LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-353

SEMANTICAL PARADIGMS: NOTES FOR AN INVITED LECTURE

Albert R. Meyer
with Two Appendices by
Stavros S. Cosmadakis

July 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Semantical Paradigms:

Notes for an Invited Lecture*

Albert R. Meyer†
MIT Lab. for Computer Science

with Two Appendices by

Stavros S. Cosmadakis
IBM Watson Research Center

Abstract. It took me quite a few years to understand the point of continuity in denotational semantics. I'm happy to report below on some recent results which justify my muddle-headedness and help explain the point too. What follows are some global comments on denotational semantics of the kind invited lecturers sometimes indulge themselves in, highlighting "goodness of fit" criteria between semantic domains and symbolic evaluators. For readers impatient with sketchy overviews, two appendices mostly by Cosmadakis provide the key parts of a long proof that Scott domains give a *computationally adequate* and *fully abstract* semantics for lambda calculus with simple *recursive* types.

CR Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory—*syntax, semantics*; F.3. [Logics and Meanings of Programs]: F.3.1: Specifying and Verifying and Reasoning about Programs—*program equivalence*; F.3.2: Semantics of Programming Languages–*operational semantics, denotational semantics*; F.3.3: Studies of Program Constructs.

General Terms: Programming Languages, Semantics, Logic, Correctness

Additional Key Words and Phrases: denotational semantics, cpo's, lattices, continuity, functors, observational equivalence, lambda calculus, full abstraction

---

*This Technical Memorandum is a slightly revised version of a paper in the Proceedings 3^{rd} IEEE Symposium on Logic in Computer Science, Edinburgh, Scotland, July, 1988.

†Supported in part by NSF Grant No. DCR-8511190 and by ONR Grant No. N00014-83-K-0125.

1 Introduction

It was Strachey's imaginative insight to identify common phrase-types such as identifiers, left- and right-expressions, declarations, and commands, and to realize how much about a programming language could be understood by describing the domains of "values" which these phrases may have [39,20]. This is an insight in comparative programming linguistics; Strachey's notions resemble those of natural language, and they can be understood and used in the same intuitive but precise way we recognize nouns and verbs—a good thing, since understanding Strachey's "values" with *mathematical* rigor involves an armamentarium of mathematical weapons otherwise unfamiliar in Computer Science.

My consistent observation is that among even that minority of programming language experts and compiler developers who make use of Strachey's insights, few have a technical understanding of Scott's signal contribution of a mathematically sound foundation for denotational semantics. These very capable people typically have the good judgment to let their minds wander when subjected to lectures about directed limits, continuous functions, retractions, or the $P\omega$ model of untyped lambda calculus. This is not meant as a criticism of the relevance of Scott's work. An analogy I heard from Scott himself helps explain why no criticism need be inferred: electrical engineers are not taught how to construct complex numbers from ordered pairs of downward closed sets of rational numbers, whereas mathematicians typically are taught about them (Dedekind's cuts). How come? Because there is a robust geometric intuition which can be conveyed about the complex plane, and there is an elegant calculus for, and axiomatization of, the complex field which gives a reliable way to verify geometric intuitions. Mathematicians with foundational concerns may study Dedekind's cuts to confirm the correctness of the logic, but engineers can skip them.

Lambda calculus and lambda reduction, and general reasoning principles like least fixed point induction, offer similar insulation of the program engineer from the foundational concepts of domain theory. One can prove a fair amount about program semantics using the kind of axiom systems supported by LCF [8,22] without mastering the intricacies of information systems (Scott's version of "Dedekind" cuts) [32], though domain theory is far from providing the pragmatically powerful and technically complete logical theory of the kind we have for the complex field.

On the other hand, domain theory remains an active area, as researchers continue to explore a surprisingly varied crop of possibilities: Scott's original continuous lattice domains [31] gave way to complete partial orders (cpo's) with continuous morphisms [24,26]; more significant, Scott's domains did not support the kind of "power-domain" type construction desirable for explaining the meaning of nondeterministic programs, and Plotkin offered the richer category of SFP domains [24]; variant SFP's have been further elaborated into profinite domains [11], 2/3-SFP's, and more. Meanwhile, it appeared from the independent solutions of Sazonov [30] and Plotkin [25] to a question raised by Scott that there was something *inherently parallel* in domains based on Scott's notion of continuity—more about this below—and the stable and dI-domains of Berry, *et al.* were proposed [2] to capture *sequential* interpreters (they don't quite). Stable domains then found an unexpected independent application as models of polymorphic types in Girard's qualitative domains [7]. Recently, L-domains have been offered [12,40,41] as an improvement on dI and SFP domains. I'll say more in the next section about the domains of *monotonic* functions; they are pedagogically

much simpler and serve surprisingly well for a widely studied case. All these domains based on functions on cpo's seem limited in their ability to model block-structure in ALGOL-like languages [13,42], leading Oles [23], and me and Sieber [19] to obtain improved, but still imperfect models using functor-categories on cpo's. Other kinds of domains whose theory has a more algebraic/categorical, as opposed to order-theoretic, flavor are presented in [6,9,10]

Too many different domains of course; I hope the best ones will emerge in time. One theme hinted at in the litany above is that each of these domains was developed to model some *kind* of computation or computational logic. But why are there so many? Doesn't Church's thesis indicate that there is only *one* kind of computation? Alan Perlis calls this the "Turing tarpit": some of the most crucial distinctions in computing methodology, such as sequential versus parallel, determinate versus multivalued, iterative versus recursive, local versus distributed, call-by-name versus call-by-value, get mired together if all you see in computation is symbol-pushing. (Note that none of these distinctions correlates much with computational complexity. I've always thought "complexity theory" was a misnomer, since a very simple computation carried out for a large number of repetitions is designated as complex, while a sophisticated fast algorithm with elaborate data structures is not called complex. "Efficiency theory" would be more accurate.) So Computer Scientists clearly make distinctions ignored in elementary Recursion Theory and Complexity Theory.

My speculation is that the proliferation of domains may be reflecting this multiplicity of computational distinctions. For example, besides the Plotkin/Sazonov results which I interpret as connecting cpo's with determinate parallel computation, the paper by Bard Bloom in the 1988 LICS symposium suggests that continuous lattices best model computation by nondeterministic interpreters. There remains a lot of fuzziness in these ideas of "kinds" of computation and how they are modeled by different domains. I'm not confident that these speculations can be precisely formulated, let alone that they will hold up. But I've found, and hope the results sketched below will persuade at least a few readers, that pursuing them has been worthwhile.

2 Some "Good Fit" Criteria

Denotational semantics allows clean mathematical concepts like partial orders, least fixed points, continuity, and higher-order functions, to be brought to bear in reasoning about programming languages. But the relevance of the mathematical facts to the computational situation depends on the nature of the fit between mathematical meaning and computational behavior, as well as the reasonableness of both the domains of meaning and the computational systems. Examining the fit provides guidance in analyzing and designing languages and their semantics.

Let me review some fundamental fitness and reasonableness criteria:

1. *Computational adequacy*: a term means 3 iff it evaluates computationally to the numeral for 3. This is the essential connection between computation and meaning. Without it, semantics is not much use in explaining computational behavior.
2. *Full abstraction*: two terms are semantically equal iff they denote the integer 3 in exactly the same contexts.
3. *Universality*: every computable value of any type is definable by a term.
4. *Structured operational semantics (SOS)*—in the style of [27]: having one is a "reasonableness" criterion for a symbolic interpreter.

The classic study connecting these criteria is Plotkin's "LCF Considered as a Programming Language" [25]. I've found it well worth using as the basis of an introductory graduate lecture course in semantics.

There is a purely symbol-pushing computational notion that programmers appreciate as fundamental: two pieces of program are "equivalent" if they can *always* be *interchanged* without affecting the visible results of the computation. More precisely,

Definition 1 *Two terms M, N are* observationally distinguishable *iff there is a context $C[\cdot]$ such that $C[M]$ evaluates to the numeral 3 and $C[N]$ does not, or vice versa. M and N are* observationally congruent, *written $M \equiv_{obs} N$, iff they are not observationally distinguishable.*

How come the numeral 3 is an important output? Well of course it isn't; if you prefer 7, then the context $C[\cdot]+4$ will distinguish M and N wrt to observing 7 whenever $C[\cdot]$ does the job wrt 3. In particular, the relation $\equiv_{obs}$ remains unchanged whether we regard 3, 7, or any nonempty subset of numerals to be visible results. For the simply typed lambda calculus we also get the same $\equiv_{obs}$ if we distinguish terms solely on the basis of whether or not their evaluation produces a numeral at all, that is M and N are observationally distinguishable iff there is a closing context $C[\cdot]$ *of integer type* such that evaluation of exactly one of $C[M]$ and $C[N]$ terminates.

So optimizations by a compiler are "correct" providing the compiler replaces program texts by observationally congruent texts. This implies that although $\equiv_{obs}$ is invariant over many choices of what observable outcomes of computation are taken to be, we don't expect to allow observers with clocks who can time computations, since the point of carrying out the observation-preserving optimization was to speed things up.

For mainstream Computer Scientists who think operationally and require a pithy explanation of how denotational semantics helps them with their own concerns, we can say that semantics provides a whole new set of ways to prove observational congruences:

> If a semantics is adequate (and compositional, but let's not be picky), then semantic equality implies observational congruence.

For example, try proving from purely operational definitions that

$$(Y\ \lambda x^{\tau}.\ x) \equiv_{obs} ((Y\ \lambda f^{int\to\tau}.\ f)\ 3).$$

It can be done, but the proof is not easy. On the other hand, it follows trivially that these terms have the same meaning in models where Y denotes a least fixed point operator since the least fixed point of the identity function is the constant $\perp$ function. Since we have many such models which are adequate, we can conclude the terms are $\equiv_{obs}$.

This is not to say that semantical proofs are shorter or simpler—after all, the trivial argument above rests on a nontrivial adequacy proof—but they certainly have an attractive flavor of their own compared to reasoning about step-by-step transformations by SECD machines.

The main theorems culminating most papers and texts on semantics are just adequacy theorems. To some degree this achieves the task of capturing semantically what matters computationally because any *adequate* semantics uniquely determines $\equiv_{obs}$ without having to mention the evaluator: $M \equiv_{obs} N$ iff

$$\forall C[\cdot].\ [\![C[M]]\!] \neq \perp_{int} \text{ iff } [\![C[N]]\!] \neq \perp_{int}\ .$$

So it's nice that adequacy is cheap, *e.g.*, Plotkin demonstrates that continuous lattice

models, cpo's with extra, infinite integers $\neq \top$, as well as Scott cpo's, each provide an *adequate* computational setup for the simply typed lambda calculus with recursion and conditional combinators and call-by-name evaluation. In fact, in joint work with D. Velleman of Amherst College, I observe that even the category of cpo's with *monotone*—as opposed to continuous—functions is adequate for the simply typed calculus. (The proof is easy using Statman's *logical relations* [35,34] to relate the monotone and continuous cpo categories.) So if all that matters is adequacy, continuity in cpo's can be ignored in favor of the pedagogically simpler, familiar notion of monotonicity. Moreover, the basic principle of fixed point induction on admissible predicates is sound in the monotone case.[1] This may explain why in my early reading about semantics I had some trouble seeing the role of continuity.

The problem is that even though adequate meanings *determine* congruence in a mathematical sense, and equal meanings implies congruence, an adequate semantics may make *more* distinctions than those definable by contexts, so observationally congruent terms may not be semantically equal. Full abstraction ensures that the only semantical distinctions made are observational ones:

> A semantics is fully abstract iff
> semantic equality coincides with $\equiv_{obs}$.

In the simply typed case, the significance of continuity only begins to emerge from Plotkin's result that, at least once a "parallel-conditional" is added to the simply typed calculus, the cpo's with continuous functions provide a fully abstract semantics. Plotkin observes that this fails for continuous lattices, which explains part of the reason why in the current Computer Science literature lattices have been largely abandoned in favor of cpo's. Finally, he shows that a further extension with a "continuous-existential" combinator yields universality for continuous cpo's.

Thinking along the lines in the Introduction, I asked whether there was some *other* language extension than parallel-conditional for which continuous lattices are fully abstract. Bloom makes the sophisticated observation in this LICS that lattices *are* fully abstract when certain computable combinators enrich the language of terms; but he then proves that all such combinators are necessarily unreasonable; they cannot fit a certain kind of SOS format among other problems. Universality necessarily fails for lattices on recursion theoretic grounds unless we admit some rather odd nondeterministic evaluators.

I also asked whether the monotone cpo models were fully abstract, and Plotkin first came up with a counter-example. Velleman went on to show that full abstraction fails *irreparably* for the monotone cpo model—no matter what language extensions are added to PCF so long as terms have an effective, adequate symbolic evaluator.

So we have a good rationale for continuous reasoning in the simply typed case. But doubts about the point of continuity are clarified by the observation that the monotone model *is* fully abstract for the language of first-order *recursive function schemes* (with parallel-or). So the use of continuous functions in standard references which consider only such schemes, *e.g.*, [15,16,28], is a red herring—everything works under the simpler monotone interpretation. There is no point in continuous reasoning without higher-order (at least third-order) types.

I keep saying "simply typed" for good reason. Another key purpose of continuity is to justify the rules for reasoning about recursive types and domain equations. For example, Abramsky and Stoughton have recently strengthened an earlier observation of Plotkin: in the monotone framework there is no model of the untyped lambda-β calculus, namely, a nontrivial solution of the re-

traction $(D \to D) \lhd D$ does not exist in the category of cpo's with monotone functions as morphisms.[2]

I hope to spell out this whole neat story about monotonicity in a joint paper with Plotkin and Velleman sometime soon.

Another story along these lines that I will be telling in more detail elsewhere [18], concerns *continuations*. I still don't understand them, but I have a better idea why. The basic theorems in the literature about continuations are all *congruence* theorems which are the recursively typed versions of logical relations, *e.g.,* [37,38,29]. The gist of these results is that a term means 3 in "direct" functional semantics iff it means 3 in continuation semantics. These are essentially adequacy results. And in fact, once I proposed looking, it was not very hard to find examples where full abstraction fails: there are simple functional terms which are equal in direct semantics but not in continuation semantics. To my amazement, only a couple of experts on the subject seemed aware of this phenomenon, and none seemed to appreciate the consequence: reasoning which is sound for programs under direct semantics may be unsound for the same programs under continuation semantics. I wish the advocates of continuation style had warned me about this problem and would offer more help in reasoning about continuations (and don't call my attention to [4,5], which, despite a titular claim, don't fill the bill.)

I recommend Stoughton's recent monograph [36] for a well-written, thorough examination of full abstraction, as well as a balanced discussion of the nature of the somewhat oversold "solution" to the full abstraction problem for sequential PCF offered in [22]. One warning though: Stoughton follows what I consider the unfortunate terminology of [14] and calls "full abstraction" a property that is actually equivalent to what I call adequacy—his "contextual full abstraction" is my full abstraction.[3]

3 Observing Termination

Now if we are willing to observe termination at integer or other printable-value types, why not observe termination at *all* types? For that matter, there may even be no obvious alternative to observing termination everywhere in many interesting situations of "pure" untyped, recursively typed, or dependently typed calculi where there are no built-in integer types with numerals to observe. And after all, even if, say, a LISP expression evaluates to a closure rather than a printable value, the fact that we get a prompt at the terminal when evaluation completes is a rather significant observable outcome—one we ought to be able to reason about semantically.

So we arrive at the final fitness criterion I want to consider:

5. *Complete adequacy*: the meaning of an arbitrary term is bottom (or undefined) iff evaluation of it does not terminate.

Plotkin, in a series of unpublished notes over the past three years, has established complete adequacy using domains of bottomless cpo's and continuous *partial* functions to assign meaning to the standard *recursively typed* lambda calculus with a standard *call-by-value* evaluation. As of our last discussion, full abstraction and universality remained unexamined for this setup. This earlier work stimulated my own questions about complete adequacy for Scott domains.

Now it is a folk theorem—which is to say that Scott, Gunter, and Abramsky said "of course" when I mentioned it, but I know of no reference—that on general principles of Scott domains, the set of terms in the *recursively typed* lambda calculus which are not identically bottom is a recursively enumerable collection of

syntactic objects. Thus, recursion theoretically speaking, there is some effective "evaluator" of recursively typed terms for which Scott domains are completely adequate.

The problem is that this evaluator is weird. Standard interpreters, whether for call-by-name or call-by-value style semantics, stop at formal abstractions. For example, let M be a closed term whose evaluation diverges. It should be a familiar fact that "hiding" M under a λ as in $\lambda x.Mx$ yields a term which terminates immediately at itself. Of course, $\lambda x.Mx$ is semantically equal to M, and indeed is observationally congruent to M (under call-by-name) if the only "printable values" or "computational observables" are numerals (or termination at *ground* type—note that M above has functional type since it applies to x). So if we allow termination behavior of the standard interpreters to be observable for terms of all types, then familiar reasoning like the (η)-axiom at functional types is *unsound*, and all the models in which it is sound are computationally inadequate! So complete adequacy certainly fails for the setup of Scott domains and simply typed lambda calculus *using the familiar evaluators.*

There is a solid clue in Wadsworth's classic study [43] of the pure lambda calculus of how a "reasonable" interpreter should work to be completely adequate for Scott domains. Wadsworth shows that an interpreter which stops reducing precisely at *head-normal forms* is fully abstract for pure untyped $\lambda\beta$-calculus. So if Cosmadakis and I could figure out how to generalize head-normal forms to the recursively typed lambda calculus, we might be able to exhibit reasonable, though nonstandard, interpreters for this calculus such that Scott domains are completely adequate *and* fully abstract. But so far we can't find an interpreter which has some kind of SOS that does the job.

So in Appendix A we work out the generalization of Plotkin's LCF study to the recursively typed lambda calculus. We do this by defining a very general class of *observed* types which includes the recursively definable versions of such printable values as integers, booleans, and lists and streams over observable atoms. Sticking with termination at observed type as the observation used to define $\equiv_{obs}$, we exhibit an interpreter which looks intuitively reasonable, and then we prove complete adequacy, full abstraction, and universality. But the reader should look at the reduction rules and judge for himself whether he likes them, since we don't know exactly what makes an SOS discipline reasonable (*cf.* [3] and Bloom's LICS '88 paper for some SOS metatheory). In particular, our interpreter uses some deterministic context-free pattern matching to control applicability of reduction rules, and we're not sure whether this control mechanism might be too powerful—enough to stick us in the Turing tarpit again.

An odd behavior of our interpreter arises from the fact that it has been optimized to stop as soon as it can, once a term is discovered to be of a canonically nonbottom form. In particular, the interpreter may stop on an integer term denoting 0 *before* evaluating to the numeral 0 if it discovers earlier that the term is nonbottom. This is probably reparable.

A criticism of our interpreter which would not be fair is that on terms which mean the pair $\langle 3,3\rangle$, it does not terminate with a standard printable representation of $\langle 3,3\rangle$. This is irreparable on recursion-theoretic grounds: an evaluator that is required to print $\langle 3,3\rangle$ would in general have to diverge on terms which meant $\langle 3,\bot\rangle$, so that $\bot$ and divergence could no longer match at type $int \times int$.

The hard part of designing an evaluator for which Scott domains are completely adequate involves sum types. In Appendix B we exhibit another interpreter for which Scott domains are

completely adequate at all recursive types not involving sums.

The theorems we have obtained, though in several respects only partial results, are not easy. There is a long way to go if we take these fitness criteria seriously and ask about the many other kinds of domains. These criteria also make sense for languages with richer types supporting power-domains and polymorphism. Work on these has not even begun.

Acknowledgment.

Thanks to Bard Bloom, Steve Brookes, Irene Greif, and Jon Riecke for late night proof-reading, to Samson Abramsky, Matthias Felleisen, Yuri Gurevich, John Mitchell, Gordon Plotkin, Vaughan Pratt, Dana Scott, and Allen Stoughton for comments and corrrections, and to Sally Bemus for an intense LaTeXing effort.

4 Notes

1. But different syntactic criteria for detecting admissible formulas are required in the monotone and continuous cases, *e.g.,* the predicate of x

   $$(x\ \lambda z.z) \sqsubseteq \bot$$

   is admissible in the continuous model, but not in the monotone model. The formal system LCF, which recognizes as admissible any predicate of the form $M \sqsubseteq N$, consequently allows proofs by fixed point induction of equations which hold in the continuous, but not in the monotone model. Dana Scott pointed this out to me, correcting an earlier remark to the contrary in the 1988 LICS proceedings version of this paper.

2. However, Abramsky has pointed out to me that, contrary to a remark in the earlier version of this paper in the 1988 LICS Proceedings, D and $D \to D$ may have the same cardinality in the monotone frame—letting D be the real numbers is an example.

3. My cryptic, provocative remarks here have already succeeded in stimulating a useful discussion among the research protagonists, *cf.* [17], which has led me to moderate my views a bit. Among other things, we are trying to reach agreement on common terminology for concepts like contextual and full abstraction.

A Adequacy at Observed Types

A.1 Syntax of Types

Let t stand for a type variable, τ for a type expression, and σ for an *observed* type:

$$\begin{array}{lcl} \tau & ::= & t \mid \tau \to \tau \mid \tau \times \tau \mid \tau \oplus \tau \mid \tau_\bot \mid \mu t.\tau \\ \sigma & ::= & t \mid \sigma \times \sigma \mid \sigma \oplus \sigma \mid \tau_\bot \mid \mu t.\sigma \end{array}$$

Definition 2 *A* type *is a closed type expression.*

Comments:
The symbol $\times$ denotes Cartesian (separated) product; we cannot handle strict (coalesced) product ($\otimes$) for reasons explained at the end of this appendix.

Function types are *not* observed.

The "lifted" type $\tau_\bot$ is observed for *any* type τ.

The symbol $\oplus$ denotes coalesced (smash) sum.

Separated sum ($+$) can be treated as "syntactic sugar" since it is definable by $\tau_1 + \tau_2 ::= (\tau_1)_\bot \oplus (\tau_2)_\bot$.

We don't have any purely semantical characterization of what makes a type observed.

Examples of types:

$$\begin{aligned} \mathit{triv} &::= \mu t.t \\ \mathbf{1} &::= \mathit{triv}_\perp \\ \mathit{bool} &::= \mathbf{1} \oplus \mathbf{1} \\ \mathit{int} &::= \mu t.\mathbf{1} \oplus t \\ \mathit{untyp} &::= \mu t.\mathbf{1} \oplus (t \to t) \end{aligned}$$

Comment: The type *untyp* is a model for the untyped $\lambda\beta$-calculus.

A.2 Terms and Typing Rules

Let M and N stand for terms, C for a canonical term, and D for a noncanonical term.

$$\begin{aligned} x^\tau &: \tau \\ \lambda x^{\tau_1}.M^{\tau_2} &: \tau_1 \to \tau_2 \\ (M^{\tau_1 \to \tau_2} N^{\tau_1}) &: \tau_2 \\ \mathit{pair}(M^{\tau_1}, N^{\tau_2}) &: \tau_1 \times \tau_2 \\ \mathit{fst}(M^{\tau_1 \times \tau_2}) &: \tau_1 \\ \mathit{snd}(M^{\tau_1 \times \tau_2}) &: \tau_2 \\ \mathit{inL}(M^{\tau_1}) &: \tau_1 \oplus \tau_2 \\ \mathit{outL}(M^{\tau_1 \oplus \tau_2}) &: \tau_1 \\ \mathit{inR}(M^{\tau_2}) &: \tau_1 \oplus \tau_2 \\ \mathit{outR}(M^{\tau_1 \oplus \tau_2}) &: \tau_2 \\ \mathit{condlr}\ M_1^{\sigma_1 \oplus \sigma_2}\ M_2^\tau\ M_3^\tau &: \tau \\ \mathit{lift}(M^\tau) &: \tau_\perp \\ \mathit{drop}(M^{\tau_\perp}) &: \tau \\ \mathit{up?}\ M^{(\tau_1)_\perp}\ N^{\tau_2} &: \tau_2 \\ \mathit{abs}(M^{[\mu t.\tau/t]\tau}) &: \mu t.\tau \\ \mathit{rep}(M^{\mu t.\tau}) &: [\mu t.\tau/t]\tau \end{aligned}$$

Comments:
The parallel case statement constructor, *condlr*, allows branching on whether a term, M_1, of *observed* sum type is in the left or right side of the sum, returning the glb of its remaining arguments, M_2 and M_3, if M_1 is bottom.

The constructor *up?* tests whether M_1 of lifted type is nonbottom, and if so returns its second argument; otherwise it returns bottom.

Some Constants:

$$\begin{aligned} Y^{(\tau\to\tau)\to\tau} &::= \lambda f^{\tau\to\tau}.\Delta_f^{(\mu t.t\to\tau)\to\tau}\, \mathit{abs}(\Delta_f), \\ \text{where } \Delta_f &::= \lambda x^{\mu t.t\to\tau}.f(\mathit{rep}(x)\ x), \\ \Omega^\tau &::= Y^{(\tau\to\tau)\to\tau}(\lambda x^\tau.x), \\ a^{\mathbf{1}} &::= \mathit{lift}(\Omega^{\mathit{triv}}), \\ \mathrm{tt}^{\mathit{bool}} &::= \mathit{inL}(a), \\ \mathrm{ff}^{\mathit{bool}} &::= \mathit{inR}(a), \\ 0^{\mathit{int}} &::= \mathit{abs}(\mathit{inL}(a)^{\mathbf{1}\oplus\mathit{int}}), \\ (+1)^{\mathit{int}\to\mathit{int}} &::= \lambda x^{\mathit{int}}.\mathit{abs}(\mathit{inR}(x)^{\mathbf{1}\oplus\mathit{int}}). \end{aligned}$$

Canonical Terms:

$$\begin{aligned} C ::= {} & \mathit{pair}(C,M) \mid \mathit{pair}(M,C) \mid \mathit{inL}(C) \mid \\ & \mathit{inR}(C) \mid \mathit{lift}(M) \mid \mathit{abs}(C) \end{aligned}$$

Definition 3 *Let $\rho_\perp$ be the valuation of variables (i.e., environment) that assigns $\perp$ of appropriate type to each variable.*

Comments:
For M an arbitrary term, $[\![M]\!]\rho \neq \perp$ for every valuation ρ iff $[\![M]\!]\rho_\perp \neq \perp$.

For C a canonical term, $[\![C]\!]\rho_\perp \neq \perp$.

A.3 Operational Rules

A "reduces in one step" relation, $\to$ on terms is defined inductively by the rules below. Let $\equiv$ denote syntactic identity of terms.

$$\begin{array}{rcl}(\lambda x.M)N & \rightarrow & [N/x]M \\ (condlr\ M\ N_1\ N_2)N_3 & \rightarrow & condlr\ M\ (N_1N_3)\ (N_2N_3)\end{array}$$

$$\frac{M \rightarrow M',\ M \not\equiv (\lambda \cdots),\ M \not\equiv condlr(\cdots)}{MN \rightarrow M'N}$$

$$\frac{M \rightarrow M',\ N \rightarrow N'}{pair(M,N) \rightarrow pair(M',N')}$$

$$\begin{array}{rcl}fst(pair(M,N)) & \rightarrow & M \\ snd(pair(M,N)) & \rightarrow & N \\ fst(condlr\ M\ N_1\ N_2) & \rightarrow & condlr\ M\ fst(N_1)\ fst(N_2) \\ snd(condlr\ M\ N_1\ N_2) & \rightarrow & condlr\ M\ snd(N_1)\ snd(N_2)\end{array}$$

$$\frac{M \rightarrow M',\ M \not\equiv pair(\cdots),\ M \not\equiv condlr(\cdots)}{fst(M) \rightarrow fst(M'),\ snd(M) \rightarrow snd(M')}$$

$$\frac{M \rightarrow M'}{inL(M) \rightarrow inL(M'),\ inR(M) \rightarrow inR(M')}$$

$$\begin{array}{rcl}outL(inL(M)) & \rightarrow & M \\ outR(inR(M)) & \rightarrow & N \\ outL(condlr\ M\ N_1\ N_2) & \rightarrow & condlr\ M\ outL(N_1)\ outL(N_2) \\ outR(condlr\ M\ N_1\ N_2) & \rightarrow & condlr\ M\ outR(N_1)\ outR(N_2)\end{array}$$

$$\frac{M \rightarrow M',\ M \not\equiv inL(\cdots),\ M \not\equiv condlr(\cdots)}{outL(M) \rightarrow outL(M')}$$

$$\frac{M \rightarrow M',\ M \not\equiv inR(\cdots),\ M \not\equiv condlr(\cdots)}{outR(M) \rightarrow outR(M')}$$

$$\begin{array}{rcl}drop(lift(M)) & \rightarrow & M \\ drop(condlr\ M\ N_1\ N_2) & \rightarrow & condlr\ M\ drop(N_1)\ drop(N_2)\end{array}$$

$$\frac{M \rightarrow M',\ M \not\equiv lift(\cdots),\ M \not\equiv condlr(\cdots)}{drop(M) \rightarrow drop(M')}$$

$$up?\ lift(M)\ N \rightarrow N$$

$$\frac{M \not\equiv lift(\cdots),\ M \rightarrow M'}{up?\ M\ N \rightarrow up?\ M'\ N}$$

$$\frac{M \rightarrow M'}{abs(M) \rightarrow abs(M')}$$

$$rep(abs(M)) \rightarrow M$$

$$rep(condlr\ M\ N_1\ N_2) \rightarrow condlr\ M\ rep(N_1)\ rep(N_2)$$

$$\frac{M \rightarrow M',\ M \not\equiv abs(\cdots),\ M \not\equiv condlr(\cdots)}{rep(M) \rightarrow rep(M')}$$

$$\begin{array}{rcl}
condlr\ inL(C)\ N_1\ N_2 & \rightarrow & N_1 \\
condlr\ inR(C)\ N_1\ N_2 & \rightarrow & N_2 \\
condlr\ D\ pair(M,N)\ pair(M',N') & \rightarrow & pair(condlr\ D\ M\ M',\ condlr\ D\ N\ N') \\
condlr\ D\ inL(M)\ inL(N) & \rightarrow & inL(condlr\ D\ M\ N) \\
condlr\ D\ inR(M)\ inR(N) & \rightarrow & inR(condlr\ D\ M\ N) \\
condlr\ D\ lift(M)\ lift(N) & \rightarrow & lift(condlr\ D\ M\ N) \\
condlr\ D\ abs(M)\ abs(N) & \rightarrow & abs(condlr\ D\ M\ N)
\end{array}$$

(if no *condlr* rule above applies)

$$\frac{M_i \rightarrow M_i' \text{ for } i \in I \neq \emptyset, \text{ and } M_j \equiv M_j' \text{ is canonical for } j \in \{1,2,3\} - I}{condlr\ M_1\ M_2\ M_3 \rightarrow condlr\ M_1'\ M_2'\ M_3'}$$

(if no rule above applies to D)

$$D \rightarrow D$$

Lemma 1 *A term M is canonical iff there is no term M' such that $M \rightarrow M'$. The relation $\rightarrow$ is a partial computable function on terms, whose domain is thus the noncanonical terms.*

Definition 4 *Let $Eval(M)$ be the necessarily unique term C, if any, such that $M \rightarrow^* C$.*

Comments:
Eval is a partial computable function on terms whose range is the set of canonical terms.

If $M \rightarrow N$, then $[\![M]\!] = [\![N]\!]$. Hence, $[\![Eval(M)]\!] = [\![M]\!]$, and $[\![M]\!]\rho_\perp \neq \perp$ whenever $Eval(M)$ is defined.

A.4 The Adequacy Theorem

Inclusive Predicate Specification

Let $[\![\tau]\!]$ be the semantic domain (cpo) corresponding to type τ, and let Λ_τ be the set of (possibly open) terms of type τ.

Definition 5 *Let $\rightsquigarrow$ be a binary relation relation between canonical terms, defined (by structural induction) as follows:*

$$
\begin{array}{rcl}
pair(C,D) \rightsquigarrow pair(C',D') & \textit{iff} & C \rightsquigarrow C' \textit{ and either } D \equiv D' \textit{ or } D \to D' \\
pair(D,C) \rightsquigarrow pair(D',C') & \textit{iff} & C \rightsquigarrow C' \textit{ and either } D \equiv D' \textit{ or } D \to D' \\
pair(C_1,C_2) \rightsquigarrow pair(C_1',C_2') & \textit{iff} & C_1 \rightsquigarrow C_1', C_2' \rightsquigarrow C_2' \\
inL(C) \rightsquigarrow inL(C') & \textit{iff} & C \rightsquigarrow C' \\
inR(C) \rightsquigarrow inR(C') & \textit{iff} & C \rightsquigarrow C' \\
abs(C) \rightsquigarrow abs(C') & \textit{iff} & C \rightsquigarrow C' \\
lift(M) & \rightsquigarrow & lift(M)
\end{array}
$$

Definition 6 *Let the set of* fully canonical *terms be defined as follows:*

$$F ::= pair(F,M) \mid pair(M,F) \mid pair(F,F) \mid inL(F) \mid inR(F) \mid lift(N) \mid abs(F)$$

where $[\![M]\!]\rho_\perp = \perp$.

Observe that every fully canonical term is canonical.

Inclusive binary predicates $\sim_\tau$ on $[\![\tau]\!] \times \Lambda_\tau$ will be defined below to satisfy the properties (A), (B) below. We first define auxiliary binary predicates Π_τ.

Definition 7 *Let* Π_τ *be a binary predicate on* $[\![\tau]\!] \times \Lambda_\tau$ *defined to be identically true for types* τ *that are not observed, and*

$$c \,\Pi_\sigma M \quad \textit{iff} \quad c \sqsubseteq [\![M]\!]\rho_\perp \textit{ and } (c \neq \perp_\sigma \textit{ implies } \exists F.\ Eval(M) \rightsquigarrow^* F).$$

Property (A) (of a relation $\sim_\tau$):

$c \sim_\tau M$ only if $c \,\Pi_\tau M$. If $c \,\Pi_\tau M$, then

$$
\begin{array}{rcl}
c \sim_{\tau_1 \to \tau_2} M & \text{iff} & e \sim_{\tau_1} N \text{ implies } c(e) \sim_{\tau_2} MN \\
c \sim_{\tau_1 \times \tau_2} M & \text{iff} & fst(c) \sim_{\tau_1} fst(M) \text{ and } snd(c) \sim_{\tau_2} snd(M) \\
c \sim_{\tau_1 \oplus \tau_2} M & \text{iff} & outL(c) \sim_{\tau_1} outL(M) \text{ and } outR(c) \sim_{\tau_2} outR(M) \\
c \sim_{\tau_\perp} M & \text{iff} & drop(c) \sim_\tau drop(M) \\
c \sim_{\mu t.\tau} M & \text{iff} & rep(c) \sim_{[\mu t.\tau / t]\tau} rep(M)\ .
\end{array}
$$

Now to define Property (B), call a pair $\langle u, U \rangle$ *ok*, where u is a function between domains and U is a function between correspondingly typed terms, if u and U are related in one of the following ways:

$$
\begin{array}{rcl@{\quad}l@{\quad}rcl}
u & = & \lambda d \in [\![\tau_1 \to \tau_2]\!].\ d(e) & \text{and} & U & = & \lambda M^{\tau_1 \to \tau_2}.\ MN \text{ for some } e \sim_{\tau_1} N, \text{ or} \\
u & = & \lambda d \in [\![\tau_1 \times \tau_2]\!].\ fst(d) & \text{and} & U & = & \lambda M^{\tau_1 \times \tau_2}.\ fst(M), \text{ or} \\
u & = & \lambda d \in [\![\tau_1 \times \tau_2]\!].\ snd(d) & \text{and} & U & = & \lambda M^{\tau_1 \times \tau_2}.\ snd(M), \text{ or} \\
u & = & \lambda d \in [\![\tau_1 \oplus \tau_2]\!].\ outL(d) & \text{and} & U & = & \lambda M^{\tau_1 \oplus \tau_2}.\ outL(M), \text{ or} \\
u & = & \lambda d \in [\![\tau_1 \oplus \tau_2]\!].\ outR(d) & \text{and} & U & = & \lambda M^{\tau_1 \oplus \tau_2}.\ outR(M) \text{ or} \\
u & = & \lambda d \in [\![\tau_\perp]\!].\ drop(d) & \text{and} & U & = & \lambda M^{\tau_\perp}.\ drop(M) \text{ or} \\
u & = & \lambda d \in [\![\mu t.\tau]\!].\ rep(d) & \text{and} & U & = & \lambda M^{\mu t.\tau}.\ rep(M).
\end{array}
$$

A sequence $\langle u_1, U_1\rangle, \ldots, \langle u_m, U_m\rangle$ is *ok* if each pair is ok, and $u_i \circ u_{i+1}$ is type-correct for $i < m$. Now property (B) is that:

> if $(u_1(\cdots u_m(c)\cdots)\ \ \Pi_{\tau_m}(U_1(\cdots U_m(M)\cdots)$ for all $m \geq 0$ and ok sequences $\langle u_1, U_1\rangle, \ldots, \langle u_m, U_m\rangle$, then $c \sim_\tau M$.

Summary of Proof: Using properties (A), (B), show by induction on the structure of M:

Lemma 2 *Let $x_1 : \tau_1, \ldots, x_k : \tau_k$ for some $k \geq 0$ be the free variables of M^τ. If $e_i \sim_{\tau_i} N_i$ for $1 \leq i \leq k$, then $[\![M]\!](\rho[x_i := e_i]) \sim_\tau [N_i/x_i]M$.*

From property (B) of $\sim_\tau$ it follows that $\perp_\tau \sim_\tau M$, for every M. Thus, applying Lemma 2 with $e_i = \perp_{\tau_i}$, $N_i \equiv x_i$, we obtain

Corollary 1 $[\![M^\tau]\!]\rho_\perp \sim_\tau M^\tau$.

Theorem 1 (Adequacy) *For all observed types σ, $Eval(M^\sigma)$ is defined iff $[\![M]\!]\rho_\perp \neq \perp_\sigma$.*

A.5 Construction of the Inclusive Predicates

Let τ be a type expression with free type variables $t_1, \ldots, t_k$. Interpret τ as a function $[\![\tau]\!]$ of k arguments from cpo's to cpo's; if τ is closed, *i.e.*, a type, then interpret $[\![\tau]\!]$ to be a cpo as usual.

We will define a function $P(\tau)$ of k arguments, where the i^{th} argument is a binary predicate p_i on $[\![\tau_i]\!] \times \Lambda_{\tau_i}$, and $P(\tau)(p_1, \ldots, p_k)$ is a binary predicate on $([\![\tau]\!]([\![\tau_1]\!], \ldots, [\![\tau_k]\!])) \times \Lambda_{[\tau_i/t_i]\tau}$.

The definition is by induction on the structure of τ. We write $\mathbf{p}$ as an abbreviation of $p_1, \ldots, p_k$; also, we abbreviate $\Pi_{[\tau_i/t_i]\tau}$ as Π_τ. Now $dP(\tau)(\mathbf{p})M$ only if $d\,\Pi_\tau M$. If $d\,\Pi_\tau M$, then

$$
\begin{array}{rcl}
d\ P(\tau_1 \to \tau_2)(\mathbf{p})\ M & \text{iff} & e\ P(\tau_1)(\mathbf{p})\ N \text{ implies } d(e)\ P(\tau_2)(\mathbf{p})\ MN \\
d\ P(\tau_1 \times \tau_2)(\mathbf{p})\ M & \text{iff} & fst(d)P(\tau_1)(\mathbf{p})fst(M) \text{ and } snd(d)P(\tau_2)(\mathbf{p})snd(M) \\
d\ P(\tau_1 \oplus \tau_2)(\mathbf{p})\ M & \text{iff} & outL(d)P(\tau_1)(\mathbf{p})outL(M) \text{ and } outR(d)P(\tau_2)(\mathbf{p})\ outR(M) \\
d\ P(\tau_\perp)(\mathbf{p})\ M & \text{iff} & drop(d)\ P(\tau)(\mathbf{p})\ drop(M) \\
d\ P(t_i)(\mathbf{p})\ M & \text{iff} & d\ p_i\ M\,.
\end{array}
$$

To complete the inductive definition of $P(\tau)$, we have to describe the remaining case $\mu t.\tau$. Let the free variables of τ be $t, t_1, \ldots, t_k$. We will use the following notation:

$[\![\mu t.\tau]\!] = \bigsqcup_{n\geq 0} [\![\tau]\!]_n$, (*cf.* [26,33,21]) where $[\![\tau]\!]_0(A_1, \ldots, A_k) = \perp$, and
$[\![\tau]\!]_{n+1}(A_1, \ldots, A_k) = [\![\tau]\!]([\![\tau]\!]_n(A_1, \ldots, A_k),\ A_1, \ldots, A_k)$.

Also, for $d \in [\![\mu t.\tau]\!](A_1, \ldots, A_k)$, let $([\![\tau]\!]_n(A_1, \ldots, A_k) \downarrow d)$ be the projection of d on $[\![\tau]\!]_n(A_1, \ldots, A_k)$.

We will now describe the case $\mu t.\tau$ of the inductive definition of $P(\tau)$.

$$d\ P(\mu t.\tau)(\mathbf{p})\ M \text{ iff } ([\![\tau]\!]_n([\![\tau_1]\!],\ldots,[\![\tau_k]\!]) \downarrow d)\ P(\mu t.\tau)_n(\mathbf{p})\ M, \text{ for all } n \geq 0,$$

where the predicates $P(\mu t.\tau)_n$ are defined (by induction on n) as follows:

$$\begin{aligned} d_0\ P(\mu t.\tau)_0(\mathbf{p})\ M &\quad \text{iff} \quad d_0\ \Pi_{\mu t.\tau} M, \\ d_{n+1}\ P(\mu t.\tau)_{n+1}(\mathbf{p})\ M &\quad \text{iff} \quad d_{n+1}\ \Pi_{\mu t.\tau} M \text{ and } d_{n+1}\ P(\tau)(P(\mu t.\tau)_n(\mathbf{p}),\mathbf{p})\ rep(M). \end{aligned}$$

Lemma 3 *If $p_1,\ldots,p_k$ satisfy (A), (B), then $P(\tau)(p_1,\ldots,p_k)$ satisfies (A), (B).*

Theorem 2 (Inclusive Predicate Existence) *For every type τ, $P(\tau)$ satisfies (A), (B).*

A.6 Full Abstraction and Universality

Lemma 4 *For every type τ, every finite (i.e., isolated) element in $[\![\tau]\!]$ equals $[\![M]\!]\rho_\perp$ for some closed term M^τ.*

Corollary 2 *Suppose $[\![M_0]\!]\rho \neq [\![M_1]\!]\rho$, for some valuation ρ. Then there is a context $C[\cdot]$ such that $C[M_0]$ and $C[M_1]$ are closed terms of observed type, and exactly one of $[\![C[M_1]]\!]$ and $[\![C[M_2]]\!]$ equals $\perp$.*

Theorem 3 (Full Abstraction) *Semantic equality of terms coincides with observational congruence.*

Theorem 4 (Universality) *Augment the language by adding $\exists^{(int\to bool)\to bool}$ (the continuous version of the existential quantifier). If $\delta \in [\![\tau]\!]$ is the lub of a recursively enumerable sequence of finite elements, then there is a closed term M^τ such that $[\![M]\!] = \delta$.*

The proofs of full abstraction and universality are simple extensions of Plotkin's [25].
Comment: We cannot have strict pairing in the Adequacy Theorem 1 without committing ourselves to observing nonbottomness at all types: a term M^τ, where τ is arbitrary, is nonbottom iff the term $stfst(stpair(a^{\mathbf{1}}, M^\tau))$ of (observed) type $\mathbf{1}$ is nonbottom.

B Complete Adequacy without Sums

B.1 Syntax of Types

Let t stand for a type variable, τ a type expression, and ν for a nontrivial type expression:

$$\begin{aligned} \tau &::= t \mid \tau \to \tau \mid \tau \times \tau \mid \tau \otimes \tau \mid \tau_\perp \mid \mu t.\tau \\ \nu &::= \tau_\perp \mid \tau \to \nu \mid \nu \times \tau \mid \tau \times \nu \mid \nu \otimes \nu \mid \mu t.\nu \end{aligned}$$

Comment: Strict pairing ($\otimes$) has been included this time.
Example: The type $triv = \mu t.\, t$ is *not* nontrivial.

Lemma 5 *An arbitrary type, τ, is nontrivial iff $[\![\tau]\!] \neq \{\bot\}$.*

B.2 Terms and Typing Rules

The rules are as in Section A.2, with the omission of typing rules for $\oplus$, and the addition of:

$$stpair(M^{\tau_1}, N^{\tau_2}) \ : \ \tau_1 \otimes \tau_2, \qquad stfst(M^{\tau_1 \otimes \tau_2}) \ : \ \tau_1, \qquad stsnd(M^{\tau_1 \otimes \tau_2}) \ : \ \tau_2.$$

Canonical Terms:

$$\begin{array}{rcl} V & ::= & x \mid VM^{\tau} \mid fst(V) \mid snd(V) \mid stfst(V) \mid stsnd(V) \mid drop(V) \mid rep(V) \\ C & ::= & V^{\nu} \mid \lambda x.C \mid pair(C, M) \mid pair(M, C) \mid stpair(C, C) \mid stfst(stpair(C, C)) \mid \\ & & stsnd(stpair(C, C)) \mid lift(M) \mid abs(C) \end{array}$$

where V^{ν} must be of *nontrivial* type.
Comment: If C is canonical, then $[\![C]\!]\rho \neq \bot$, for some valuation ρ.

B.3 Operational Rules

$$\frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

$$\begin{array}{rcl} (\lambda x.M)N & \to & [N/x]M \\ stfst(stpair(M_1, M_2))N & \to & stfst(stpair(M_1 N, M_2)) \\ stsnd(stpair(M_1, M_2))N & \to & stsnd(stpair(M_1, M_2 N)) \end{array}$$

$$\frac{M \to M',\ M \not\equiv \lambda(\cdots),\ M \not\equiv stfst(stpair\cdots),\ M \not\equiv stsnd(stpair\cdots)}{MN \to M'N}$$

$$\frac{M \to M',\ N \to N'}{pair(M, N) \to pair(M', N')}$$

$$\begin{array}{rcl} fst(pair(M, N)) & \to & M \\ snd(pair(M, N)) & \to & N \\ fst(stfst(stpair(M, N))) & \to & stfst(stpair(fst(M), N)) \\ snd(stfst(stpair(M, N))) & \to & stfst(stpair(snd(M), N)) \\ fst(stsnd(stpair(M, N))) & \to & stsnd(stpair(M, fst(N))) \\ snd(stsnd(stpair(M, N))) & \to & stsnd(stpair(M, snd(N))) \end{array}$$

$$\frac{M \to M',\ M \not\equiv pair(\cdots),\ M \not\equiv stfst(stpair\cdots),\ M \not\equiv stsnd(stpair\cdots)}{fst(M) \to fst(M'),\ snd(M) \to snd(M')}$$

$$\frac{M \to M',\ N \to N'}{stpair(M, N) \to stpair(M', N')}$$

$$\frac{M \to M'}{stpair(M,C) \to stpair(M',C),\ stpair(C,M) \to stpair(C,M')}$$

$$\begin{array}{rcl} stfst(stfst(stpair(M,N))) & \to & stfst(stpair(stfst(M),N)) \\ stsnd(stfst(stpair(M,N))) & \to & stfst(stpair(stsnd(M),N)) \\ stfst(stsnd(stpair(M,N))) & \to & stsnd(stpair(M,stfst(N))) \\ stsnd(stsnd(stpair(M,N))) & \to & stsnd(stpair(M,stsnd(N))) \end{array}$$

$$\frac{M \to M',\ M \not\equiv stfst(stpair\cdots),\ M \not\equiv stsnd(stpair\cdots)}{stfst(M) \to stfst(M'),\ stsnd(M) \to stsnd(M')}$$

$$\begin{array}{rcl} drop(lift(M)) & \to & M \\ drop(stfst(stpair(M,N))) & \to & stfst(stpair(drop(M),N)) \\ drop(stsnd(stpair(M,N))) & \to & stsnd(stpair(M,drop(N))) \end{array}$$

$$\frac{M \to M',\ M \not\equiv lift(\cdots),\ M \not\equiv stfst(stpair\cdots),\ M \not\equiv stsnd(stpair\cdots)}{drop(M) \to drop(M')}$$

$$up?\ lift(M)\ N \to N$$

$$\frac{M \to M',\ M \not\equiv lift(\cdots)}{up?\ M\ N \to up?\ M'\ N}$$

$$\frac{M \to M'}{abs(M) \to abs(M')}$$

$$\begin{array}{rcl} rep(abs(M)) & \to & M \\ rep(stfst(stpair(M,N))) & \to & stfst(stpair(rep(M),N)) \\ rep(stsnd(stpair(M,N))) & \to & stsnd(stpair(M,rep(N))) \end{array}$$

$$\frac{M \to M',\ M \not\equiv abs(\cdots),\ M \not\equiv stfst(stpair\cdots),\ M \not\equiv stsnd(stpair\cdots)}{rep(M) \to rep(M')}$$

(if no rule above applies to M and M not canonical)

$$M \to M$$

Definition 8 *$Eval(M)$ as in Appendix A.*

B.4 The Complete Adequacy Theorem

For every valuation ρ, define binary predicates Π^ρ_τ on $[\![\tau]\!] \times \Lambda_\tau$ by the rule $c\,\Pi^\rho_\tau M$ iff

$c \sqsubseteq [\![M]\!]\rho$ and
($c \neq \bot_\tau$ implies $Eval(M)$ is defined).

As in Appendix A, the binary predicates $\sim^\rho_\tau$ on $[\![\tau]\!] \times \Lambda_\tau$ satisfy corresponding properties (A), (B) expressed in terms of the predicates Π^ρ_τ).

Using properties (A), (B) of the predicates $\sim^\rho_\tau$, we show by induction on the structure of M:

Lemma 6 *For variables $x_1 : \tau_1, \ldots, x_k : \tau_k$, where $k \geq 0$, if $e_i \sim^\rho_{\tau_i} N_i$ for $1 \leq i \leq k$, then $[\![M^\tau]\!](\rho[x_i := e_i]) \sim^\rho_\tau [N_i/x_i]M$.*

Corollary 3 *$[\![M]\!]\rho \sim^\rho_\tau M$ for all M^τ, ρ.*

Theorem 5 (Complete Adequacy) *Eval(M) is defined iff $\exists\rho.\ [\![M]\!]\rho \neq \bot$.*

Remark: For any term M, the meaning of the lambda closure of M is nonbottom iff the meaning of M is nonbottom in *some* valuation.

The construction of the inclusive predicates $\sim^\rho_\tau$ is as in Appendix A, simply replacing Π_τ by Π^ρ_τ.

B.5 Discussion

Observe that, without sum types, every pair of values is consistent (*i.e.*, they have a common upper bound), and consequently all definable types happen to be lattices even under the cpo interpretation. This is crucial for our complete adequacy theorem. The presence of inconsistent pairs in the semantic domains together with a strict pairing operator complicates the problem of observing nonbottomness at function types. For example, in the cpo semantics, the term $\lambda x.stpair(outL(xM), outR(xN))$ is nonbottom iff there is a valuation ρ such that $[\![M]\!]\rho$ and $[\![N]\!]\rho$ are inconsistent.

The sum type-constructor over cpo's introduces types with inconsistent elements; moreover, even in the absence of strict pairs, sum types involve similar connections between nonbottomness and inconsistency. We conjecture that our complete adequacy result can be extended to the language with sum types with a parallel conditional, if our semantic domains are *lattices*. We're still wondering about sums in the cpo case.

Since we do not have a conditional in the language of this appendix, the isolated elements are not all definable and we cannot prove full abstraction following Plotkin. However, different methods (based on Böhm trees) have been used to prove such results for the untyped lambda calculus, *cf.* [1], and we expect such methods are also applicable in our case.

References

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

[2] G. Berry, P. Curien, and J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 3, pages 89–132, Cambridge Univ. Press, 1985.

[3] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced: preliminary report. In 15^{th} *Symp. Principles of Programming Languages*, pages 229–239, ACM, 1988.

[4] M. Felleisen, D. Friedman, E. Kohlbecker,

and B. Duba. Reasoning with continuations. In *Symp. Logic in Computer Sci.*, pages 131–141, IEEE, 1986.

[5] M. Felleisen, D. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Sci.*, 52:205–237, 1987.

[6] J. H. Gallier. The semantics of recursive programs with function parameters of finite types: n-rational algebras and logic of inequalities. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 9, pages 313–362, Cambridge Univ. Press, 1985.

[7] J. Girard. The system F of variable types, fifiteen years later. *Theoretical Computer Sci.*, 45:152–192, 1986.

[8] M. J. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1979.

[9] I. Guessarian. *Algebraic Semantics.* Volume 99 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1981.

[10] I. Guessarian. Survey on some classes of interpretations and some of their applications. *SIGACT News*, 15(3):45–71, 1983.

[11] C. Gunter. *A Universal Domain Technique for Profinite Posets.* Technical Report CMU-CS-85-142, Carnegie-Mellon Univ., 1985.

[12] C. A. Gunter and A. Jung. *Coherence and Consistency in Domains (Extended Outline).* Technical Report MS-CIS-88-20, Dept. of Computer and Information Science, Univ. Pennsylvania, 1988.

[13] J. Y. Halpern, A. R. Meyer, and B. A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In 11^{th} *Symp. on Principles of Programming Languages*, pages 245–257, ACM, 1984.

[14] M. C. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming language. In *Math. Found. Computer Science, Proc.*, pages 108–120, Volume 74 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1979.

[15] J. Loeckx and K. Sieber. *The Foundations of Program Verification. Wiley-Teubner Series in Computer Science*, John Wiley and Sons, 1984.

[16] Z. Manna. *Mathematical Theory of Computation.* McGraw Hill, 1974.

[17] A. R. Meyer. 313 lines about full abstraction from Stoughton and Meyer. 1988. 14 May. Communication from `meyer@theory.lcs.mit.edu` to the TYPES electronic forum, internet: `types-request-@theory.lcs.mit.edu`.

[18] A. R. Meyer and J. G. Riecke. Continuations may be unreasonable. In *Proc. of Conf. LISP and Functional Programming*, ACM, July 1988. To appear.

[19] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In 15^{th} *Symp. Principles of Programming Languages*, pages 191–203, ACM, 1988.

[20] R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[21] P. D. Mosses and G. D. Plotkin. On proving limiting completeness. *SIAM J. Computing*, 16:179–194, 1987.

[22] K. Mulmuley. *Full Abstraction and Semantic Equivalence. ACM Doctoral Dissertation Award 1986*, MIT Press, 1987.

[23] F. J. Oles. Type algebras, functor categories, and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*, chapter 15, pages 544–573, Cambridge Univ. Press, 1985.

[24] G. D. Plotkin. A powerdomain construction. *SIAM J. Computing*, 5:452–487, 1976.

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

[26] G. D. Plotkin. $\mathbf{T}^{\omega}$ as a universal domain. *J. Computer and System Sci.*, 17:209–236, 1978.

[27] G. D. Plotkin. *A structural approach to operational semantics.* Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[28] J. Raoult and J. Vuillemin. Operational and semantic equivalence between recursive programs. *J. ACM*, 27:772–796, 1980.

[29] J. C. Reynolds. On the relation between direct and continuation semantics. In *Proc. 2^{nd} Coll. on Automata, Languages, and Programming*, pages 141–156, Volume 14 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1974.

[30] V. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra i Logika*, 15:308–330, 1976. (Russian).

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

[32] D. Scott. Domains for denotational semantics. In M. Nielson and E. Schmidt, editors, *9^{th} Int'l. Coll. on Automata, Languages and Programming*, pages 577–613, Volume 140 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1982.

[33] M. Smyth and G. D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing*, 11:761–783, 1982.

[34] R. Statman. Equality between functionals revisited. In L. A. Harrington, *et al.*, editor, *Harvey Friedman's Research on the Foundations of Mathematics*, pages 331–338, Volume 117 of *Studies in Logic*, North-Holland, 1985.

[35] R. Statman. Logical relations in the typed λ-calculus. *Information and Control*, 65:86–97, 1985.

[36] A. Stoughton. *Fully Abstract Models of Progamming Languages. Research Notes in Theoretical Computer Science*, Pitman/Wiley, 1988. Revision of Ph.D thesis, Dept. of Computer Science, Univ. Edinburgh, Report No. CST-40-86, 1986.

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

[38] J. E. Stoy. The congruence of two programming language definitions. *Theoretical Computer Sci.*, 13:151–174, 1981.

[39] C. Strachey. Fundamental concepts in programming languages. 1967. Lecture Notes, Int'l. Summer School in Computer Programming, Copenhagen.

[40] P. Taylor. *Recursive Domains, Indexed Category Theory, and Polymorphism.* Ph.D. thesis, Trinity College, Cambridge, Aug. 1986.

[41] P. Taylor. L-domains. 1988. 20 January.

Communication from `mcvax!doc.ic.ac.uk!pt@uunet.uu.net` to the TYPES electronic forum, `internet: types-request@theory.lcs.mit.edu`.

[42] B. A. Trakhtenbrot, J. Y. Halpern, and A. R. Meyer. From denotational to operational and axiomatic semantics for ALGOL-like languages: an overview. In E. Clarke and D. Kozen, editors, *Logic of Programs, Proceedings 1983*, pages 474–500, Volume 164 of *Lect. Notes in Computer Sci.*, Springer-Verlag, 1984.

[43] C. Wadsworth. The relation between computational and denotational properties for Scott's D_∞ models. *SIAM J. Computing*, 5:488–521, 1976.

Cambridge, Massachusetts
June 30, 1988