldots, X_n]$ with P_i substituted for X_i, as follows:

Definition 2.1 *If $C[X_1, \ldots, X_n]$ is a context, then $C[P_1, \ldots, P_n]$ is:*

- *If $C = X_i$, then $C[P_1, \ldots, P_n] = P_i$.*
- *If C is a variable other than an X_i, or a constant, then $C[P_1, \ldots, P_n] = C$.*
- *If $C = C_1C_2$, then $C[P_1, \ldots, P_n] = C_1[P_1, \ldots, P_n]C_2[P_1, \ldots, P_n]$.*
- *If $C = \lambda x.C'$, then $C[P_1, \ldots, P_n] = \lambda x.C'[P_1, \ldots, P_n]$.*

For example, if $C[X] = \lambda x.X$, then $C[x+y] = \lambda x.(x+y)$. By contrast, $(C[X])[X := (x+y)] = \lambda z.(x+y)$, as the bound variable was renamed to avoid capture.

We say that terms M and N of the same type are *congruent with respect to $\mathcal{L}$*, $M \equiv^{\mathcal{L}}_{obs} N$, iff, for all contexts C driving them to closed ground terms, $C[M] \equiv_{obs} C[N]$. It is possible, for example, to show that $\lambda x.x$ and $\lambda y.y$ are congruent in this sense.

Another notion of program equivalence comes from a denotational semantics for PCF, assigning appropriate mathematical values to all the terms of the language. We choose meanings such that both $\lambda x.x$ and $\lambda y.y$ in fact mean the identity function on some set, and therefore are identified. It is nontrivial to find such a model for PCF; in particular, it is hard to assign a meaning to the $\mathbf{Y}$ combinator that behaves properly. Fortunately, Dana Scott [Sco69] and Plotkin, among others, have found several models.

A *type frame* is a collection of sets $[\![\sigma]\!]$ indexed by types σ, such that $[\![\sigma \to \tau]\!]$ is a set of functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$. An *environment* ρ is a type-respecting mapping from variables to values. An *interpretation* $\mathcal{I}$ is a type-respecting mapping from constants to values. We may try to build a model of the typed λ-calculus by giving a family of functions $[\![\cdot]\!]$ taking terms of type σ and environments to values of type σ, satisfying:

$$
\begin{aligned}
[\![c]\!]\rho &= \mathcal{I}(c) && (1)\\
[\![x]\!]\rho &= \rho(x) && (2)\\
[\![(MN)]\!]\rho &= ([\![M]\!]\rho)([\![N]\!]\rho) && (3)\\
[\![\lambda x.M]\!]\rho &= f \text{ where } f(d) = [\![M]\!]\rho[x \mapsto d] && (4)
\end{aligned}
$$

For an arbitrary type frame, it will not always be possible to find an f satisfying (4); necessary and sufficient conditions are given in [Mey82]. We omit ρ and $\mathcal{I}$ whenever possible.

We say that a model $[\![\cdot]\!]$ is *sound* if

- $[\![\mathit{tt}]\!] \neq [\![\mathit{ff}]\!]$. This suffices to make the model nontrivial.
- If $M \to N$, then $[\![M]\!] = [\![N]\!]$.

A large number of sound models for PCF can be built using Scott domains. We sketch the basic definitions, largely to state our choices between alternative standard terminology; this material is developed in greater detail in, *e.g.*, [Sco76, Sco69,Sto77,Bar81]. If S is a set, then $S_\perp$ is the partially ordered set $S \cup \{\perp\}$ ordered by $\perp \sqsubseteq x$ for every x and no other inequalities. A function $f : C \to D$ is *monotone* iff, whenever $x \sqsubseteq y$, then $f(x) \sqsubseteq f(y)$. A nonempty set $X \subseteq D$ is *directed* iff every pair of elements of X has an upper bound in X. A partially ordered set

C is a complete partial order (cpo) iff C has a least element $\bot_C$, and each directed subset X of C has a least upper bound $\bigsqcup X$. A monotone function f between cpo's is *continuous* iff, whenever X is directed, then $f(\bigsqcup X) = \bigsqcup f(X)$. The set $C \to D$ of continuous functions from C to D is a cpo if C and D are cpo's. It is straightforward to show that, if f is continuous, then the sequence

$$\bot, f(\bot), f(f(\bot)), \ldots$$

is directed, and that its supremum is the least fixed point of f. In fact, the function taking f to its least fixed point is itself a continuous function, allowing us to interpret $\mathbf{Y}$.

2.1 Denotational Semantics

Definition 2.2 *A language $\mathcal{L}$ with operational and denotational semantics is* adequate *if, whenever M and N are terms with $[\![M]\!] = [\![N]\!]$, then $M \equiv^{\mathcal{L}}_{obs} N$.*

For our languages, there is a useful alternate characterization of adequacy. This depends on the denotational semantics being *compositional*; that is, the meaning of a composite term is a function of the meanings of the subterms. Models defined by (1)-(4), including all models used in this paper, are necessarily compositional.

Fact 2.3 *In a compositional model, observing numerals, adequacy is equivalent to the statement that, for all closed ground terms M and N, $M \equiv_{obs} N$ iff $[\![M]\!] = [\![N]\!]$.*

The converse to adequacy is in general false; the case in which it holds deserves a name.

Definition 2.4 *A language $\mathcal{L}$ is* fully abstract *(with respect to a given denotational semantics $[\![\cdot]\!]$) if, whenever M and N are arbitrary $\mathcal{L}$-terms, $M \equiv^{\mathcal{L}}_{obs} N$ iff $[\![M]\!] = [\![N]\!]$.*

An essential feature of Scott domains for PCF is the existence of a countable set of *isolated* values, also called finite, compact, and basic values:

Definition 2.5 *The value $e \in D$ is* isolated *if, whenever $\bigsqcup X \sqsupseteq e$ for a nonempty directed set X, then there is some element $x \in X$ such that $x \sqsupseteq e$.*

We remind the reader of the following standard facts of domain theory.

Definition 2.6 *If $d \in D$ and $e \in E$, then the* step function $d\searrow e \in D \to E$ *is given by:*

$$(d\searrow e)d' = \begin{cases} e & d' \sqsupseteq d \\ \perp & \textit{otherwise} \end{cases}$$

Fact 2.7 1. *Every element of D is the supremum of its isolated approximants:*

$$x = \bigsqcup\{e : e \textit{ isolated}.\ e \sqsubseteq x\}$$

2. *Two elements x, y are different iff there is some isolated element approximating one of x, y but not the other.*

3. *The isolated elements of $D \to E$ have the form $\bigsqcup X$, where X is a finite set of compatible step functions $d\searrow e$ with d and e isolated. (Two terms are compatible if they have a supremum.)*

4. *In the domains used as models of PCF in this paper, the isolated elements have a Gödel numbering e_n^σ at each type σ, such that application, approximation, compatibility, supremum, and infimum are effective in the indices of the elements.*

We are now ready to define the models V and $\diamondsuit$. By analogy with $S_\perp$, we write $S_\perp^\top$ for $S \cup \{\perp, \top\}$, ordered $\perp \sqsubseteq x \sqsubseteq \top$ for all x: the *flat lattice on S*. We refer to $\diamondsuit$ as the *flat lattice model* despite the fact that only the base domains are actually flat; however, all domains are complete lattices. The meanings of constants are given in Figure 5.

$\mathsf{V}[\![o]\!]$	$=$	$\{\mathrm{tt}, \mathrm{ff}\}_\perp$	$\diamondsuit[\![o]\!]$	$=$	$\{\mathrm{tt}, \mathrm{ff}\}_\perp^\top$
$\mathsf{V}[\![\iota]\!]$	$=$	$\{0, 1, 2, \ldots\}_\perp$	$\diamondsuit[\![\iota]\!]$	$=$	$\{0, 1, 2, \ldots\}_\perp^\top$
$\mathsf{V}[\![\sigma \to \tau]\!]$	$=$	$\mathsf{V}[\![\sigma]\!] \to \mathsf{V}[\![\tau]\!]$	$\diamondsuit[\![\sigma \to \tau]\!]$	$=$	$\diamondsuit[\![\sigma]\!] \to \diamondsuit[\![\tau]\!]$

We say that a language $\mathcal{L}$ is *fully abstract* for a model $[\![\cdot]\!]$ if $M \equiv_{obs}^{\mathcal{L}} N$ iff $[\![M]\!] = [\![N]\!]$. Full abstraction is a very intimate connection between a language and its semantics, and is usually hard to achieve.

Definition 2.8 *An element $v \in D$ is* computable *iff $\{n : e_n \sqsubseteq v\}$ is recursively enumerable.*

A denotational semantics is denotationally universal *iff every computable element in any of its semantic domains is definable: whenever $v \in [\![\sigma]\!]$ is computable, then there is some term $M_v : \sigma$ with $[\![M_v]\!] = v$.*

Plotkin defines the constant $\exists$, which is the best continuous approximation of an existential quantifier:

$$[\![\exists]\!]f = \begin{cases} \mathrm{tt} & f(n) = \mathrm{tt} \text{ for some } n \\ \mathrm{ff} & f(\bot) = \mathrm{ff} \\ \bot & \text{otherwise} \end{cases}$$

Theorem 2.9 *([Plo77,Saz76])*

- $\mathcal{L}_A$ *is adequate for the models* V *and* $\diamondsuit$.
- $\mathcal{L}_A$ *is not fully abstract for either* V *or* $\diamondsuit$.
- $\mathcal{L}_A(\mathsf{pcond})$ *is fully abstract for* V, *and not for* $\diamondsuit$.
- $\mathcal{L}_A(\mathsf{pcond})$ *is not universal for either* V *or* $\diamondsuit$.
- $\mathcal{L}_A(\mathsf{pcond} + \exists)$ *is universal for* V, *and not for* $\diamondsuit$.

3 Failure of Denotational Universality

One of the primary mathematical advantages of $\diamondsuit$ over V is that $\diamondsuit$ is a complete lattice; all sets of elements, and in particular all pairs of elements, have suprema. However, this is not true operationally: we will show that the curried supremum function $\sqcup : o \to o \to o$ is not definable, although it is clearly computable. Plotkin's function "parallel or" **por** is also not definable, with essentially the same proof. It is more striking that $\sqcup$ is not definable; the reason that $\diamondsuit$ is attractive is that suprema exist. This foreshadows the disagreement between operational and denotational semantics to come.

Let $\mathcal{L}$ be any effective extension of $\mathcal{L}_A$ by the addition of typed constants, such that there is some meaning function $[\![\cdot]\!]$ giving meanings in $\diamondsuit$ to all terms of $\mathcal{L}$, agreeing with the definitions in [Plo77], which are repeated in figure 5.

For the moment, we will consider any possible evaluation mechanism for $\mathcal{L}$. An *evaluation function for* $[\![\cdot]\!]$, Eval, is a partial recursive function from closed ground terms of $\mathcal{L}$ to numerals of $\mathcal{L}$, such that $[\![\cdot]\!]$ is a computationally adequate denotational semantics. By Fact 2.3, we know that $[\![M]\!] = \bot$ iff $\mathsf{Eval}(M)$ diverges; also, $[\![M]\!] = [\![n]\!] = n$ for some standard integer or Boolean n iff $\mathsf{Eval}(M) = n$. So, if there is an $\mathcal{L}$-term M with $[\![M]\!] = \top$, $\mathsf{Eval}(M)$ must be defined and cannot equal any standard numeral; in this case, there must be a numeral $\top$ of $\mathcal{L}$ denoting $\top$.

$$
\begin{array}{rcl}
[\![n]\!] & = & n \\
[\![\mathsf{tt}]\!] & = & \mathsf{tt} \\
[\![\mathsf{ff}]\!] & = & \mathsf{ff} \\
[\![{}^{-}\mathbf{1+}]\!]n & = & \begin{cases} \top & n = \top \\ 0 & n = 0 \\ n-1 & n > 0 \\ \bot & n = \bot \end{cases} \\
[\![\mathbf{1+}]\!]n & = & \begin{cases} \top & n = \top \\ n+1 & n \in \mathbf{N} \\ \bot & n = \bot \end{cases} \\
[\![\mathbf{0=}]\!]n & = & \begin{cases} \top & n = \top \\ \mathsf{tt} & n = 0 \\ \mathsf{ff} & n > 0 \\ \bot & n = \bot \end{cases} \\
[\![\mathsf{cond}]\!]bxy & = & \begin{cases} \top & b = \top \\ x & b = \mathsf{tt} \\ y & b = \mathsf{ff} \\ \bot & b = \bot \end{cases} \\
[\![\mathbf{Y}]\!]f & = & \bigsqcup\left\{ f^{(n)}(\bot) : n \geq 0 \right\}
\end{array}
$$

Figure 5: Meanings of Basic Terms

Theorem 3.1 *Let $\mathcal{L}$ be any language as above. There is no evaluation function* Eval *for $[\![\cdot]\!]$ for which there is an $\mathcal{L}$-term P such that $[\![P]\!] = \sqcup : o \to o \to o$.*

Proof: Suppose that P were such a term. $[\![P\ \mathsf{tt}\ \mathsf{ff}]\!] = \top$, so the language must include a numeral $\top : o$. If N is a closed Boolean term such that $\mathsf{Eval}(N)$ is defined, then $[\![N]\!] \neq \bot$. By definition, the ordinary conditional cond is doubly strict; that is, $[\![\mathsf{cond}]\!]bxy = b$ when $b = \bot$ or $b = \top$. So, computing in the domain, we discover that if $\mathsf{Eval}(N)$ is defined then $[\![\mathsf{cond}\ N\ \mathsf{then}\ \top\ \mathsf{else}\ \top]\!] = \top$, and if $\mathsf{Eval}(N)$ is undefined, then $[\![\mathsf{cond}\ N\ \mathsf{then}\ \top\ \mathsf{else}\ \top]\!] = \bot$.

Define

$$H = \lambda x.P\ \mathsf{tt}\ (\mathsf{cond}\ x\ \mathsf{then}\ \top\ \mathsf{else}\ \top). \tag{5}$$

Straightforward calculations show that, for any $N : o$,

$$\begin{array}{rcl} \mathsf{Eval}(N) \text{ converges} & \text{iff} & [\![HN]\!] = \top \\ \mathsf{Eval}(N) \text{ diverges} & \text{iff} & [\![HN]\!] = \mathsf{tt} \end{array}$$

By adequacy, $\mathsf{Eval}(HN)$ converges for any closed Boolean term N, and N diverges iff $\mathsf{Eval}(HN) = \mathsf{tt}$. The evaluator is able to solve the halting problem for $\mathcal{L}$-terms. Since $\mathcal{L}$ includes PCF, $\mathcal{L}$ has general computing power, and therefore an undecidable halting problem. So, no term P with $[\![P]\!] = \sqcup$ exists. □

We could merely have shown that the function $[\![H]\!]$ is undefinable. We will use the undefinability of $\sqcup$ in the non-full-abstraction results of Section 5.

The theorem could be stated somewhat more generally: all that is important is that Eval always halt on non-$\bot$ terms, and that the value it returns on terms meaning tt be distinguishable from the value on terms meaning $\top$. Note that this observation, though precisely analogous to Plotkin's choice of observation, is a bit peculiar: it is *non-monotonic*. It can be argued that these observations are not appropriate, that any observation which can be made about tt should also be possible to make about $\top$. In Section 7 we present such a system. The results of this paper may be interpreted as an exploration of the horrible things which can happen if non-monotone observations are allowed.

$\diamondsuit$ is mathematically appealing because suprema always exist. Unfortunately, this cannot be reflected into the programming language: even the simplest supremum function is not definable. Although both the failure of universality and the way in which it fails are disappointing, they are not enough to condemn $\diamondsuit$. Universality is the least often used of our relations between operational and denotational semantics; and we usually do not expect all mathematically-useful functions to be definable. We might be content with a fully abstract language.

4 Full Abstraction

Full abstraction seems on first glance to be a quite strong connection between an operational and denotational semantics. A more detailed look shows that the situation is complex. Full abstraction not in general a monotone relation: if $\mathcal{L}_1$ is a fully abstract language for some model, and $\mathcal{L}_2$ is a perfectly reasonable-looking extension of $\mathcal{L}_1$ which can be interpreted in that model, $\mathcal{L}_2$ may fail to be fully abstract. Full abstraction is a delicate balance between the expressive power of a language and its distinguishing power, its ability to tell its own terms apart. When operations are added to the language, both sorts of power increase, and the balance may be lost. There are at least three ways a language may be fully abstract for a Scott model:

Expressively: It could have enough power to *define* all isolated elements in the model. For example, PCF with pcond is fully abstract for V, observing numerals, in this sense: all isolated elements are definable. If two functional terms F and G have different meanings, we can give them enough arguments to drive them to have different meanings at ground type.

Introspectively: It could have enough power to *observe* all distinctions that the semantics makes; *i.e.*, tests for approximation to isolated elements are all definable, even though not all of the isolated elements themselves are. If F and G have different meanings, it is because there is some isolated value e which approximates one of them and not the other. Applying the test T_e for approximation by e to F and G will therefore give observably different results. The general construction described below gives this sort of full abstraction. [5]

Inhibitedly: It could be so weak that it cannot describe the distinctions that the semantic domains make. For example, in the simply typed pure λ-calculus (without constants), the set-theoretic model built from an infinite set is fully abstract for observing normal forms [Fri75]; although the model makes non-trivial use of higher-order functions, the language cannot define many of them and cannot test for most elements it cannot define.

Plotkin's proof of full abstraction for V showed that parallel conditional makes PCF expressively fully abstract for V; all isolated elements are definable. However,

[5]This method of distinguishing terms is of course possible for an expressively fully abstract language.

we already know that some isolated elements, such as $\sqcup$, are not definable in $\diamondsuit$, so we cannot use Plotkin's method. Adding certain constants makes PCF introspectively fully abstract for $\diamondsuit$.

Recall that two denotational values are different iff there is an isolated element e which approximates only one of the two. So, if all of the tests $e \sqsubseteq [\![M]\!]$ for isolated e and closed M were definable — *e.g.*, if there is a term $T_e : \sigma \to o$ such that $\mathsf{Eval}(T_e M) = \mathsf{tt}$ if $e \sqsubseteq [\![M]\!]$ and $T_e M$ diverges otherwise — then we could distinguish all pairs of semantically different terms. If $[\![M]\!] \neq [\![N]\!]$, then we proceed as follows. Let $x_1, \ldots, x_n$ be the variables appearing in M or N, and let $C_1[Z] = \lambda x_1, \ldots, x_n.Z$. $C_1[M]$ and $C_1[N]$ are closed terms with different meanings, and so there is an isolated e which approximates only one of the two. Let $C_2[Z] = T_e C_1[Z]$; then $C_2[Z]$ is a context distinguishing M and N. So, PCF is fully abstract if all of the T_e's are definable.

It happens that the tests can be computed by a syntactic analysis of the closed terms M without recourse to terms with value e_n^σ. We introduce constants $\mathtt{approx}^\sigma : \iota \to \sigma \to o$, with the intended meaning:

$$[\![\mathtt{approx}^\sigma]\!]\, nf = \begin{cases} \top & n = \top \\ \mathsf{tt} & e_n^\sigma \sqsubseteq f \\ \bot & \text{otherwise} \end{cases}$$

The meanings of all PCF terms are recursively enumerable elements of the semantic domains, *i.e.*, for each M, testing $e_n \sqsubseteq [\![M]\!]$ is uniformly semidecidable in n. We must show that this is semidecidable uniformly in both M and n.

The interpreter evaluates $\mathtt{approx}\ N\,M$ by the following program. It first evaluates N until it yields an integer, n. (If N evaluates to $\top$, $\mathtt{approx}\ N\,M$ returns $\top$.) If M is not closed, it then stops.

When M is closed, the evaluator converts M to combinatory form [Bar81]. If M is a constant (including the combinators **S**, **K**, **Y** and the new constant $\mathtt{approx}$), the evaluator uses a built-in rule to decide if $e_n^\sigma \sqsubseteq [\![M]\!]$. Lemma 4.1 will show that this is uniformly decidable in n and the constant M.

If the combinatory term M is not a constant, it must be an application $M_1 M_2$. In this case, $e_n^\sigma \sqsubseteq [\![M]\!]$ iff there is an integer k such that $(e_k^\tau \searrow e_n^\sigma) \sqsubseteq [\![M_1]\!]$ and $e_k^\tau \sqsubseteq [\![M_2]\!]$. So, the interpreter performs a dovetailed search over all such k, computing $\mathtt{approx}\ (s_{\sigma,\tau,k,n})\, M_1$ and $\mathtt{approx}\ k\, M_2$ (where $s_{\sigma,\tau,k,n}$ is the code of $e_k^\tau \searrow e_n^\sigma$; note that $s_{\sigma,\tau,k,n}$ is a recursive function of σ, τ, k and n.) If there is no such k, then the dovetailed search will diverge as desired; otherwise, it will return tt. It is possible to program this procedure as a set of reduction rules (Figure 6) which branch on the structure of their arguments and do some nontrivial side computations on Gödel numbers; this system is not confluent, though it is single-valued.

Lemma 4.1 *The set* $\{\langle n, M\rangle : e_n^\sigma \sqsubseteq [\![M]\!], M$ *a combinatory constant of* $\mathcal{L}_A(\texttt{approx})\}$ *is decidable.*

Proof: The following lemmas give a decision procedure for $e_n^\sigma \sqsubseteq [\![M]\!]$ where M is a constant. Recall that application, equality, and, approximation of isolated elements are recursive in the codes of the elements; Lemma 4.3 shows that the fixed point of an isolated element is computable. The lemmas reduce $e_n^\sigma \sqsubseteq [\![M]\!]$ to decidable questions. □

From the fact that $x\searrow(y\sqcup z) = (x\searrow y)\sqcup(x\searrow z)$, it follows that each isolated element can be expressed in a simple form. If $\sigma = \sigma_1 \to \sigma_2 \to \cdots \to \sigma_r$, then there are k and a_{ji} such that

$$e_n^\sigma = \bigsqcup\{a_{1i}\searrow a_{2i}\searrow \cdots \searrow a_{ri} : i = 1, \ldots, k\}$$

and furthermore, such a k and the codes of the a_{ji} are recursively computable from n. Call this an *r-representation* of e_n^σ. (In general, there will be many such representations, but this will not concern us. It is possible to compute an r-representation of e_n^σ uniformly from σ and n, which we will call *the* r-representation.)

Every type σ can be written as $\sigma_1 \to \sigma_2 \to \cdots \to \sigma_r$ with σ_r a base type, and this is the most frequently used r-representation of types. In this section, we will use some shorter representations..

Lemma 4.2 *The following are necessary and sufficient conditions for approximation to the constants of PCF, and* **S** *and* **K** *combinators.*

- $e_n^\sigma \sqsubseteq [\![\mathbf{K}]\!]$ *iff for every* $a\searrow b\searrow c$ *in the 3-representation of* e_n^σ*, we have* $c \sqsubseteq a$.
- $e_n^\sigma \sqsubseteq [\![\mathbf{S}]\!]$ *iff for every* $a\searrow b\searrow c\searrow d$ *in the 4-representation of* e_n^σ*, we have* $d \sqsubseteq ac(bc)$.
- $e_n^\sigma \sqsubseteq [\![\mathsf{cond}]\!]$ *iff for every* $a\searrow b\searrow c\searrow d$ *in the 4-representation of* e_n^σ*, one of the following holds:*
    - $a = \bot$ *and* $d = \bot$
    - $a = \mathsf{tt}$ *and* $d \sqsubseteq b$

– $a = \mathrm{ff}$ *and* $d \sqsubseteq c$

– $a = \top$ *and* d *is arbitrary.*

- $e_n^{\iota\to\iota} \sqsubseteq [\![\mathbf{1+}]\!]$ *iff for each* $a\searrow b$ *in the 2-representation of* $e_n^{\iota\to\iota}$*, we have* $b \sqsubseteq a+1$*, and similarly for the other arithmetic operators.*

- $e_n^{\iota\to\sigma\to o} \sqsubseteq [\![\mathtt{approx}^\sigma]\!]$ *iff for every* $a\searrow b\searrow c$ *in the 3-representation of* e_n^σ*, we have either* $a = \top$ *or* $a = c = \bot$ *or* $b \sqsubseteq e_a^\sigma$ *and* $c \sqsubseteq \mathrm{tt}$*.*

- *Let* $\tau = (\sigma \to \sigma) \to \sigma$*. Then* $e_n^\tau \sqsubseteq [\![\mathbf{Y}_\sigma]\!]$ *iff for each* $a\searrow b$ *in the* 2*-representation of* e_n^τ*, we have* $b \sqsubseteq [\![\mathbf{Y}]\!]a$

Proof:

We show this for **K**; the others are similar. $e_n^\sigma \sqsubseteq [\![\mathbf{K}]\!]$ iff for all x, y we have $e_n^\sigma xy \sqsubseteq [\![\mathbf{K}]\!]xy = x$. Suppose that $c \sqsubseteq a$ for each $a\searrow b\searrow c$ in the 3-representation $\mathcal{T}$ of e_n^σ; then

$$\begin{aligned} e_n^\sigma xy &= \bigsqcup\{c : a \sqsubseteq x \text{ and } b \sqsubseteq y \text{ and } a\searrow b\searrow c \in \mathcal{T}\} \\ &\sqsubseteq \bigsqcup\{a : a \sqsubseteq x \text{ and } b \sqsubseteq y \text{ and } a\searrow b\searrow c \in \mathcal{T}\} \\ &\sqsubseteq \bigsqcup\{a : a \sqsubseteq x\} \\ &= x \end{aligned}$$

Conversely, if $a\searrow b\searrow c \sqsubseteq e_n^\sigma \sqsubseteq [\![\mathbf{K}]\!]$, then

$$(a\searrow b\searrow c)\, a\, b = c \sqsubseteq [\![\mathbf{K}]\!]\, a\, b = a$$

as desired. □

Lemma 4.3 *Least fixed points of isolated elements are computable: given an integer* n *and a type* σ*, it is possible to compute an integer* n' *such that* $e_{n'}^\sigma$ *is the least fixed point of* $e_n^{\sigma\to\sigma}$*.*

Proof:

Let $e = e_n^{\sigma\to\sigma} = \bigsqcup\{a_i\searrow b_i : i = 1, \ldots, k\}$ be the 2-representation of the isolated element. By general domain theory, e has a least fixed point $[\![\mathbf{Y}]\!]e$. Since $[\![\mathbf{Y}]\!]e = e([\![\mathbf{Y}]\!]e)$, the value $[\![\mathbf{Y}]\!]e$ is in the range of the function e. However, e is the supremum of a finite set of step functions, and there are only a finite number of values it can take; the range of e is at most $\{\bigsqcup\{b_j : j \in J\} : J \subseteq \{1, \ldots, k\}\}$. So, we can simply

$$
\begin{array}{rcl}
\texttt{approx}^{\sigma}\, n\,(MN) & \rightarrow & \textsf{cond}\,(\texttt{approx}^{\tau\rightarrow\sigma} s_{\sigma,\tau,k,n} M)\ \textsf{then}\ \texttt{approx}^{\tau} k N\ \textsf{else}\ \Omega \\
 & & \text{one such rule for each } k \\
\texttt{approx}^{\sigma}\, n\, \mathbf{S} & \rightarrow & f_{\sigma,\mathbf{S}} n \\
\texttt{approx}^{\sigma}\, n\, \mathbf{K} & \rightarrow & f_{\sigma,\mathbf{K}} n \\
\texttt{approx}^{\sigma}\, n\, \mathbf{Y} & \rightarrow & f_{\sigma,\mathbf{Y}} n \\
 & \vdots &
\end{array}
$$

where $f_{\sigma,\delta}$ is a term of type $\iota \rightarrow \iota$ which returns tt if $e_n^{\sigma} \sqsubseteq [\![\delta]\!]$.

Figure 6: Rewrite Rules for approx

search this finite set for the least value b such that $b = eb$; and this is the least fixed point. □

The construction used to generate $\mathcal{L}_A$(approx) is fairly independent of the constants actually available in $\mathcal{L}$, as long as the set of isolated approximants of the constants is always uniformly recursively enumerable. We refrain from giving precise necessary and sufficient conditions under which the approx construction is applicable, but it is a very general construction. The same constants approx, changing only the details of the coding of isolated elements, will be fully abstract for V and V_∞. It fulfills our general technical requirements, but it is neither informative nor satisfying.

5 No PCF-like Fully Abstract Extension

It is not possible to add informative and satisfying operators to PCF to make it fully abstract for ◇. We formalize the notion of "informative and satisfying" in two ways. Our formalizations are generous: many things will be formally acceptable but still be neither informative nor satisfying. This makes the theorems stronger; they cover languages which are tolerable only formally.

The first formalization is operational extensionality. Extensionality of mathematical functions is the statement that, if $\forall x. fx = gx$, then $f = g$. Operational extensionality is the analogous property of terms; we give the full definition below. Note that $\mathcal{L}_A$(approx) is not operationally extensional; the terms M_t and M_f below behave the same on all arguments, but differ in the context $C[X] = \mathtt{approx}\, n X$ where n is a code of the isolated element $[\![M_t]\!]$.

The second formalization, in Section 6, is a way of saying that the language has a nice interpreter. PCF is defined by clean-looking structured operational rules; we generalize the form of those rules. We will show that any language defined by "PCF-like" rules is operationally extensional. This adds weight to the claim that "PCF-like" languages are not unpleasant; by the previous paragraph, it also shows that that they cannot be fully abstract for the flat lattice model.

Definition 5.1 *Two closed terms of the same type are* applicatively congruent, $M \equiv_{ap} M'$, *if whenever* $\vec{N}$ *is a vector of enough closed arguments to drive* M *and* M' *to ground type, then* $M\vec{N} \equiv_{obs} M'\vec{N}$.

A language $\mathcal{L}$ *is* operationally extensional *if all applicatively congruent terms are observationally congruent.*

Intuitively, a language is operationally extensional if the only thing it can do to a function is to apply it to a term, and pass it around as an argument. Any language with a universal and adequate semantics is operationally extensional; in Section 6, we will show that any PCF-like language, universal or not, is operationally extensional.

Theorem 5.2 *If* $\mathcal{L}$ *is an operationally extensional extension of* $\mathcal{L}_A$ *with an r.e. evaluator, then* $\mathcal{L}$ *is not fully abstract with respect to* $\diamondsuit$.

Proof: Recall that $\sqcup$ is not definable in any extension of $\mathcal{L}_A$ with an r.e. evaluator. We construct terms M_t and M_f of type $(o \to o \to o) \to o$ such that

$$\begin{array}{rcll} [\![M_t]\!]x & = & [\![M_f]\!]x & \quad x \neq \sqcup \\ [\![M_t]\!]\sqcup & = & \mathtt{tt} & \\ [\![M_f]\!]\sqcup & = & \mathtt{ff} & \end{array}$$

M_t and M_f check their argument at all the places where $\sqcup$ is $\mathtt{tt}$ or $\mathtt{ff}$: $x\bot\mathtt{tt}$, $x\mathtt{tt}\bot$, and $x\mathtt{tt}\mathtt{tt}$ should all evaluate to $\mathtt{tt}$, and similarly for $\mathtt{ff}$. Any monotone function x which agrees with $\sqcup$ at all these places must be equal to $\sqcup$; for example, $x\bot\bot \sqsubseteq x\bot\mathtt{tt} = \mathtt{tt}$ and $x\bot\bot \sqsubseteq x\bot\mathtt{ff} = \bot$, hence $x\bot\bot = \bot$. If the argument passes these six tests, then M_t returns $\mathtt{tt}$ and M_f returns $\mathtt{ff}$. If it fails any of them, then M_t and M_f diverge or return $\top$ together. This is similar to Plotkin's construction in [Plo77].

M_t and M_f agree on all definable arguments, and by operational extensionality are congruent. However, they have different denotations. Therefore, the semantics is not fully abstract. □

6 Rules and Operational Extensionality

Operational extensionality is a reasonable-sounding property, but it is not instantly obvious that a given language — even one with a reasonable-sounding definition — will be operationally extensional. In this section, we give rather generous conditions on the form of the language definition which guarantee operational extensionality.

LCF is defined by clean-looking rules. We shall see that the very form of the rules guarantees that LCF, and a large class of extensions, are operationally extensional. We start by discussing a relatively simple and familiar form of rule, adequate for all the operators of Plotkin's paper except $\exists$. We then consider a more powerful and less standard type of system, an *observation calculus*, in which $\exists$ can be defined. We show that observation calculi are operationally extensional, and that they are conservative extensions of simple LCF-like systems; so the simple and more familiar systems are operationally extensional as well.

The rewrite rules of Figure 6 almost fit our format, but by the theorems of this section must depart from it. Indeed, the first rule (the reduction starting from $\mathtt{approx}^{\sigma}n(MN)$) looks too deeply at the structure of its arguments; it determines that the second argument is an application. This is enough to violate operational extensionality.

6.1 Simple LCF-like Systems

A rule in a programming language is essentially the same as an axiom (or inference rule) scheme in logic. This is not the place for giving a precise definition, but some terminology will be helpful. A rule may have some typed *place-holding variables*; the intent of the rule is that every well-formed instantiation of those variables by terms shall be an axiom.

The following definition is adequate for all the rules of LCF, including pcond but not $\exists$.

Definition 6.1 *A* simple LCF-like rule ρ for the constant δ *is a rule of the form:*

$$\delta X_1 \dots X_n \to P[\vec{x} := \vec{X}]$$

satisfying the following conditions. Each X_i is either a place-holding variable or a numeral. The X_i's which are place-holders must be distinct. P is a term with $\mathrm{FV}(P) \subseteq \{x_1, \ldots, x_n\}$.

We define $I = I_\rho$ to be the set of indices i such that X_i is a numeral.

For example, the rule

$$\mathsf{cond}\ \mathsf{tt}\ \mathsf{then}\ M\ \mathsf{else}\ N \rightarrow M$$

is a simple LCF-like rule for cond, with $X_1 = \mathsf{tt}$ and X_2 and X_3 placeholding variables M and N. See Figure 4 for examples; note that rule β is not a simple LCF-like rule, but the $\mathbf{Y}$-rule is.

We say that a term M *satisfies* ρ if $M = \delta\vec{M}$ and $M_i = c_i$ for each $i \in I_\rho$. For example, pcond tt phen tt plse tt satisfies two of the rules for pcond. The rule ρ is said to *define* δ despite the fact that there will usually be several rules defining the same δ.

We allow tests for equality to constants because we started by considering constants observable; we do not allow other sorts of tests, because we do not consider them observable. Most reduction rules proposed for LCF in the literature are simple; *e.g.*, all first-order functions (taking only arguments of ground type) are definable by simple rules. Some, such as $\mathbf{1}+$, may require infinite sets of rules; in most circumstances, infinite *recursive* sets are preferred.

To illustrate the restrictions, consider the rules:

$$\begin{aligned} \delta_1 XX &\rightarrow \mathsf{tt} \\ \delta_2(\lambda x.Z) &\rightarrow \mathsf{tt} \end{aligned}$$

where $X : \iota \rightarrow \iota$. δ_1 gives us a test for syntactic equality, and so we may distinguish between $\lambda x.x$ and $\lambda x.{}^{-}\mathbf{1}{+}\mathbf{1}{+}x$, which should be equal. δ_2 allows us to see if a term reduces to an abstraction; ${}^{-}\mathbf{1}+$ and $\lambda x.{}^{-}\mathbf{1}{+}x$ differ under the δ_2 operation. It is possible that a system including rules like δ_1 or δ_2 could behave sensibly, *e.g.*, have V or something else reasonable as a model; see [MC88] and [BR89] for such systems. We do not claim that such systems are senseless; however, their sensibility is a nontrivial theorem (typically adequacy) rather than a consequence of the form of the rules via the metatheory of this section.

We use substitution for variables $P[\vec{x} := \vec{X}]$ rather than substitution in a context $C[\vec{X}]$ to be sure that variables occurring in the X's are treated correctly. It is not clear that this is necessary, given our choice of observations and reduction rules:

open terms cannot be directly observed, and we do not reduce inside of λ-bindings. There is no difference for closed terms X, as such terms have no free variables to be accidentally captured. However, one is occasionally interested in the derived equational theory, interpreting a reduction rule $M \rightarrow N$ as an equation $M = N$ and using ordinary equational reasoning. The operation $\delta_3 X \rightarrow \lambda y.X$ can capture a free y in X; the equational theory of δ_3 is inconsistent. Adding a side condition to fix it leaves the simple LCF-like framework, and is tantamount to using $P[x := X]$.

We could prove operational extensionality directly for languages defined by simple rules. Since we need the same theorems for a more powerful sort of language, we will prove them only for that language. The argument that operational extensionality for simple rules follows from that of general rules requires several theorems arguing that the general rules are still reasonable.

6.2 Observation Calculi and General Rules

We give a more general form of rule, powerful enough to express $\exists$. Plotkin's rules for $\exists$ are these:

$$\frac{fn \twoheadrightarrow \mathrm{tt}}{\exists f \rightarrow \mathrm{tt}}$$

$$\frac{f\Omega \twoheadrightarrow \mathrm{ff}}{\exists f \rightarrow \mathrm{ff}}$$

Notice that these rules involve $\twoheadrightarrow$ as well as simply $\rightarrow$.

In this study of LCF, our only observation is $M \twoheadrightarrow c$ for closed ground terms M. However, our reduction rules give much more behavior to far more terms: the forking and joining reduction paths of open higher-order terms are visible in the calculus, and it is not clear why we are ignoring all of this behavior. In fact, we can pay some attention to it and get a coherent theory; see for example [BR89].

We propose an alternate form of LCF and operational semantics in general, called *observation calculi*, so called because they do neither more nor less than calculate observable facts about terms. By contrast, the ordinary typed λ-calculus computes behavior of terms which we do not want to observe. We do care that $(\lambda f, x.f(fx))(\mathbf{1}+)(\mathbf{1}+3)$ evaluates to 6; we do not care that the reduction graph (as PCF is presented in this paper) of this term forks three times, nor that all the paths from start to end are length 5.

In this paper, we will describe the observation calculi relevant to LCF. Other authors have used similar systems [Plo88]. It is straightforward to give a observation calculus for other sorts of observations; the lazy λ-calculus of Abramsky [Abr] and Ong [Ong88] is an observation calculus. Observation calculi will be described further in later work, and their virtues and vices examined. In this paper, it suffices that the calculi described here are a conservative (with respect to our choice of observation) and proper extension of simple rules, and that they are operationally extensional.

The basic judgment of a LCF-like observation calculus is that a closed ground term M evaluates to the numeral c; we write this $M \looparrowright c$. The rules will be written in a style in which only this sort of judgment appears; see Figure 7 for the observation-calculus versions of some LCF rules. This definition is intended only for observing constants; a more general definition will appear elsewhere.

Definition 6.2 *A* observation-calculus δ-rule *is a rule θ of the form*

$$\frac{C_i[\vec{X}] \looparrowright c_i \text{ for } i \in I}{\delta\vec{X} \looparrowright c} \tag{6}$$

where δ is a constant, $\vec{X}$ is a vector of distinct typed placeholding variables such that $\delta\vec{X}$ is of ground type; each $C_i[\cdot]$ is a typed λ-calculus context such that $C_i[\vec{X}]$ is of ground type, and c and c_i are numerals. There must be at least one antecedent $C_i[\vec{X}] \looparrowright c_i$ unless δ is a numeral.

An observation calculus $\mathcal{F}$ *consists of a set of typed constants, and a set of observation-calculus rules over those constants. That is, each δ and C_i is a constant or context over the set of operators. We say that "M is a term of $\mathcal{F}$" if M is a term of typed λ-calculus over $\mathcal{F}$'s constants.*

In designing languages, or classes of observation calculi suitable for programming languages, it may be desirable to impose some restrictions on the structure of an observation calculus. For example, it may be desirable that the set of rules applying to a given term be recursive, or that the number of antecedents of a given rule be finite, or that the set of rules be consistent in that they give a single-valued evaluator. For the moment, we will not impose any such restrictions.

The observation calculus version of the β-rule

$$\frac{M[x := N]\vec{A} \looparrowright d}{(\lambda x.M)N\vec{A} \looparrowright d} \tag{7}$$

deserves discussion. First, note that it is a rule scheme with an instance for each constant d as well as for each M and N. Like the λ-calculus, there are neither

$$d \Downarrow d$$

$$\frac{B \Downarrow \mathsf{tt}, \quad M \Downarrow d}{\mathsf{cond}\ B\ \mathsf{then}\ M\ \mathsf{else}\ N \Downarrow d}$$

$$\frac{M(\mathbf{Y}M)\vec{A} \Downarrow d}{\mathbf{Y}M\vec{A} \Downarrow d}$$

$$\frac{M \Downarrow d}{\mathbf{1}{+}M \Downarrow (d+1)}$$

$$\frac{M \Downarrow d, \quad N \Downarrow d}{\mathsf{pcond}\ B\ \mathsf{phen}\ M\ \mathsf{plse}\ N \Downarrow d}$$

Figure 7: Typical Observation Calculus Rules for PCF

substition operators or metavariables ranging over numerals within the language. Second, note that we cannot simply say that $(\lambda x.M)N$ evaluates to $M[x := N]$; the notion of "evaluates to" is only defined for ground terms, and even ground terms can only evaluate to numerals. For convenience, we build this rule into the definition of an observation calculus.

There are no operator and operand evaluation rules. If we need to evaluate something, we evaluate it fully as a hypothesis of a rule. This makes the most natural interpreter — which simply searches for a proof tree — rather inefficient; if we need the value of a term twice, we evaluate it twice. However, efficiency is not a concern in this study.

An observation calculus computes by building proof trees. The definition of a proof tree is standard, and we omit the details. If π is a proof tree, we write $\pi \supset M \Downarrow c$ to indicate that the conclusion of π is the fact that $M \Downarrow c$.

We define $|\pi|$ to be the depth of the proof tree π, which may in general be an infinite ordinal. The clause that there must be antecedents is a technical trick to make the only depth-0 proofs be those showing that $d \Downarrow d$.

The notions of congruence and applicative congruence can be defined quite naturally in an observation calculus.

Definition 6.3 *$M \equiv^{\mathcal{F}}_{\looparrowright} N$ iff M and N are arbitrary terms of the same type, and for each context $C[\cdot]$ we have $C[M] \looparrowright c$ iff $C[N] \looparrowright c$.*

$M \equiv^{ap,\mathcal{F}}_{\looparrowright} N$ iff M and N are closed terms of the same type, and for each vector of arguments $\vec{A}$ driving them to ground type and each numeral c, we have $M\vec{A} \looparrowright c$ iff $N\vec{A} \looparrowright c$.

$\equiv^{\mathcal{F}}_{\looparrowright}$ is the natural notion of congruence for an observation calculus; $\equiv^{ap,\mathcal{F}}_{\looparrowright}$ is the natural notion of applicative congruence. As desired, the two coincide. (*cf.* [Mil77])

Theorem 6.4 (Operational Extensionality of Observation Calculi) *If $\mathcal{F}$ is an observation calculus, then for closed terms M and N, $M \equiv^{\mathcal{F}}_{\looparrowright} N$ iff $M \equiv^{ap,\mathcal{F}}_{\looparrowright} N$.*

The proof is deferred to the appendix, Section A.

6.3 Connections between Simple and General Rules

Our mathematical justification of observation in this paper will be restricted to showing that they are a conservative extension of simple LCF-like rules, in an appropriate sense. The basic result is an adequacy theorem: $M \twoheadrightarrow c$ iff $M \looparrowright c$ for closed ground terms.

Definition 6.5 *Two simple LCF-like rules ρ_0 and ρ_1*

$$\rho_i : \delta X_1 \ldots X_{n_i} \to P_i[\vec{x} := \vec{X}]$$

are consistent *iff $n_0 = n_1$ and, whenever $\vec{M}$ satisfies both ρ_0 and ρ_1, then*

$$P_0[\vec{x} := \vec{M}] = P_1[\vec{x} := \vec{M}]$$

where equality is as usual syntactic equality up to renaming of bound variables. Rules defining different constants are always consistent. A set of rules is consistent if every pair of rules is consistent.

For example, $\mathsf{pcond}\ \mathsf{tt}\ \mathsf{phen}\ \mathsf{tt}\ \mathsf{plse}\ \mathsf{tt}$ matches both the rules $\mathsf{pcond}\ \mathsf{tt}\ \mathsf{phen}\ M\ \mathsf{plse}\ N \to M$ and $\mathsf{pcond}\ B\ \mathsf{phen}\ \mathsf{tt}\ \mathsf{plse}\ \mathsf{tt} \to \mathsf{tt}$. In both cases, the result is tt as required. It is no surprise that these rules are consistent.

For the remainder of this section, let $\mathcal{L}$ be typed λ-calculus over a set $\mathcal{C}$ of constants with operational rules given by a consistent set of simple LCF-like rules. (PCF itself is such a system.) $\mathcal{L}$'s operational semantics are clear: $\mathcal{L} \vdash M \to N$ iff there is a proof of $M \to N$ from the rules of $\mathcal{L}$, the β-rule, and the rules of operator and operand evaluation. As we are not doing much proof theory of $\mathcal{L}$, we will not formalize these proofs.

$\mathcal{L}!$ will be the corresponding observation calculus. $\mathcal{L}!$ has a rule scheme $\rho!$ corresponding each rule ρ of $\mathcal{L}$. If

$$\rho : \delta\vec{X} \to P[\vec{x} := \vec{X}]$$

(where $X_i = c_i$ for $i \in I$ and X_i a placeholding variable otherwise) then for each numeral c $\mathcal{L}!$ has the rule scheme $\rho!$:

$$\frac{X_i \looparrowright c_i \text{ for each } i \in I, \quad P[\vec{x} := \vec{X}]\vec{A} \looparrowright c}{\delta\vec{X}\vec{A} \looparrowright c}$$

where $\vec{A}$ is a vector of enough placeholding arguments to drive $\delta\vec{X}$ to ground type; if $\delta\vec{X}$ is already ground type, $\vec{A}$ will be an empty vector. We will refer to instances of this scheme as $\rho!$ as well when no confusion can arise. The operator and operand evaluation rules are unnecessary in an observation calculus.

Theorem 6.6 *If M is a closed ground term of $\mathcal{L}$, then $M \twoheadrightarrow c$ iff $M \looparrowright c$ in $\mathcal{L}!$.*

Proof: An easy consequence of Lemmas B.2 and B.4; the full proof appears in appendix B. □

6.4 Summary of Syntactic Theory

Theorem 6.7

1. *If $\mathcal{F}$ is a observation calculus, $\mathcal{F}$ is operationally extensional.*
2. *If $\mathcal{L}$ is a consistent language defined by simple LCF-like δ-rules, then $\mathcal{L}$ is operationally extensional.*

Proof: The first claim is Theorem 6.4. The second follows easily from the first and Theorem 6.6. □

The point of this excursion into syntactic theory for this paper is the following:

Theorem 6.8 *No extension of LCF with an r.e. evaluator, either by a consistent set of simple LCF-like rules or an observation calculus, can be fully abstract for $\diamondsuit$.*

Proof: Immediate from Theorem 5.2. □

7 LCF Can Be Topped

The negative results in Sections 3 and 5 hinged on a recursion-theoretic argument. The evaluator had to halt and return tt on HN if N diverges, and halt and return $\top$ if N converges. We will avoid this difficulty with a non-confluent language $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$. The non-confluence is pervasive and drastic: there are terms of all types which do not conflue, and there are built-in numerals which reduce to all values at their types. It is hard even to justify calling this a functional programming language. One branch of the reduction of HN, for example, will almost immediately return tt; another will try to evaluate N, and if that evaluation ever terminates will return $\top$.

$\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ includes all of $\mathcal{L}_A$, together with constants $\top_\sigma$ and pcond_σ for each ground type σ. Predictably, $\top_\sigma$ denotes $\top$ of type σ. pcond is a parallel conditional, the most often proper-valued conditional function; its denotation is given in equation (8). The operational rules are those of $\mathcal{L}_A(\mathsf{pcond})$, together with the rules in Figure 8. It is only necessary to define pcond at ground type.

$$[\![\mathsf{pcond}]\!]bxy = \begin{cases} x \sqcap y & b = \bot \\ x & b = \mathsf{tt} \\ y & b = \mathsf{ff} \\ x \sqcup y & b = \top \end{cases} \tag{8}$$

A minor variation on Plotkin's proof for $\mathcal{L}_A$ shows that $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ is computationally adequate. A program M will reduce to all constants c with $[\![c]\!] \sqsubseteq [\![M]\!]$. As before, we can observe the fact that $M \twoheadrightarrow c$. We cannot observe the fact that $M \not\twoheadrightarrow d$, because this is not semidecidable and we insist on having a computer program as interpreter.

In most semantics for λ-calculus, $M \to N$ implies $[\![M]\!] = [\![N]\!]$. This does not hold in $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$, but a similar fact does hold, and in fact can be seen by inspection of the rules:

$$M \to N \quad \text{implies} \quad [\![M]\!] \sqsupseteq [\![N]\!]$$

Note that the multi-valuedness is of a rather restricted form: if M reduces to two distinct integers, then it reduces to all integers and $\top$ as well. There are no terms which reduce to, say, 0 and 21 without reducing to 1-20 as well, and such terms could have no meaning in $\Diamond$.

$$
\begin{array}{rcll}
\mathsf{pcond}\ \mathsf{tt}\ \mathsf{phen}\ M\ \mathsf{plse}\ N & \rightarrow & M & \\
\mathsf{pcond}\ \mathsf{ff}\ \mathsf{phen}\ M\ \mathsf{plse}\ N & \rightarrow & N & \\
\mathsf{pcond}\ M\ \mathsf{phen}\ c\ \mathsf{plse}\ c & \rightarrow & c & \\
\mathsf{pcond}\ \top\ \mathsf{phen}\ c\ \mathsf{plse}\ d & \rightarrow & \top & \text{where } c \neq d \\
\top_\sigma & \rightarrow & c_\sigma & \\
{}^{-}\mathbf{1}{+}\top & \rightarrow & \top & \\
\mathbf{1}{+}\top & \rightarrow & \top & \\
\mathbf{0}{=}\top & \rightarrow & \top & \\
\mathsf{cond}\ \top\ \mathsf{then}\ M\ \mathsf{else}\ N & \rightarrow & \top &
\end{array}
$$

c and d range over numerals.

Figure 8: Additional rules for $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$

The basic observation is still $M \twoheadrightarrow c$, which still makes sense despite the fact that $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$ is not confluent. For example, the definition of $M {\equiv_{obs}} N$ is that for all numerals c, $M \twoheadrightarrow c$ iff $N \twoheadrightarrow c$.

Theorem 7.1 $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$ *is adequate for* $\diamondsuit$. *That is,* $[\![M]\!] \sqsupseteq [\![c]\!]$ *(where* c *is a numeral and* M *a closed ground term) iff* $M \twoheadrightarrow c$.

The proof is by Tait's method, and and closely follows [Plo77]. We defer it to appendix C.

7.1 Full Abstraction and Universality

The following lemma will be useful for both full abstraction and universality. Recall that e_n^σ is an enumeration of the isolated elements of type σ.

Lemma 7.2 *For each type* σ, *there is a term* $E^\sigma : \iota \rightarrow \sigma$ *such that for all integers* n,

$$[\![E^\sigma n]\!] = e_n^\sigma$$

and a term $G^\sigma : \iota \to \sigma \to o$ *such that for all integers* n *and* $f \in [\![\sigma]\!]$

$$[\![G^\sigma]\!]nf = \begin{cases} \mathsf{tt} & f \sqsupseteq e^\sigma_n \\ \bot & \textit{otherwise} \end{cases}$$

Proof:
Let

$$\begin{aligned} M \sqcup N &= \begin{cases} \mathsf{pcond}\ \top\ \mathsf{phen}\ M\ \mathsf{plse}\ N & \text{at base type} \\ \lambda x^\sigma.((Mx)\sqcup(Nx)) & \text{at type } \sigma \to \tau \end{cases} \\ M\textbf{andthen}N &= \mathsf{cond}\ M\ \mathsf{then}\ N\ \mathsf{else}\ \Omega \end{aligned}$$

It is straightforward, given the Gödel-numbering of isolated elements, to find recursive functions L, S and T such that

$$e^{\sigma\to\tau}_n = \bigsqcup\{e^\sigma_{Sni} \searrow e^\tau_{Tni} : 1 \leq i \leq Ln\}$$

The definitions of E and G at base type are trivial. At higher type, they are almost trivial; the existence of suprema in both model and language makes this programming exercise much easier than the corresponding exercises in [Plo77]. To compute $E^{\sigma\to\tau}nx^\sigma$, take the suprema of the e^τ_{Tni} such that $x \sqsupseteq e^\sigma_{Sni}$, using G^σ to perform these tests. To compute $G^{\sigma\to\tau}nf^{\sigma\to\tau}$, take the conjunction of the answers to the questions "Is $f(e^\sigma_{Sni}) \sqsupseteq e^\tau_{Tni}$?", using G^τ and E^σ to perform these tests.

$E^{\sigma\to\tau}nx^\sigma =$
  let *loop* i =
    if $i = 0$ **then** Ω
    else
        (**if** $G^\sigma(Sni)x$ **then** $E^\tau(Tni)$ else Ω)
      $\sqcup$
        $loop(i-1)$
  in $loop(Ln)$

and

$G'^{\sigma\to\tau}n^\iota f^{\sigma\to\tau} =$
  let *loop* i =
    if $i = 0$ **then** tt
    else

$$\Big(G^{\tau}(Tni)(f(E^{\sigma}(Sni)))\Big)$$

andthen

$loop\ (i-1)$

in $loop(Ln)$

□

From this fact, the desired properties follow easily.

Theorem 7.3 $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ *is a fully abstract language for the interpretation* $\diamondsuit$.

Proof:

If M and N are terms of the same type with different meanings, let $\vec{x}$ be a list of their free variables, and $C[X] = \lambda\vec{x}.X$. $C[M]$ and $C[N]$ are closed terms of some type σ with different meanings; let e^{σ}_{n} be an isolated element approximating one and not the other. Then $C'[X] = G^{\sigma}nC[X]$ is a context which distinguishes M and N.

Conversely, let M and N have the same meaning. Induction on the structure of contexts shows that for all $C[X]$, $C[M]$ and $C[N]$ have the same meaning. By adequacy, M and N are congruent. □

Theorem 7.4 $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ *is denotationally universal for* $\diamondsuit$.

Proof:

Suppose that f is an r.e. element of type σ. There is a recursive and therefore programmable function $N : \iota \to \iota$ such that

$$f = \bigsqcup\{e^{\sigma}_{Ni} : i = 0, 1, 2, \ldots\}.$$

Let F be the term defined by the functional program

$$Fn = (E^{\sigma}(Nn)) \sqcup (F(n+1))$$

Then

$$f = [\![F0]\!]$$

□

Theorem 7.5 $\tilde{\mathcal{L}}(\mathsf{pcond}+\top)$ *is operationally extensional.*

Proof: The isolated elements are dense in the Scott topology at each type, and so two functions agree on all isolated elements iff they are equal. All isolated elements are definable. So, if M and N agree in all applicative contexts, they agree at all isolated elements, and so $[\![M]\!] = [\![N]\!]$. By adequacy, M and N are congruent. □

It is worth noting that the analog of Theorem 7.4 does not hold in $\mathcal{L}_A$; Plotkin must introduce an extra operator $\exists$ to define all computable elements. The proofs of Lemma 7.2 and Theorem 7.4 are much easier than Plotkin's proofs of the corresponding facts for $\mathcal{L}_A$; the fact that suprema always exist, and that the supremum operator is definable, pay off here.

8 Conclusion

$\Diamond$ is a mathematically attractive model. All suprema exist at all types, making proofs and reasoning simpler. Adding the supremum operator to the programming language makes the language universal as well as fully abstract; in contrast, the cpo model requires a new and fairly subtle operator $\exists$ to achieve universality.

This mathematical beauty conceals peril. We have examined four desirable properties of programming languages and their semantics: full abstraction, denotational universality, operational extensionality, and single-valuedness. For the flat lattice model $\Diamond$, the four are not attainable simultaneously. In contrast, the flat cpo semantics V admits all four properties.

Plotkin's paper [Plo77] is in part an example of how a well-chosen denotational semantics can inform programming language design. The results in this paper exemplify the dual fact: how a poorly-chosen denotational semantics can confound programming language design. The designer has a choice of three unappealing alternatives, listed in Figure 9. There is no obviously correct choice among these alternatives. The better choice seems to lie outside them, in the use of cpo or other more appropriate models.

	$\mathcal{L}_A(\mathsf{approx})$	$\mathcal{L}_A$(PCF-style)	$\check{\mathcal{L}}(\mathsf{pcond} + \top)$
Universality	–	–	yes
Full Abstraction	yes	–	yes
Op. Extensionality	–	yes	yes
Single-Valued	yes	yes	–

Figure 9: Choices of Semantic Properties

APPENDIX

A Proof of Theorem 6.4

Theorem A.1 *If $\mathcal{F}$ is an observation calculus, then for closed terms M and N, $M \equiv^{\mathcal{F}}_{\looparrowright} N$ iff $M \equiv^{ap,\mathcal{F}}_{\looparrowright} N$.*

Proof:

Trivially $M \equiv^{\mathcal{F}}_{\looparrowright} N$ implies $M \equiv^{ap,\mathcal{F}}_{\looparrowright} N$. Suppose that $M \equiv^{ap,\mathcal{F}}_{\looparrowright} N$ and $D[X]$ is a context driving M and N to ground type, such that $\pi \supset D[M] \looparrowright d$ for some proof π. We must show that $D[N] \looparrowright d$. The proof is of course by induction on $|\pi|$ simultaneously for all contexts D, and then cases on the structure of $D[X]$;

The only depth-0 proofs are those of the form $d \looparrowright d$. In this case, either $D[X] = d$ or both $D[X] = X$ and $M = d$. In both cases the argument is trivial.

Suppose that π is depth α for some ordinal α, and that the theorem is true for all contexts, and all proofs of depth less than α. We say that a proof is *shallow* if it is of depth less than α.

Case 1: If

$$D[X] = (\lambda y.D_0[X])D_1[X]\ldots D_k[X]$$

then the only rule applicable to $D[M]$ is the β-rule. If $D[M] \looparrowright d$, then π contains a shallow proof of

$$D_{01}[M]D_2[M]\ldots D_k[M] \looparrowright d$$

where $D_{01}[M] = D_0[y := D_1[M]][M]$ (Recall that M is closed; there is no substitution into M itself.) By induction (taking the context $D'[X] = D_{01}[X]D_2[X]\ldots D_k[X]$, and realizing that $D_{01}[N] = D_0[N][y := D_1[N]])$ we have $D_{01}[N]D_2[N]\ldots D_k[N] \looparrowright d$, and therefore $D[N] \looparrowright d$.

Case 2: If

$$D[X] = \delta D_1[X]\ldots D_k[X]$$

then π must start with some use of a rule θ for δ. There are shallow proofs for each of the antecedents of θ:

$$C_i[D_1[M],\ldots,D_k[M]] \looparrowright c_i \qquad i \in I.$$

By induction, we know that

$$C_i[D_1[N],\ldots,D_k[N]] \looparrowright c_i \qquad i \in I$$

and so rule θ gives us $D[N] \looparrowright d$.

Case 3: The other possibility is that

$$D[X] = XD_1[X]\ldots D_k[X]$$

Define

$$D'[X] = MD_1[X]\ldots D_k[X]$$

As $D[M] = D'[M]$, clearly $D'[M] \looparrowright d$. Note that $D'[X]$ is of form 1 or 2, and hence by the previous cases

$$D'[N] \looparrowright d.$$

We therefore have

$$D'[N] = MD_1[N]\ldots D_k[N] \looparrowright d.$$

Recall that $M \equiv^{ap,\mathcal{F}}_{\looparrowright} N$. Thus we conclude

$$D[N] = ND_1[N]\ldots D_k[N] \looparrowright d$$

as desired. $\square$

B Proof of Theorem 6.6

Theorem 6.6 *If M is a closed ground term of a simple LCF-like language $\mathcal{L}$, then $M \twoheadrightarrow c$ iff $M \looparrowright c$.*

Theorem 6.6 follows easily from two lemmas, B.2 and B.4, which we now prove.

First we must develop some basic properties of $\looparrowright$ in $\mathcal{L}!$. We say that the closed ground term M is *single-valued* iff, whenever $M \looparrowright c$ and $M \looparrowright d$, then $c = d$. If M is single-valued and $M \looparrowright c$, then by definition $M \equiv_{\looparrowright}^{ap,\mathcal{L}!} c$, and, by operational extensionality, $M \equiv_{\looparrowright}^{\mathcal{L}!} c$.

Lemma B.1 *All closed ground terms are single-valued in $\mathcal{L}!$.*

Proof:

Let π be a proof of $M \looparrowright c$. We proceed by induction on both the structure of M and the depth of π, with the hypothesis that if $\pi' \supset M' \looparrowright c'$ where either M' is a subterm of M or π' is shallower than π, then M' is single-valued.

The base step is $M = c$ or $|\pi| = 0$, which happen to be equivalent as we have defined $\mathcal{L}!$. By the definition of $\looparrowright$, $c \looparrowright c'$ implies $c = c'$ as desired.

The inductive step is a case analysis on the structure of M. M must be a closed ground term other than a constant, and so is either of the form $(\lambda x.R)S\vec{A}$ or $\delta\vec{R}$. If $M = (\lambda x.R)S\vec{A}$, then $M \looparrowright d$ iff $R[x := S]\vec{A} \looparrowright d$. As $R[x := S]\vec{A} \looparrowright c$ by a subproof of π, $R[x := S]\vec{A}$ is single-valued; therefore M is single-valued.

Otherwise $M = \delta\vec{R}$. Let $\rho!$ be the main rule of π; so $R_i \looparrowright c_i$ for each $i \in I$, and $P[\vec{x} := \vec{R}] \looparrowright c$. Suppose that some other rule $\rho'!$ also applies to M, proving that $M \looparrowright c'$. Then the antecedents of $\rho'!$ are satisfied: $R_i \looparrowright c'_{i'}$ for each $i' \in I'$ and $P'[\vec{x} := \vec{R}] \looparrowright c'$. We must show $c = c'$.

As each R_i for $i \in I \cup I'$ is a subterm of M, each is single-valued. So, $c_i = c'_i$ when $i \in I \cap I'$. Define

$$S_j = \begin{cases} c_j & j \in I \\ c'_j & j \in I' \\ R_j & \text{otherwise} \end{cases}$$

For $j \in I \cup I'$ we know $R_j \looparrowright S_j$; for other j, R_j and S_j are the same term. So, for each j, we have $R_j \equiv_{\looparrowright}^{\mathcal{L}!} S_j$. Contexts of congruent terms are congruent; in particular $P[\vec{x} := \vec{R}] \equiv_{\looparrowright}^{ap,\mathcal{L}!} P[\vec{x} := \vec{S}]$ and $P'[\vec{x} := \vec{R}] \equiv_{\looparrowright}^{ap,\mathcal{L}!} P'[\vec{x} := \vec{S}]$.

However, $\delta\vec{S}$ satisfies both ρ and ρ'. Therefore, by consistency, $P[\vec{x} := \vec{S}] = P'[\vec{x} := \vec{S}]$. Thus, we have $P[\vec{x} := \vec{R}] \equiv_{\looparrowright}^{ap,\mathcal{L}!} P'[\vec{x} := \vec{R}]$, and so $P[\vec{x} := \vec{R}] \looparrowright c'$. As $P[\vec{x} := \vec{R}] \looparrowright c$ by a subproof of π, we know that $P[\vec{x} := \vec{R}]$ is single-valued. Hence $c' = c$ as desired. $\Box$

Lemma B.2 *If $M \to N$ and M is closed, then $M \equiv_{\looparrowright}^{\mathcal{L}!} N$.*

Proof: By operational extensionality, it suffices to show that $M\vec{A} \looparrowright c$ iff $N\vec{A} \looparrowright c$. The proof is by induction on the structure of M, and then cases on why $M \to N$. The only nontrivial case is for a rule

$$\rho : \delta X_1 \ldots X_n \to P[\vec{x} := \vec{X}]$$

in which case $M = \delta R_1 \ldots R_n$, $\forall i \in I.R_i = c_i$, and $N = P[\vec{x} := \vec{R}]$. If $N\vec{A} \looparrowright c$, it is trivial to construct a proof that $M\vec{A} \looparrowright c$. Suppose, then, that $\pi \supset M\vec{A} \looparrowright c$; we must show $N\vec{A} \looparrowright c$.

The difficulty in this case comes from the fact that rules other than $\rho!$ will be applicable to $M\vec{A}$; we use consistency to prove the desired fact. The main rule of π is some δ-rule $\rho'!$, the observation-calculus form of the rule

$$\rho' : \delta X_1 \ldots X_n \to P'[\vec{x} := \vec{X}]$$

where $X_{i'} = c'_{i'}$ for $i' \in I'$.

The proof π contains proofs that $R_{i'} \looparrowright c_{i'}$ for each $i' \in I'$, and a proof that $P'[\vec{x} := \vec{R}] \looparrowright c$. By single-valuedness of $\mathcal{L}!$, if $i \in I \cap I'$, then $c'_i = c_i$. Define

$$S_j = \begin{cases} c'_j & j \in I' \\ R_j & \text{otherwise} \end{cases}$$

Then $R_j \equiv^{\mathcal{L}!}_{\looparrowright} S_j$ and hence $P'[\vec{x} := \vec{R}]\vec{A} \equiv^{\mathcal{L}!}_{\looparrowright} P'[\vec{x} := \vec{S}]\vec{A}$ and $P[\vec{x} := \vec{R}]\vec{A} \equiv^{\mathcal{L}!}_{\looparrowright} P[\vec{x} := \vec{S}]\vec{A}$. Therefore, $P'[\vec{x} := \vec{S}]\vec{A} \looparrowright c$.

Both ρ and ρ' apply to $\delta\vec{S}$, and by consistency we have $P[\vec{x} := \vec{S}] = P'[\vec{x} := \vec{S}]$, and so $N\vec{A} = P[\vec{x} := \vec{S}]\vec{A} \looparrowright c$ as desired. □

We have shown that computations in $\mathcal{L}!$ include those of $\mathcal{L}$. We now show the converse; first, we need to know a little about the proof theory of $\mathcal{L}!$. If M is a closed ground term which evaluates to a numeral m, and $C[M]$ does something, then $C[m]$ does the same thing with no more effort than $C[M]$ took.

Lemma B.3 *Let M be a closed ground term. If $M \looparrowright m$ and $\pi \supset C[M] \looparrowright c$, then there is a proof π' of $C[m] \looparrowright c$ such that $|\pi'| \leq |\pi|$.*

Proof: The proof is by induction on $|\pi|$. It is trivial for $|\pi| = 0$. Otherwise, we must examine the context: $C[X]$ is either X, $\delta\vec{C[X]}$, or $(\lambda x.R[X])S[X]\vec{A}[X]$. The first case follows from Lemma B.1 and the fact that the proof of $c \looparrowright c$ is depth 0; the others are trivial. □

Lemma B.4 *Let M be a closed ground term. If $M \looparrowright c$ then $M \twoheadrightarrow c$.*

Proof: Let π be a proof of $M \looparrowright c$; we induct on the depth of π. If π is trivial, then $M = c$ already. Otherwise, M is either of the form $\delta\vec{R}$ or $(\lambda x.R)S\vec{A}$. In the former case, π has main rule $\rho!$. The hypotheses $R_i \looparrowright c_i$ for $i \in I$ and $P[\vec{x} := \vec{R}] \looparrowright c$ are proved by subproofs of π, and so we have $R_i \twoheadrightarrow c_i$. Let

$$S_i = \begin{cases} c_i & i \in I \\ R_i & i \notin I \end{cases}$$

Note that $R_i \equiv^{\mathcal{C}!}_{\looparrowright} S_i$. As $P[\vec{x} := \vec{R}] \looparrowright c$ by a proof shallower than π, Lemma B.3 guarantees that there is a proof π'' of $P[\vec{x} := \vec{S}] \looparrowright c$ which is shorter than π. By induction,

$$P[\vec{x} := \vec{S}] \twoheadrightarrow c \tag{9}$$

It is straightforward to assemble these pieces into a reduction $M \twoheadrightarrow c$.

$$M = \delta\vec{R} \twoheadrightarrow \delta\vec{S} \rightarrow P[\vec{x} := \vec{S}] \twoheadrightarrow c$$

The first reduction sequence is justified by operand evaluation rules; the next part by rule ρ; and the last by equation 9.

If $M = (\lambda x.RS)\vec{A}$, then a subproof of π shows that $R[X := S]\vec{A} \looparrowright c$. By induction

$$R[X := S]\vec{A} \twoheadrightarrow c$$

and the conclusion follows easily by the β-rule. $\square$

C Proof of Theorem 7.1

Theorem C.1 $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$ *is adequate for $\Diamond$. That is, $[\![M]\!] \sqsupseteq [\![c]\!]$ (where c is a numeral and M a closed ground term) iff $M \twoheadrightarrow c$.*

Definition C.2 *A term M is* polite *iff:*

- *M is a closed term of base type, and for each numeral c of that type, $[\![M]\!] \sqsupseteq [\![c]\!]$ implies $M \twoheadrightarrow c$.*

- *M is a closed term of type $\sigma \rightarrow \tau$, and for all closed polite terms N of type σ, MN is polite.*

- *M is an open term, and every substitution instance of M, replacing all free variables by closed polite terms, is polite. We call such an instantiation a* polite instantiation.

Definition C.3 *$\Omega_\sigma = \mathbf{Y}(\lambda x^\sigma.x)$ is the standard divergent term of type σ; we omit the type σ. Let $\Omega' = (\lambda x.x)\Omega$. $\mathbf{Y}^{(0)} = \Omega$, and $\mathbf{Y}^{(n+1)} = \lambda f.f(\mathbf{Y}^{(n)}f)$ are the approximants of $\mathbf{Y}$.*

Note that $\Omega\vec{M}$ always diverges, and that $\Omega \to \Omega' \to \Omega$ is the only reduction sequence of Ω.

Lemma C.4 *For any closed term M, M is polite iff for all vectors $\vec{N}$ of closed polite terms such that $M\vec{N}$ is of ground type, $M\vec{N}$ is polite. Also, if $M \twoheadrightarrow M'$ where M' is polite and $[\![M]\!] = [\![M']\!]$, then M is polite.*

Proof: Easy. □

Lemma C.5 *Every term is polite.*

Proof: This is a proof by induction on the structure of terms. The distinctive feature of Tait's method is that that the "polite" predicate is defined by induction on types.

Variables are trivial. Whenever M is polite and the right type, $x[x := M] = M$ is polite.

Applications: Let M and N be polite terms. If MN is open, then any substitution instance by polite terms is of the form $M'N'$. M' and N' are polite instantiations of M and N, and therefore M' and N' are polite. The application of polite closed terms is polite; so $M'N'$ is polite for every polite instantiation as required.

Abstractions: Let M be a polite term. To show $\lambda x.M$ polite, we must consider an arbitrary polite instantiation $\lambda x.M'$. This term is polite iff for every closed polite N, $(\lambda x.M')N$ is polite. $(\lambda x.M')N \to M'[x := N]$, and $M'[x := N]$ is a polite instantiation of M and hence polite. As usual, $[\![(\lambda x.M')N]\!] = [\![M'[x := N]]\!]$. Therefore, $\lambda x.M$ is polite by Lemma C.4.

Constants must be checked individually. We present $\mathbf{1}+$ as a typical constant, and $\mathbf{Y}$ which is the only atypical constant. To show that $\mathbf{1}+$ is polite, we consider an arbitrary closed polite M; we must show that $[\![\mathbf{1}+M]\!] \sqsupseteq [\![c]\!]$ implies $\mathbf{1}+M \twoheadrightarrow c$. If $c = \top$, then $[\![\mathbf{1}+M]\!] = \top$, then we must have $[\![M]\!] = \top$ and (since M is polite) $M \twoheadrightarrow \top$; consequently $\mathbf{1}+M \twoheadrightarrow \mathbf{1}+\top \to \top$ as required.

If $c = n$, then we must have $[\![M]\!] = \top$ or $[\![M]\!] = n-1$. In the first case

$$\mathbf{1}{+}M \twoheadrightarrow \mathbf{1}{+}\top \to \mathbf{1}{+}(n-1) \to n$$

and otherwise

$$\mathbf{1}{+}M \twoheadrightarrow \mathbf{1}{+}(n-1) \to n$$

The rest of the first-order constants are polite by a similar case analysis. $\mathbf{Y}$ is nontrivial. It suffices to prove that, for each tuple $\vec{N}$ of polite closed terms grounding $\mathbf{Y}$, that $L = \mathbf{Y}\vec{N}$ is polite. Let $L^{(n)} = \mathbf{Y}^{(n)}\vec{N}$. $\mathbf{Y}^{(0)} = \Omega$ is clearly polite; $\mathbf{Y}^{(n+1)}$ is built from Ω by λ-abstraction and application and is therefore polite, and so $L^{(n+1)}$ is polite as well.

Suppose, then, that $[\![L]\!] \sqsupseteq [\![c]\!]$. $[\![L]\!] = \bigsqcup_{n=0}^{\infty} \left[\!\left[L^{(n)}\right]\!\right]$. All chains in the domains at ground type are finite — the longest are of length three — and so any directed set must contain its limit. Therefore, for some n, we must have $\left[\!\left[L^{(n)}\right]\!\right] \sqsupseteq [\![c]\!]$. As $L^{(n)}$ is polite, we have $L^{(n)} \twoheadrightarrow c$. By Lemma C.9 below, $L \twoheadrightarrow c$ as well, concluding the proof. □

Corollary C.6 (Adequacy) *$\diamondsuit$ is a computationally adequate model of $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$. That is, for any closed ground term M of $\tilde{\mathcal{L}}(\mathsf{pcond} + \top)$ and numeral c, $[\![M]\!] \sqsupseteq [\![c]\!]$ iff $M \twoheadrightarrow c$.*

We are finished, except for a missing lemma showing that $\mathbf{Y}$ behaves correctly. This requires some detailed examination of reductions. We define $\preceq$ on terms of the same type inductively:

1. $\Omega \preceq M$ and $\Omega' \preceq M$ for all M.
2. $\mathbf{Y}^{(n)} \preceq \mathbf{Y}$
3. $M \preceq M$
4. If $M \preceq M'$ and $N \preceq N'$ then $\lambda x.M \preceq \lambda x.M'$ and $MN \preceq M'N'$.

In other words, $M \preceq M'$ if M is M' with some $\mathbf{Y}$'s changed to $\mathbf{Y}^{(n)}$'s and some other subterms replaced by Ω's or Ω''s. This weak form of syntactic approximation will help prove that $\mathbf{Y}^{(n)}$ is truly an approximation of $\mathbf{Y}$.

Lemma C.7 *If $c \preceq M$ where c is a numeral, then $M = c$.*

Proof: What else could M be? □

Lemma C.8 *If $M \preceq N$ and $M \to M'$ then there is some N' such that $N \twoheadrightarrow N'$ and $M' \preceq N'$.*

Proof:

By structural induction on M and tedious case analysis on why $M \to M'$. Assume inductively that the lemma holds for all subterms of M, and that $M \to M'$ and $M \preceq N$; we will find N'. We present only one case.

If the rule proving $M \to M'$ was the rule for true conditional, then $M = \mathsf{cond}\ \mathsf{tt}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ and $M' = P$. Now, M is neither Ω, Ω', nor $\mathbf{Y}^{(n)}$, and since $M \preceq N$, either $N = M$, in which case the lemma clearly holds, or $N = \mathsf{cond}\ B\ \mathsf{then}\ P'\ \mathsf{else}\ Q'$ with $\mathsf{tt} \preceq B$, $P \preceq P'$ and $Q \preceq Q'$. In the latter case, $\mathsf{tt} \preceq B$, then $B = \mathsf{tt}$; and so the conditional rule gives us $N \to P'$. P' is the desired N'. □

Lemma C.9 *If $\mathbf{Y}^{(n)}\vec{N} \twoheadrightarrow c$ for some numeral c, then $\mathbf{Y}\vec{N} \twoheadrightarrow c$.*

Proof: Note $\mathbf{Y}^{(n)}\vec{N} \preceq \mathbf{Y}\vec{N}$. Look at the derivation of $\mathbf{Y}^{(n)}\vec{N} \twoheadrightarrow c$. For each term L_i in this sequence, there is a descendant M_i of $\mathbf{Y}\vec{N}$ such that $L_i \preceq M_i$:

$$\begin{array}{rcccccl} \mathbf{Y}^{(n)}\vec{N} & \to & L_1 & \to & \cdots & \to & L_n = c \\ \mathbf{Y}\vec{N} & \twoheadrightarrow & M_1 & \twoheadrightarrow & \cdots & \twoheadrightarrow & M_n \end{array}$$

By lemma C.7, $M_n = c$ as required. □

D Acknowledgements

I would like to thank Albert Meyer for asking the question which inspired this line of research. A variety of people, including Alan Fekete, Gordon Plotkin, John Reynolds, Jon Riecke, Dana Scott, Rick Statman, and Allen Stoughton provided valuable insight in discussions, and other help and encouragement.

References

[Abr] Samson Abramsky. The lazy lambda calculus. December 17, 1987. Imperial College of Science and Technology.

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 1981. Revised Edition, 1984.

[BCL85] G. Berry, P.-L. Curien, and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. Reynolds, editors, *Algebraic Methods in Semantics*, page ?? Cambridge Univ. Press, 1985.

[BR89] Bard Bloom and Jon Riecke. Lcf should be lifted. In *Proceedings of Algebraic Methodology and Software Technology*, 1989.

[Fri75] H. Friedman. Equality between functionals. In R. Parikh, editor, *Logic Colloqium, '73*, volume 453, pages 22–37. Springer-Verlag, 1975.

[MC88] Albert R. Meyer and Stavros Cosmadakis. The fundamental theorem of scott domains. Technical report, MIT, 1988.

[Mey82] Albert R. Meyer. What is a model of the lambda calculus? *Information and Computation*, 52:87–122, 1982.

[Mil77] Robin Milner. Fully abstract models of typed λ-calculus. *Theoretical Computer Sci.*, 4:1–22, 1977.

[Mul85] Ketan Mulmuley. *Fully Abstract Submodels of Typed Lambda Calculi.* PhD thesis, CMU, 1985.

[Mul86] Ketan Mulmuley. *Full Abstraction and Semantic Equivalence.* MIT Press, 1986.

[Ong88] Luke Ong. *The lazy lambda calculus: an investigation into the foundations of functional programming.* PhD thesis, Imperial College, University of London, 1988.

[Plo77] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–256, 1977.

[Plo88] Gordon Plotkin. Notes on a variant language and information systems. Unpublished notes, 1988.

[Saz76] V.Yu. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra and Logic*, 15:192–206, 1976. (Original in Russian).

[Sco69] Dana Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Manuscript, Oxford Univ. Used by permission of the author., 1969.

[Sco76] D. Scott. Data types as lattices. *SIAM J. Computing*, 5:522–587, 1976.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.