Reasoning with Continuations II: Full Abstraction for Models of Control

Dorai Sitaram and Matthias Felleisen*
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

A fully abstract model of a programming language assigns the same meaning to two terms if and only if they have the same operational behavior. Such models are well-known for functional languages but little is known about extended functional languages with sophisticated control structures. We show that a direct model with error values and the conventional continuation model are adequate for functional languages augmented with first- and higher-order control facilities, respectively. Furthermore, both models become fully abstract on adding a control delimiter and a parallel conditional to the programming languages.

1 The Control Structure of Lisp

Many programming languages, in particular Lisp [17, 24] and Scheme [22, 26], contain sophisticated control structures with constructs for labeling the control state at an arbitrary point for later reuse. Since Lisp is an expression-oriented language, a jump to a labeled control state—just like a procedure—may also pass a parameter so that it is possible to communicate results between control points. In most Lisp systems, control points have a special status, distinct from all other values. The Scheme dialect treats its control points—referred to as *continuations*—as ordinary, first-class procedural objects and even uses the same syntax for continuation invocations and procedure calls.

For many programming tasks, control facilities are an important means of abstraction. They avoid the cumbersome encoding of control mechanisms through boolean predicates and tests, and explicate the purpose of otherwise unreadable programming patterns. Moreover, by incorporating explicit statements about the control behavior, a program provides clues to the compiler for possible optimizations. However, control constructs also invalidate some common laws of reasoning that hold for functional programming languages like pure Lisp [6, 18, 27]. More technically, the operational equivalence relation for the functional subset of Lisp is not a subset of the operational equivalence relation for Lisp with control facilities. Given the importance of operational equivalence laws for reasoning about programs, this raises two crucial questions:

1. How can we reason with control facilities?

2. How can we relate the reasoning system for functional languages to the system for extended languages with control facilities?

Talcott's [27, 28] and our own [6, 7, 11] results on functional languages with first-class control abstractions provide a partial answer to the first question.[1] There is a small number of fundamental laws for the operational equivalence relation of such languages that play a crucial role in equivalence

*The work of both authors was partially supported by NSF and DARPA.

[1]The recent work by Aiken et al [1] on program transformations in the presence of errors can also be viewed as addressing this problem although their main concern is quite different.

proofs of Lisp-like programs with continuations. Based on these laws, it is possible to manipulate such programs with the same algebraic elegance as functional programs. Our work [6, 7, 11] on conservative extensions of Plotkin's [20] λ_v-calculus for pure Lisp also contributes to an answer for the second question. It shows that all equational proofs in the *syntactic* theory of functional languages carry over to the extended framework with control mechanisms.

Unfortunately, both pieces of research are only incomplete answers to the above questions. Reasoning directly about operational equivalence is a powerful approach but suffers from the sensitivity of the relation to language extensions. This approach also ignores the existing mathematical models for functional languages and the potential for exploiting traditional mathematical techniques for reasoning about programs. Our extended λ_v-calculus, on the other hand, is necessarily a weak reasoning tool because it accomodates language extensions. It particularly lacks an induction principle, which automatically comes with denotational models.

In this paper we present another step toward a reasoning system for languages with control structure. Following Plotkin's [21] approach, we analyze extensions of a typed functional language with Lisp-style control facilities and their natural models. Adjusting the programming languages so that these models become fully abstract leads to the surprising re-discovery of first-class prompts: We had earlier suggested the inclusion of prompts in languages with control structures but for entirely different reasons [8, 23]. The full abstraction theorem for the conventional continuation model also verifies a conjecture by Meyer and Riecke [18:65].

The following section introduces PCF$_\text{V}$-C, an extension of a call-by-value variant of PCF [21] with a sophisticated control structure. It also contains a formal discussion on the above-mentioned problematic relationship between the operational equivalence relations of PCF$_\text{V}$ and the extended languages, providing a comprehensive overview of the expressive power of the various control constructs [9]. The third section describes the properties of a denotational semantics for PCF$_\text{V}$ and the related notions of denotational equivalence, adequacy and full abstraction. The fourth and fifth sections present direct and continuation models for first- and higher-order control structures, respectively. In both cases, full abstraction fails for the original languages, but language extensions can fix both instances of the problem. The last section interprets our results with respect to programming language design.

2 PCF$_\text{V}$-C

PCF$_\text{V}$-C is an extension of call-by-value PCF [21] with control facilities. PCF$_\text{V}$, the programming language of computable functions, is a variant of the typed λ-value-calculus with ground constants, conditionals, and fixpoint combinators.

Figure 1 describes the types and syntax of PCF$_\text{V}$. Subscripts on terms indicate the types of the terms; superscripts indicate the types of variables. The language includes the *numeral* $\ulcorner n \urcorner$ for each natural number n. As usual, λ-abstractions introduce *bound* variables; those variables in a term that are not bound are *free*. A term with no free variables is *closed*. PCF$_\text{V}$ *programs* are closed PCF$_\text{V}$ terms of ground type. The notation $M[x{:=}N]$ denotes the *substitution* of all occurrences of a free variable x in M with the term N. Finally, a *context* $C[\ ]$ is a term with a "hole" where a subterm should be; $C[M]$ is the term where the hole is filled with a subterm M of appropriate type, with the context possibly capturing free variables in M. We adopt the usual conventions governing the use of concrete syntax for λ-expressions [2].

Figure 1 also specifies the primitive transition rules ($\longrightarrow$) of an abstract machine for PCF$_\text{V}$. The numerals represent numbers. The functional constants $\mathsf{1+}$ and $\mathsf{1-}$ have the behavior of "add 1" and "subtract 1." A *thunk* is a procedure with a dummy argument: we use $\ulcorner 0 \urcorner$ to discharge a thunk. The constant ef is a conditional function. The constant $\ulcorner 0 \urcorner$ also acts as *false*, every other ground value being *true*. The *form* **if** is syntactic sugar relative to ef based on thunks:

$$\textbf{if}\ M\ N\ P \equiv (\mathsf{ef}\ M\ (\lambda d.N)\,(\lambda d.P))\ulcorner 0 \urcorner,$$

where d is not free in N, P. The combinators $\mathbf{Y}^{s \to t}$ create recursive functions of type $s \to t$.

In contrast to the original PCF, the abstractions in our variant are call-by-value and evaluation proceeds left-to-right. On evaluation, a program is searched for a *redex*, i.e., a subterm that matches the left-hand side of some primitive machine rule.

Types:

$$\begin{array}{rcll} t & ::= & \iota & \text{(ground type)} \\ & | & t \to t & \text{(procedure type)} \end{array}$$

Syntax:

$$\begin{array}{rcll} M & ::= & V \mid (M_{s\to t} M_s)_t & \text{(terms)} \\ V & ::= & \mathsf{c} \mid x^t \mid (\lambda x^s.M_t)_{s\to t} & \text{(values)} \\ \mathsf{c} & ::= & \mathsf{b} \mid \mathsf{f} & \text{(constants)} \\ \mathsf{b} & ::= & \ulcorner n \urcorner_\iota & \text{(basic constants)} \\ \mathsf{f} & ::= & \mathsf{1+}_{\iota\to\iota} \mid \mathsf{1-}_{\iota\to\iota} \mid \mathsf{ef}_{\iota\to s\to s\to s} \mid \mathbf{Y}^{s\to t}_{((s\to t)\to(s\to t))\to(s\to t)} & \text{(functional constants)} \end{array}$$

Behavior:

$$\begin{array}{rclrclc} \mathsf{1+}\,\ulcorner n \urcorner & \longrightarrow & \ulcorner n+1 \urcorner & \mathsf{ef}\,\ulcorner n+1 \urcorner & \longrightarrow & \lambda x,y.x & (\lambda x.M)V \longrightarrow M[x{:=}V] \\ \mathsf{1-}\,\ulcorner n+1 \urcorner & \longrightarrow & \ulcorner n \urcorner & \mathsf{ef}\,\ulcorner 0 \urcorner & \longrightarrow & \lambda x,y.y & \mathbf{Y}V \longrightarrow \lambda x.V(\mathbf{Y}V)x \end{array}$$

FIGURE 1: Call-by-value PCF_V.

This search is guided by recognizing an *evaluation context*, a special kind of context, given by the following grammar:

$$E[\,] ::= [\,] \mid V E[\,] \mid E[\,]M.$$

The appropriate machine rule transforms the redex, giving a new program. This single evaluation *step* is defined as

$$E[M] \vartriangleright E[M'], \text{ if } M \longrightarrow M'.$$

The stepping procedure is continued until the program reduces to a ground constant. The evaluation process is abstracted by the partial function *eval* from programs to ground constants:

$$\mathit{eval}(M) = \mathsf{b} \text{ iff } M \vartriangleright^* \mathsf{b}.$$

Programs that fail to yield a value are called *non-terminating* or *diverging*. For any value V, $\Omega \equiv (\mathbf{Y}\,(\lambda fx.fx)\,V)$ is a diverging term.

A starting point for reasoning with terms is the ability to identify language terms that have the same behavior.

Definition 2.1 *For* PCF_V *terms* M *and* N *of type* t, M operationally approximates N, $M \sqsubseteq_v N$, *if for all contexts* $C[\;]$, $\mathit{eval}(C[M]) = \mathsf{b}$ *implies* $\mathit{eval}(C[N]) = \mathsf{b}$. M *is* operationally equivalent *to* N, $M \simeq_v N$, *if* $M \sqsubseteq_v N$ *and* $N \sqsubseteq_v M$.

For an example of operationally equivalent terms, consider $M_1 \equiv \lambda f.\Omega$ and $M_2 \equiv \lambda f.f\ulcorner 0 \urcorner\Omega$. Take any program context $C[\,]$ enclosing M_i. The evaluation of the program may never place M_i in a function position, in which case the programs $C[M_1]$ and $C[M_2]$ clearly cannot give different results. If M_1 shows up in a function position, then so does M_2 and both programs are guaranteed to diverge. Thus the two terms M_i are interchangeable in any program context, and hence $M_1 \simeq_v M_2$.

Our first control extension is the *abort*-expression of the form $\mathcal{A}M$. Many programming languages provide this feature, which goes by various names: error, stop, abort, exit, and halt. An $\mathcal{A}$-expression stops the evaluation of a program and returns the value of its sub-expression as the result of the entire program. Thus, the sub-expression of $\mathcal{A}$ must be of ground type, whereas the type of the $\mathcal{A}$-expression itself is arbitrary:

$$M ::= \ldots \mid (\mathcal{A}M_\iota)_t.$$

We call this extended language $\text{PCF}_\text{V}+\mathcal{A}$. The additional stepping rule is

$$E[\mathcal{A}M] \vartriangleright M.$$

The evaluation function eval_a and the operational equivalence relation $\simeq_a$ for $\text{PCF}_\text{V}+\mathcal{A}$ are defined along the same lines as for PCF_V. However, the addition of $\mathcal{A}$ invalidates some of the operational equivalences in PCF_V, showing that functional languages cannot express *abort* [9].

Proposition 2.2 $\simeq_v \not\subseteq \simeq_a$

Proof. The terms M_1 and M_2 above, which are operationally equivalent in PCF_V, are inequivalent in $\text{PCF}_\text{V}+\mathcal{A}$, since the context $[\;](\lambda x.\mathcal{A}x)$ distinguishes the two. ∎

Our second control extension, the $\mathcal{K}$-expression of the form $\mathcal{K}M$, provides more interesting possibilities for control manipulations than $\mathcal{A}$. The programming language Scheme [22, 26] provides the equivalent of $\mathcal{K}$-expressions with the procedure call-with-current-continuation or call/cc. A limited version of $\mathcal{K}$ is provided by the **goto**-construct and labels of Algol-like languages, the **catch/throw** or **errset/error** mechanism of traditional Lisp systems, and the exception handlers of ML [19].

A $\mathcal{K}$-expression applies its sub-expression to an abstraction of the surrounding context or the *rest of the computation*, the *continuation*. This continuation is a procedural object that can be called anywhere in the program, just like any other procedure. When it is invoked, evaluation reinstates the captured context in place of the current one, and fills it with the continuation's argument. It is easy to simulate other non-local control actions in terms of $\mathcal{K}$ [5, 12, 14, 15, 26].

A $\mathcal{K}$-expression can occur anywhere inside a program; thus it can have any type s. Also, the program can invoke the continuation anywhere, but the latter's argument must be of the same type as the $\mathcal{K}$-expression. Further, the $\mathcal{K}$-subexpression returns a value of the type of the $\mathcal{K}$-expression. Given this, we have the following extension of PCF_V's syntax: [2]

$$M ::= \ldots \mid (\mathcal{K}M_{(s\to t)\to s})_s.$$

Assuming $\mathcal{A}$-expressions in the state transition language, the additional stepping rule for $\mathcal{K}$ is:

$$E[\mathcal{K}M] \rhd E[M\lambda x.\mathcal{A}E[x]].$$

The evaluation function and operational equivalence relation for $\text{PCF}_\text{V}+\mathcal{K}$ are $eval_k$ and $\simeq_k$, respectively. Once again, operational equivalences are not necessarily preserved when PCF_V is extended to $\text{PCF}_\text{V}+\mathcal{K}$.

Proposition 2.3 $\quad \simeq_v \not\subseteq \simeq_k$

Proof. The same counterexample as for Proposition 2.2 holds. The distinguishing context is now $\mathcal{K}(\lambda a.[\;]a)$. ∎

[2] Bruce Duba [private communication] and Tim Griffin [13] independently formulated similar typing rules for $\mathcal{A}$ and $\mathcal{K}$.

The full language $\text{PCF}_\text{V}+\mathcal{A}+\mathcal{K}$, which we shall call PCF_V-C, is also a non-conservative extension of $\text{PCF}_\text{V}+\mathcal{A}$, i.e., $\mathcal{K}$ adds true expressive power [9]. The evaluation function and operational equivalence relation for PCF_V-C are $eval_c$ and $\simeq_c$, respectively.

Proposition 2.4 $\quad \simeq_a \not\subseteq \simeq_c$

Proof. Let

$$\mathsf{equal?} = \lambda xy.\mathbf{if}\ x\ (\mathbf{if}\ y\ \ulcorner 1\urcorner\ \ulcorner 0\urcorner)\ (\mathbf{if}\ y\ \ulcorner 0\urcorner\ \ulcorner 1\urcorner).$$

Consider the terms

$$M_u \equiv \lambda f.\mathsf{equal?}\ (f\lambda k.k\ulcorner 1\urcorner\Omega)\ (f\lambda k.k\ulcorner u\urcorner\Omega),$$

for $u = 0, 1$. The only way these terms can differ is by invoking the variables k without evaluating the Ω's, so that the equality test takes place. An *abort* could be used to elude the Ω, but the first argument of equal?, which is the same for both terms, is the first and only site where the *abort* may happen. Thus, owing to left-to-right evaluation, the terms are operationally equivalent in $\text{PCF}_\text{V}+\mathcal{A}$. In $\text{PCF}_\text{V}+\mathcal{K}$, on the other hand, the context $[\;](\lambda x.\mathcal{K}x)$ distinguishes them. ∎

Finally, $\mathcal{K}$-expressions provide the power of *abort* in a certain sense. Hence, operational equivalences are not affected by extending $\text{PCF}_\text{V}+\mathcal{K}$ to PCF_V-C.

Proposition 2.5 $\quad \simeq_k \subseteq \simeq_c$

Proof. The only extra construct in PCF_V-C is $\mathcal{A}$. From the proof for Proposition 2.3, we note that an $\mathcal{A}$-expression can be simulated in $\text{PCF}_\text{V}+\mathcal{K}$ by enclosing the program with $\mathcal{K}\lambda a.[\;]$, and using the variable a in place of the aborting construct. ∎

3 Models, Adequacy, and Full Abstraction

A denotational model for a language like PCF_V provides a collection of *domains*, an interpretation for the constant symbols, and a meaning function for the language terms. A domain is a partially ordered ($\sqsubseteq$) collection of elements. In subsequent sections, we use one primitive domain $\mathbb{N}_\perp$ and the following domain constructions: *function* domains ($\to$, with functions notated by $\underline{\lambda}$), *strict* function domains ($\to_s$, with functions notated by $\underline{\lambda}^s$), *disjoint* union ($\oplus$), and *lifting* ($\cdot_\perp$). Appendix A gives a brief description of domains, domain constructions, and their salient properties.

Each type t maps to a domain $\mathbb{D}_t$, starting with the ground domain $\mathbb{D}_\iota$. The domain for the arrow type $s \to t$ is built from the domains for s and t using appropriate domain constructions. The interpretation $\mathfrak{A}$ maps the constant terms of the language to elements in the appropriate domains. The semantic functional $\hat{\mathfrak{A}}$, an extension of the interpretation, maps each term of type t to a *denotation*, usually a function that maps some auxiliary arguments to a value in $\mathbb{D}_t$. For example, the semantic function for a *direct* model has only an environment as an auxiliary argument; in the *continuation* model, the arguments include a continuation argument. Finally, the meaning function for programs maps programs to ground values:

$$\hat{\hat{\mathfrak{A}}} : \mathit{Programs} \to \mathbb{D}_\iota.$$

A minimum requirement on a model is that it not contradict and faithfully implement the operational semantics. This property is called *adequacy*.

Definition 3.1 [Adequacy] *A model is* adequate *if for any program M,*

$$\mathit{eval}(M) = \mathsf{b} \text{ iff } \hat{\hat{\mathfrak{A}}}[\![M]\!] = \mathfrak{A}[\![\mathsf{b}]\!].$$

Analogous to operational approximation and equivalence, we can define notions of approximation and equivalence among terms based on their *denotations*.

Definition 3.2 *For terms M and N of type t, M* denotationally approximates *N, $M \sqsubseteq N$, if $\hat{\mathfrak{A}}[\![M]\!] \sqsubseteq \hat{\mathfrak{A}}[\![N]\!]$. M is* denotationally equivalent *to N, $M \equiv N$, if $M \sqsubseteq N$ and $N \sqsubseteq M$.*

A meaning function is *compatible* if $M \sqsubseteq N$ implies $C[M] \sqsubseteq C[N]$ for all contexts C. For such a meaning function, it follows from adequacy that denotationally equivalent terms are also operationally equivalent.

Lemma 3.3 *Let $\hat{\mathfrak{A}}$ be the compatible semantic function of an adequate model. Then, for terms M and N, (1) $M \sqsubseteq N$ implies $M \underset{\sim}{\sqsubset} N$, and (2) $M \equiv N$ implies $M \simeq N$.*

Proof. (1) Suppose $M \sqsubseteq N$, but $M \not\underset{\sim}{\sqsubset} N$. Then there is a context $C[\ ]$ such that $\mathit{eval}(C[M]) = \mathsf{b}$ and $\mathit{eval}(C[N]) \neq \mathsf{b}$. By adequacy, this gives $C[M] \not\sqsubseteq C[N]$, which is a contradiction since, by compatibility, $M \sqsubseteq N$ implies $C[M] \sqsubseteq C[N]$. (2) The second claim follows from the first. ∎

If operational equivalence also implies equivalence in the model, the model is *fully abstract*.

Definition 3.4 [Full Abstraction] *A model is* fully abstract *for a language if for any two terms M, N in the language,*

$$M \underset{\sim}{\sqsubset} N \text{ iff } M \sqsubseteq N.$$

Given that the operational semantics and the model are constructed independently, full abstraction should not be expected automatically.

4 Direct Models for PCF$_{\rm V}$+$\mathcal{A}$

Plotkin [21] showed that a simple extension of call-by-name PCF has a fully abstract direct model. It is part of programming language folklore that this result extends to the call-by-value variant of PCF, PCF$_{\rm V}$, as well as its simple control extension PCF$_{\rm V}$+$\mathcal{A}$. We confirm this by constructing the appropriate models for PCF$_{\rm V}$ and PCF$_{\rm V}$+$\mathcal{A}$.

We begin with the model for PCF$_{\rm V}$ (Figure 2). Its ground domain $\mathbb{D}_\iota$ is the traditional cpo of natural numbers, $\mathbb{N}_\perp$. The arrow type $s \to t$ maps to the *lifted strict* function domain $\mathbb{D}_s \to_s \mathbb{D}_t$. The *lift* operation reflects the fact that there are diverging terms of all types; the *strictness* of the function spaces accounts for the call-by-value behavior of procedures. The interpretation $\mathfrak{A}$ maps the constants of PCF$_{\rm V}$ to their standard denotations, i.e., $\ulcorner 1 \urcorner$ to 1, 1+ to the "add one" function, etc. The semantic functionals $\hat{\mathfrak{A}}_t : \mathit{Terms}_t \to \mathit{Env} \to \mathbb{D}_t$ extend $\mathfrak{A}$ to all PCF$_{\rm V}$-terms. *Env* is the set of environments, which are type-respecting maps from variables to elements in the domains.

It is straightforward to show that this model is *adequate*. This result was also obtained by Breazu-Tannen et al [3] and Riecke [personal communication].

Theorem 4.1 (Adequacy) *For any program M in* PCF$_{\rm V}$,

$$\mathit{eval}(M) = \mathsf{b} \text{ iff } \hat{\hat{\mathfrak{A}}}[\![M]\!] = \mathfrak{A}[\![\mathsf{b}]\!].$$

Domains:

$$\begin{array}{rcl} \mathbb{D}_\iota & = & \mathbb{N}_\perp \\ \mathbb{D}_{s\to t} & = & [\mathbb{D}_s \to_s \mathbb{D}_t]_\perp \end{array}$$

Semantic functions:

$$\begin{array}{rcl}
\hat{\hat{\mathfrak{A}}} & : & \mathit{Programs} \to \mathbb{D}_\iota \\
\hat{\hat{\mathfrak{A}}}[\![M]\!] & = & \hat{\mathfrak{A}}[\![M]\!]\perp \\
\\
\hat{\mathfrak{A}} & : & \mathit{Terms}_t \to \mathit{Env} \to \mathbb{D}_t \\
\hat{\mathfrak{A}}[\![\mathsf{b}]\!]\rho & = & \mathfrak{A}[\![\mathsf{b}]\!] \\
\hat{\mathfrak{A}}[\![x]\!]\rho & = & \rho[\![x]\!] \\
\hat{\mathfrak{A}}[\![\lambda x.M]\!]\rho & = & \mathsf{lift}(\underline{\lambda}^s v.\hat{\mathfrak{A}}[\![M]\!]\rho[x/v]) \\
\hat{\mathfrak{A}}[\![\mathbf{Y}]\!]\rho & = & \mathit{fix}_{\mathrm{cbv}} \\
\hat{\mathfrak{A}}[\![MN]\!]\rho & = & \mathit{apply}(\hat{\mathfrak{A}}[\![M]\!]\rho, \hat{\mathfrak{A}}[\![N]\!]\rho) \\
\\
\mathit{fix}_{\mathrm{cbv}} & : & \mathbb{D}_{((s\to t)\to(s\to t))\to(s\to t)} \\
\mathit{fix}_{\mathrm{cbv}}(\perp) & = & \perp \\
\mathit{fix}_{\mathrm{cbv}}(\mathsf{lift}(f)) & = & \mathit{fix}(f) \text{ where } \mathit{fix}(f) = \bigsqcup_{i=0}^{\infty}\{a_i\} \text{ for } a_0 = \perp, a_{i+1} = \mathit{apply}(f, a_i) \\
\\
\mathit{apply} & : & \mathbb{D}_{s\to t} \times \mathbb{D}_s \to \mathbb{D}_t \\
\mathit{apply}(\perp, v) & = & \perp \\
\mathit{apply}(\mathsf{lift}(f), v) & = & f\,v
\end{array}$$

FIGURE 2: The direct model for call-by-value $\mathrm{PCF_V}$.

Proof. We define two auxiliary functions: $eval_r$ and η. The partial function $eval_r$, from terms and syntactic environments (maps from variables to tuples) to tuples of terms and environments, defines an equivalent operational semantics for $\mathrm{PCF_V}$. The semantic function η on these tuples is then:

$$\begin{array}{rcl} \eta(\mathsf{c};\emptyset) & = & \mathfrak{A}[\![\mathsf{c}]\!], \\ \eta(\lambda x.M; r) & = & f \end{array}$$

where $\mathit{apply}(f, \eta(a))$

$$= \begin{cases} \eta(eval_r(M)r[x/a]) & \\ \quad \text{if } eval_r(M)r[x/a] \text{ converges,} & \\ \perp \quad \text{otherwise.} & \end{cases}$$

We can then show that η satisfies:

$$\hat{\mathfrak{A}}[\![M]\!](\eta \circ r) = \begin{cases} \eta(eval_r(e)r) & \text{if } eval_r(e)r \text{ converges,} \\ \perp & \text{otherwise.} \end{cases}$$

A specialization of this result to complete programs proves the theorem:

1. Assume $eval(M) = \mathsf{b}$. Then, $eval_r(M)\emptyset = \mathsf{b};\emptyset$, and so
$$\hat{\hat{\mathfrak{A}}}[\![M]\!] = \hat{\mathfrak{A}}[\![M]\!]\perp = \mathfrak{A}[\![\mathsf{b}]\!].$$
2. Assume $\hat{\hat{\mathfrak{A}}}[\![M]\!] = \mathfrak{A}[\![\mathsf{b}]\!]$. Then, $\hat{\mathfrak{A}}[\![M]\!]\perp = \mathfrak{A}[\![\mathsf{b}]\!] \neq \perp$. Thus $eval_r(M)\emptyset$ converges, and $\eta(eval_r(M)\emptyset) = \mathfrak{A}[\![\mathsf{b}]\!]$. Now, since $\mathsf{b};\emptyset$ is the only tuple mapped by η to $\mathfrak{A}[\![\mathsf{b}]\!]$, we have $eval_r(M)\emptyset = \mathsf{b};\emptyset$, and $eval(M) = \mathsf{b}$. ∎

From Lemma 3.3, since $\hat{\mathfrak{A}}$ is compatible, it follows that denotationally equivalent call-by-value $\mathrm{PCF_V}$-terms are also operationally equivalent. However, the converse is not true: from Plotkin's treatment of PCF, we can easily derive a counterexample for $\mathrm{PCF_V}$. Consider N_u $(u = 0, 1)$:

$$\begin{array}{l} N_u = \lambda x.\mathbf{if}\ (x\ (\lambda d.\ulcorner 1 \urcorner)\ (\lambda d.\Omega)) \\ \qquad\qquad (\mathbf{if}\ (x\ (\lambda d.\Omega)\ (\lambda d.\ulcorner 1 \urcorner)) \\ \qquad\qquad\quad (\mathbf{if}\ (x\ (\lambda d.\ulcorner 0 \urcorner)\ (\lambda d.\ulcorner 0 \urcorner))\ \Omega\ \ulcorner u \urcorner) \\ \qquad\qquad\quad \Omega) \\ \qquad\qquad \Omega. \end{array}$$

Although $N_0 \simeq_v N_1$, $N_0 \not\equiv N_1$, for applying the denotation of N_u to

$$\underline{por} = \underline{\lambda}^s m, n. \begin{cases} 1 & \text{if } m0 > 0 \text{ or } n0 > 0 \\ 0 & \text{if } m0 = 0 \text{ and } n0 = 0 \\ \perp & \text{otherwise} \end{cases}$$

Domains:

$$
\begin{aligned}
\mathbb{D}'_\iota &= \mathbb{D}_\iota\\
\mathbb{D}'_{s\to t} &= [\mathbb{D}'_s \to_s \mathbb{E}_t]_\perp\\[1ex]
\mathbb{E}_\iota &= \mathbb{D}'_\iota \oplus \mathbb{D}_\iota\\
\mathbb{E}_{s\to t} &= \mathbb{D}'_{s\to t} \oplus \mathbb{D}_\iota
\end{aligned}
$$

Semantic functions (As in Figure 2, with the following changes):

$$
\begin{aligned}
\hat{\hat{\mathfrak{A}}} &: \mathit{Programs} \to \mathbb{D}'_\iota\\
\hat{\hat{\mathfrak{A}}}[\![M]\!] &= \mathit{strip}(\hat{\mathfrak{A}}[\![M]\!]\perp)\\[1ex]
\hat{\mathfrak{A}} &: \mathit{Terms}_t \to \mathit{Env} \to \mathbb{E}_t\\
\hat{\mathfrak{A}}[\![\mathcal{A}M]\!]\rho &= \mathsf{inR}(\mathit{strip}(\hat{\mathfrak{A}}[\![M]\!]\rho))\\[1ex]
\mathit{fix}_{\mathrm{cbv}} &: \mathbb{D}'_{(s\to)\to(s\to t)} \to \mathbb{E}_{s\to t}\\
\mathit{fix}_{\mathrm{cbv}}(\perp) &= \perp\\
\mathit{fix}_{\mathrm{cbv}}(\mathsf{lift}(f)) &= \mathit{fix}(f) \text{ where } \mathit{fix}(f) = \textstyle\bigsqcup_{i=0}^{\infty}\{a_i\} \text{ for } a_0 = \perp, a_{i+1} = \mathit{apply}(f, a_i) = f(a_i)\\[1ex]
\mathit{apply} &: \mathbb{E}_{s\to t} \times \mathbb{E}_s \to \mathbb{E}_t\\
\mathit{apply}(\perp, v) &= \perp\\
\mathit{apply}(\mathsf{inL}(f), \mathsf{inL}(v)) &= f\,v\\
\mathit{apply}(\mathsf{inL}(f), \mathsf{inR}(v)) &= \mathsf{inR}(v)\\
\mathit{apply}(\mathsf{inR}(f), \mathsf{inL}(v)) &= \mathsf{inR}(f)\\
\mathit{apply}(\mathsf{inR}(f), \mathsf{inR}(v)) &= \mathsf{inR}(f)\\[1ex]
\mathit{strip} &: \mathbb{E}_t \to \mathbb{D}_t\\
\mathit{strip}(\mathsf{inL}(v)) &= v\\
\mathit{strip}(\mathsf{inR}(v)) &= v
\end{aligned}
$$

FIGURE 3: The extended direct model for $\mathrm{PCF_V}{+}\mathcal{A}$.

yields differing results.

A deterministically parallel operator that eliminates the counterexample is *parallel*-**if**:

$$M ::= \ \ldots \mid (\mathbf{pif}\ M_\iota\ M_\iota\ M_\iota)_\iota.$$

The construct **pif** is like **if**, but can also give a result if the sub-expression in the test position diverges. The choice of **pif** rather than **por** as the new $\mathrm{PCF_V}$-construct facilitates the proof of full abstraction below. The stepping rules for **pif** are:

$$
E[\mathbf{pif}\ M\ N\ P] \triangleright \begin{cases}
E[N] & \text{if } \ M \ \triangleright^* \ulcorner n \urcorner \text{ and } n > 0\\
E[P] & \text{if } M \ \triangleright^* \ulcorner 0 \urcorner\\
E[\mathsf{c}] & \text{if } \ N \ \triangleright^* \mathsf{c} \ \text{ and } P \ \triangleright^* \mathsf{c}
\end{cases}
$$

The clause for **pif**-expressions in the meaning function $\hat{\mathfrak{A}}$ is:

$$
\hat{\mathfrak{A}}[\![\mathbf{pif}\ M\ N\ P]\!]\rho = \begin{cases}
\hat{\mathfrak{A}}[\![N]\!]\rho & \text{if } \hat{\mathfrak{A}}[\![M]\!]\rho > 0\\
\hat{\mathfrak{A}}[\![P]\!]\rho & \text{if } \hat{\mathfrak{A}}[\![M]\!]\rho = 0\\
\hat{\mathfrak{A}}[\![N]\!]\rho & \text{if } \hat{\mathfrak{A}}[\![N]\!]\rho = \hat{\mathfrak{A}}[\![P]\!]\rho\\
\perp & \text{otherwise}
\end{cases}
$$

A parallel disjunction function on thunks is defined as:

$$\mathsf{por} = \lambda ab.\mathbf{pif}\ (a \ulcorner 0 \urcorner)\ \ulcorner 1 \urcorner\ (\mathbf{pif}\ (b \ulcorner 0 \urcorner)\ \ulcorner 1 \urcorner\ \ulcorner 0 \urcorner).$$

It is easy to verify that $\hat{\mathfrak{A}}[\![\mathsf{por}]\!]\perp = \underline{\mathit{por}}$ and that the terms N_u are operationally inequivalent in $\mathrm{PCF_V}$ with **pif**.

The direct model is fully abstract with respect to $\mathrm{PCF_V}$ with **pif**. The same result for Plotkin's related model of partial functions was independently obtained by Sieber [personal communication].

Theorem 4.2 *For* PCF_V+**pif** *terms* M, N, *(1)* $M \precapprox N$ *iff* $M \sqsubseteq N$, *and (2)* $M \simeq N$ *iff* $M \equiv N$.

Proof. The proof uses Plotkin's [21] method of defining terms for the finite elements of the domains. ∎

The direct model is easily extended to accommodate $\text{PCF}_V+\mathcal{A}$ [10]. Each domain is augmented with *exception values* to correspond to denotations of *abort* terms of that type. Since the exception values are always ground, they can be represented by $\mathbb{D}_\iota$. The domains and the semantic functionals for $\text{PCF}_V+\mathcal{A}$ are shown in Figure 3.

The direct model is *adequate* for $\text{PCF}_V+\mathcal{A}$, but is not fully abstract. As for PCF_V, $\text{PCF}_V+\mathcal{A}$ requires a *parallel*-**if** construct. The operational semantics of **pif** slightly differs from the one above to accommodate the presence of *aborts*:

$$E[\textbf{pif}\ M\ N\ P] \rhd \begin{cases} E[N] & \text{if} \quad M \ \rhd^* \ulcorner n \urcorner \text{ and} \\ & n > 0 \\ E[P] & \text{if } M \ \rhd^* \ulcorner 0 \urcorner \\ \mathsf{c} & \text{if } E[N] \ \rhd^* \mathsf{c} \text{ and} \\ & E[P] \ \rhd^* \mathsf{c} \end{cases}$$

Equality of the branches now also requires that their control actions are identical; *aborts* in the test-position are delimited. The $\hat{\mathfrak{A}}$-clause for **pif** remains the same as above but is interpreted in the larger model. Unlike in the functional case, **pif** is not sufficient for full abstraction.

Proposition 4.3 *The extended direct model is not fully abstract for* $\text{PCF}_V+\mathcal{A}$+**pif**.

Proof. Consider the $\text{PCF}_V+\mathcal{A}$ terms

$$M_u = \lambda f.\mathsf{equal?}\ (f\ \lambda d.\mathcal{A}\ulcorner 5 \urcorner)\ (f\ \lambda d.\mathcal{A}\ulcorner 5 + u \urcorner),$$

where u is either 0 or 1. The only way the terms could potentially differ in $\text{PCF}_V+\mathcal{A}$+**pif** is by applying f's argument, but delimiting the *abort*. Since the only delimiter available is the test-position of a **pif**, and since the only deductions possible from this position is whether the delimited test-value was zero or non-zero, the terms cannot be distinguished in $\text{PCF}_V+\mathcal{A}$+**pif**.

However, the model distinguishes them by applying the denotation of M_0 and M_1 to

$$\underline{\lambda}^s m.\mathsf{inL}(strip(m0)).$$

∎

To fix this problem, we augment the language with a control delimiter or *prompt* [8, 23]. A *prompt*-expression, $\%M$, runs its sub-expression M as an independent program insofar as control action is concerned. Aborts cannot extend beyond the dynamically closest prompt. Since a *prompt*-expression is treated as a program, it is always of ground type:

$$M ::= \ \ldots \mid (\%M_\iota)_\iota.$$

The additional stepping rule for $\text{PCF}_V+\mathcal{A}$+**pif** with prompts is

$$E[\%M] \ \rhd E[V] \text{ if } M \ \rhd^* V.$$

The clause that extends the semantic functional $\hat{\mathfrak{C}}$ to include prompt-expressions is:

$$\hat{\mathfrak{A}}[\![\%M]\!]\rho = \mathsf{inL}(strip(\hat{\mathfrak{A}}[\![M]\!]\rho)).$$

The terms M_u in Proposition 4.3 are operationally inequivalent in $\text{PCF}_V+\mathcal{A}$+**pif** with prompts. For, taking the context $C[\ ] = [\ ](\lambda f.\%(f\ulcorner 0 \urcorner))$, we have $C[M_0]$ and $C[M_1]$ evaluating to different results.

Theorem 4.4 *The extended direct model is adequate and fully abstract for* $\text{PCF}_V+\mathcal{A}$+**pif**+%.

Proof. The proof is a modification of the proofs of Theorems 4.1 and 4.2 that accounts for the exception elements. ∎

5 Adequacy and Full Abstraction of the CPS Model

The typed direct model does not easily extend to PCF_V-C with its higher-order continuations. The traditional model for such languages is the CPS (*continuation-passing style*) model [25]. The ground domain $\mathbb{D}_\iota$ remains the traditional cpo of natural numbers, but the domains for arrow types are no longer simple function spaces. Instead, a PCF_V-C procedure maps to a function that maps the argument denotation and a denotation representing the rest of the computation—a *continuation* function—to an answer. This scheme accounts for control operations in the procedure body that can affect the rest of the computation. Initially, we assume non-strict function spaces for all the domains; Figure 4 displays the actual solution that emerges for reasons discussed below.

Domains:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathbb{D}_\iota & = & \mathbb{N}_\perp & & & \\
\mathbb{D}_{\iota\to t} & = & \mathbb{D}_\iota \to_s \mathbb{C}_t \to \mathbb{D}_\iota & \mathbb{C}_\iota & = & \mathbb{D}_\iota \to_s \mathbb{D}_\iota \\
\mathbb{D}_{s\to t} & = & \mathbb{D}_s \to \mathbb{C}_t \to \mathbb{D}_\iota,\ s \neq \iota & \mathbb{C}_{s\to t} & = & \mathbb{D}_{s\to t} \to \mathbb{D}_\iota
\end{array}
$$

Semantic functions:

$$
\begin{array}{rcl}
\hat{\hat{\mathfrak{C}}} & : & \mathit{Programs} \to \mathbb{D}_\iota \\
\hat{\hat{\mathfrak{C}}}[\![M]\!] & = & \hat{\mathfrak{C}}[\![M]\!]\perp(\underline{\lambda}^s x.x) \\
 & & \\
\hat{\mathfrak{C}} & : & \mathit{Terms}_t \to \mathit{Env} \to \mathbb{C}_t \to \mathbb{D}_\iota \\
\hat{\mathfrak{C}}[\![\mathsf{b}]\!]\rho\kappa & = & \kappa\mathfrak{C}[\![\mathsf{b}]\!] \\
\hat{\mathfrak{C}}[\![x]\!]\rho\kappa & = & \kappa(\rho[\![x]\!]) \\
\hat{\mathfrak{C}}[\![\lambda x.M]\!]\rho\kappa & = & \kappa(\underline{\lambda} v, \kappa'.\hat{\mathfrak{C}}[\![M]\!]\rho[x/v]\kappa') \\
\hat{\mathfrak{C}}[\![\mathbf{Y}]\!]\rho\kappa & = & \kappa\, \mathit{fix}_{\mathrm{cps}} \\
\hat{\mathfrak{C}}[\![MN]\!]\rho\kappa & = & \hat{\mathfrak{C}}[\![M]\!]\rho(\underline{\lambda} m.\hat{\mathfrak{C}}[\![N]\!]\rho(\underline{\lambda} n.mn\kappa)) \\
\hat{\mathfrak{C}}[\![\mathcal{A}\ M]\!]\rho\kappa & = & \hat{\mathfrak{C}}[\![M]\!]\rho(\underline{\lambda}^s v.v) \\
\hat{\mathfrak{C}}[\![\mathcal{K}\ M]\!]\rho\kappa & = & \hat{\mathfrak{C}}[\![M]\!]\rho(\underline{\lambda} m.m(j\kappa)\kappa) \\
 & & \\
j & : & \mathbb{C}_s \to \mathbb{D}_{s\to\iota} \\
j(\kappa) & = & \underline{\lambda} v, \kappa'.\kappa v \\
 & & \\
\mathit{fix}_{\mathrm{cps}} & : & \mathbb{D}_{((s\to t)\to(s\to t))\to(s\to t)} \\
\mathit{fix}_{\mathrm{cps}} & = & \underline{\lambda} f, \kappa. \bigsqcup_{i=0}^{\infty}\{a_i\},\ \text{where } a_0 = f\perp\kappa, a_1 = f\perp(\underline{\lambda} n.fn\kappa), a_2 = f\perp(\underline{\lambda} n.fn(\underline{\lambda} n.fn\kappa)),\quad \ldots
\end{array}
$$

FIGURE 4: The CPS model for $\mathrm{PCF_V}$-C.

The CPS model is *adequate* for $\mathrm{PCF_V}$, $\mathrm{PCF_V}+\mathcal{A}$, $\mathrm{PCF_V}+\mathcal{K}$ and $\mathrm{PCF_V}$-C.

Theorem 5.1 (Adequacy of CPS) *For any program M in $\mathrm{PCF_V}$-C,*

$$\mathit{eval}(M) = \mathsf{b} \text{ iff } \hat{\mathfrak{C}}[\![M]\!]\perp(\underline{\lambda}^s x : \mathbb{D}_\iota.x) = \mathfrak{C}[\![\mathsf{b}]\!].$$

Proof. The proof is similar to the one for the direct model. ∎

By Lemma 3.3, since the CPS meaning function is compatible, terms that are denotationally equivalent in our four languages are also operationally equivalent. Unfortunately, but not surprisingly, the model is *not* fully abstract for any of these languages.

Proposition 5.2 *The CPS model is not fully abstract for plain $\mathrm{PCF_V}$.*

Proof. Full abstraction breaks down in three distinct ways in the naïve (fully non-strict) CPS model for $\mathrm{PCF_V}$:

1. The simple model for $\mathrm{PCF_V}$ assigns non-strict functions to *apparently* non-strict procedures.[3] As a result, call-by-value procedures that are operationally equivalent like

   $$e_1 = (\lambda d.\ulcorner 0 \urcorner)$$

   and

   $$e_2 = (\lambda d.\mathbf{if}\ d \ulcorner 0 \urcorner \ulcorner 0 \urcorner)$$

   have distinct meanings in the model:

   $$\hat{\mathfrak{C}}[\![e_1]\!]\perp\kappa = \kappa\underline{\lambda} d, \kappa'.\kappa' 0$$

   versus

   $$\hat{\mathfrak{C}}[\![e_2]\!]\perp\kappa = \kappa\underline{\lambda}^s d, \kappa'.\kappa' 0.$$

   If $\kappa = \underline{\lambda} m.m\perp_\iota(\underline{\lambda} x.x)$, the former evaluates to 0 and the latter to $\perp$.

2. Consider the terms from Section 4:

[3] Olivier Danvy and Andrzej Filinski pointed out this problem with the naïve model and provided the idea for the counterexample.

$$N_u = \lambda x.\mathbf{if}\ (x\ (\lambda d.\ulcorner 1 \urcorner)\ (\lambda d.\Omega))\ (\mathbf{if}\ (x\ (\lambda d.\Omega)\ (\lambda d.\ulcorner 1 \urcorner))\ (\mathbf{if}\ (x\ (\lambda d.\ulcorner 0 \urcorner)\ (\lambda d.\ulcorner 0 \urcorner))\ \Omega\ \ulcorner u \urcorner)\ \Omega)\ \Omega,$$

for $u = 0, 1$.

We have $N_0 \simeq_v N_1$, but $N_0 \not\equiv N_1$. Applying the denotations of N_u to the continuation $\kappa_1 = \underline{\lambda} m.m(\underline{por}_c)(\underline{\lambda} x : \mathbb{D}_\iota.x)$ produces 0 and 1, respectively. The function $\underline{por}_c$ is the CPS version of parallel-**or**:

$$\underline{por}_c = \underline{\lambda} m, \kappa.\kappa \underline{\lambda} n, \kappa. \begin{cases} \kappa 1 & \text{if } m0\underline{\lambda} x.x > 0 \\ & \text{or } n0\underline{\lambda} x.x > 0 \\ \kappa 0 & \text{if } m0\underline{\lambda} x.x = 0 \\ & \text{and } n0\underline{\lambda} x.x = 0 \\ \perp & \text{otherwise.} \end{cases}$$

3. Recall $M_1 = \lambda f.\Omega$ and $M_2 = \lambda f.f\ulcorner 0 \urcorner\Omega$ from above. M_1 and M_2 are operationally but not denotationally equivalent in PCF_V. In particular, let

$$\kappa_0 = \underline{\lambda} m : \mathbb{D}_{(\iota\to(\iota\to\iota))\to\iota}.\ m(\underline{\lambda} v : \mathbb{D}_\iota, \kappa : \mathbb{C}_{\iota\to\iota}.v)(\underline{\lambda} x : \mathbb{D}_\iota.x).$$

Then we have

$$\hat{\mathfrak{C}}[\![M_1]\!]\perp\kappa_0 = \perp,$$

whereas

$$\hat{\mathfrak{C}}[\![M_2]\!]\perp\kappa_0 = 0. \qquad \blacksquare$$

We can eliminate the first problem by using strict spaces for those function domains that originate from a ground domain (see Figure 4). The second counterexample shows that the model can express deterministic parallelism. The continuation κ_0 in the third counterexample is clearly the equivalent of an *aborting* action. Thus, adding $\mathcal{A}$, for example, suffices to eliminate this problem (Propositions 2.2 and 2.3). Control extensions to PCF_V do not eliminate the first problem but they do remove counterexamples of the second kind. In the following propositions, we gradually work out the control specific deficiencies of our PCF_V extensions.

Proposition 5.3 *There is a control-specific counterexample to full abstraction of the CPS model for* $\text{PCF}_\text{V}+\mathcal{A}$.

Proof. Consider the terms M_u $(u = 0, 1)$ from the proof of Proposition 2.4:

$$M_u = \lambda f.\mathsf{equal?}(f\lambda k.k\ulcorner 1 \urcorner\Omega)(f\lambda k.k\ulcorner u \urcorner\Omega).$$

Recall that $M_0 \simeq_a M_1$, but $M_0 \not\equiv M_1$. For, taking

$$\kappa_0 = \underline{\lambda} m.m(\underline{\lambda} f, \kappa.f(j\kappa)(\underline{\lambda}^s x.x)),$$

we have $\hat{\mathfrak{C}}[\![M_0]\!]\perp\kappa_0 = 0$ and $\hat{\mathfrak{C}}[\![M_1]\!]\perp\kappa_0 = 1$. ∎

The function $(j\kappa)$ corresponds to a procedural value that substitutes one continuation for another, but there is no expression in the language that has this denotation. This is exactly the property of continuation objects provided by $\mathcal{K}$. Hence, adding $\mathcal{K}$ to the language makes the terms M_u operationally inequivalent (Proposition 2.4). The choice between $\mathcal{A}$ and $\mathcal{K}$ suggested by Proposition 5.2 is thus illusory: the proper language for a CPS model *must* have a $\mathcal{K}$-like facility, whether or not it has an $\mathcal{A}$. Indeed, it is easy to see that the control facility must give *first-class* access to continuations. Still, not even the addition of $\mathcal{K}$ suffices to recover full abstraction.

Proposition 5.4 *There is a control-specific counterexample to full abstraction of the CPS model for* PCF_V-C *and* $\text{PCF}_\text{V}+\mathcal{K}$.

Proof. Consider the PCF_V-C terms

$$M_u = \lambda z.\mathsf{equal?}\ (z\ \lambda d.\mathcal{A}\ulcorner 1 \urcorner)\ (z\ \lambda d.\mathcal{A}\ulcorner u \urcorner),$$

where u is either 0 or 1. Distinguishing these terms operationally requires that z apply its argument, and, since the first z-argument aborts, the second z-argument has no opportunity to affect the answer. In short, $M_0 \simeq_c M_1$.

On the other hand, let

$$\kappa_0 = \underline{\lambda} m.m[\underline{\lambda} f, \kappa_1.(\kappa_1(f0(\underline{\lambda}^s x.x)))](\underline{\lambda}^s x.x).$$

Then, $\hat{\mathfrak{C}}[\![M_u]\!]\perp\kappa_0 = \mathfrak{C}[\![\mathsf{u}]\!]$ and hence, $M_0 \not\equiv M_1$.

$\text{PCF}_\text{V}+\mathcal{K}$ has essentially the same counterexample, but since it does not have *abort*, the $\mathcal{A}$-expressions in the M_u have to be replaced by invocations of an appropriate continuation, e.g.,

$$M'_u = \lambda z.\mathcal{K}(\lambda a.\mathsf{equal?}(z\lambda d.a\ulcorner 1 \urcorner)(z\lambda d.a\ulcorner u \urcorner)). \quad \blacksquare$$

The continuation κ_0 in the above proof first runs the thunk f in an identity continuation before passing the result on to κ_1, the continuation that came along with f. This makes the result of calling f available for immediate examination, even if it involved an abort or a continuation invocation. In other words, the model has a facility to run a sub-computation as an *independent* program whose control actions are isolated from the enclosing program. On finishing the sub-computation, its result is used to resume the rest of the computation. PCF_V-C cannot simulate this *control delimiting* action since aborts or continuation invocations are irrevocable.

To fix the problem posed by Proposition 5.4, we augment PCF_V-C with a control delimiter or *prompt*, as for $\text{PCF}_V+\mathcal{A}$ (Section 4). In this setting, a *prompt*-expression delimits *all* control actions that it dynamically encloses, including aborts, context captures by $\mathcal{K}$-expressions, or context switches on continuation invocations. The operational semantics of the *prompt* is the same as the one given in Section 4. The clause that extends the semantic functional $\hat{\mathfrak{C}}$ to include prompt-expressions is:

$$\hat{\mathfrak{C}}[\![\%M]\!]\rho\kappa = \kappa(\hat{\mathfrak{C}}[\![M]\!]\rho\underline{\lambda}^s x.x).$$

The operator that eliminates the second counterexample in the proof of Proposition 5.2 is once again a parallel disjunction procedure **por** based on **pif**, introduced in Section 4. The $\hat{\mathfrak{C}}$-clause for **pif**-expressions is:

$$\begin{aligned}&\hat{\mathfrak{C}}[\![\mathbf{pif}\ M\ N\ P]\!]\rho\kappa\\ &= \begin{cases} \hat{\mathfrak{C}}[\![N]\!]\rho\kappa & \text{if } \hat{\mathfrak{C}}[\![M]\!]\rho\underline{\lambda}^s x.x > 0,\\ \hat{\mathfrak{C}}[\![P]\!]\rho\kappa & \text{if } \hat{\mathfrak{C}}[\![M]\!]\rho\underline{\lambda}^s x.x = 0,\\ \hat{\mathfrak{C}}[\![N]\!]\rho\kappa & \text{if } \hat{\mathfrak{C}}[\![N]\!]\rho\kappa = \hat{\mathfrak{C}}[\![P]\!]\rho\kappa,\\ \bot & \text{otherwise.}\end{cases}\end{aligned}$$

The operational semantics of **pif** is identical to the one for $\text{PCF}_V+\mathcal{A}+$**pif**. Also as above, the addition of **pif** is orthogonal to the addition of *prompt*.

By augmenting PCF_V-C with **pif** and *prompt*, we arrive at a language for which the CPS model is fully abstract. The proof follows Plotkin's plan [21] and essentially consists in showing that there are PCF_V-C terms for defining all finite elements. In our case, this includes elements from both the value *and* the continuation domains.

Definition 5.5 [Definability]

1. *A value $v \in \mathbb{D}_s$ is* $\mathbb{D}$-definable *if there is a* PCF_V-C *term V such that for all ρ, κ:*

$$\hat{\mathfrak{C}}[\![V]\!]\rho\kappa = \begin{cases} \bot & \text{if } v = \bot_\iota : \mathbb{D}_\iota,\\ \kappa v & \text{otherwise.}\end{cases}$$

2. *A continuation $\kappa \in \mathbb{C}_s$ is* $\mathbb{C}$-definable *if there is a* PCF_V-C *program evaluation context $E[\,]$ such that for all ρ, M:*

$$\hat{\mathfrak{C}}[\![E[M]]\!]\rho(\underline{\lambda}^s x.x) = \hat{\mathfrak{C}}[\![M]\!]\rho\kappa.$$

To prove full abstraction of the CPS model, we need to show that all *finite* elements can be defined in PCF_V-C.

Lemma 5.6 *For all types s,*

1. *$v \in \mathbb{D}_s$ is $\mathbb{D}$-definable;*
2. *$\kappa \in \mathbb{C}_s$ is $\mathbb{C}$-definable.*

Proof. The proof is by induction on s, with the following strengthening of the hypothesis: for each type s, we prove that a collection of values in $\mathbb{D}_{s\to\iota}$ and $\mathbb{D}_{(s\to\iota)\to\iota}$ are also definable, and then use this fact in the induction step.

Thus, we need to show that for every finite $v, v' \in \mathbb{D}_s$, $\kappa, \kappa' \in \mathbb{C}_s$:

1. v is $\mathbb{D}$-definable;
2. $v \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$ is $\mathbb{D}$-definable, if it exists; and
3. $(v \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1) \sqcup (v' \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 0)$ is $\mathbb{D}$-definable, if it exists.

and also,

1. κ is $\mathbb{C}$-definable;
2. $j(\kappa) \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1$ is $\mathbb{D}$-definable; and
3. $(j(\kappa) \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 1)$ $\sqcup\, (j(\kappa') \Rightarrow (\bot \Rightarrow \bot) \Rightarrow 0)$ is $\mathbb{D}$-definable, if it exists.

The notation $a \Rightarrow b$ denotes a one-step function in the appropriate domain. The transformation j on continuations is specified in Figure 4. ∎

From this lemma, it follows that operational equivalent terms are also denotationally equivalent.

Theorem 5.7 (Full Abstraction of CPS) *For terms M and N in* PCF_V-C *(1) $M \sqsubseteq\!\!\!\!\sim N$* iff *$M \sqsubseteq N$, and (2) $M \simeq N$* iff *$M \equiv N$.*

Proof. The proof is standard, based on Lemma 5.6 [21]. ∎

6 Enhancing Control Structures

In our study of models for $\mathrm{PCF_V}$ with control extensions, we looked at both a simple aborting facility and a higher-order control facility, using a direct and a CPS model, respectively. In each case, the model is not fully abstract as it has more capabilities than the language. Both direct and CPS models can

- express deterministic parallelism; and
- restrict the actions of control constructs.

To get full abstraction, we could either restrict the model or further extend the language. Taking the latter recourse, we need to incorporate a facility for deterministic parallelism (**pif**) and a control delimiter (*prompt*) into the language. Some of the above language extensions are already present in Lisp and Scheme, though not always in their most general form. In traditional Lisp, the first-order combination of *prompt* and $\mathcal{A}$ occurs as errset and error. Common Lisp [24] no longer provides a facility like errset, i.e., it misses a control delimiter for error; its **catch** mechanism is sufficient as a *prompt* for control actions by **throw**. Scheme has higher-order continuations based on call/cc but also lacks a control delimiter.[4] Neither programming language has deterministic parallelism.

Purely functional languages reflect their conventional direct model accurately but for the non-sequential **pif**. To establish an analogous relationship, languages with control require a control delimiter to express all sequential aspects of their models. This extension is also supported by practical evidence. Our own investigation of control delimiters found them to yield several useful programming paradigms for languages like Scheme [8, 23]; Danvy and Filinski [4] describe other interesting uses of *prompt* and related concepts. Recently, Hieb and Dybvig [16] argued for an extension of *prompt*, called *spawn*, as a crucial programming tool for concurrent systems with control operators. Thus, on the strength of both denotational completeness and practical utility, the *prompt* merits consideration for inclusion in sequential languages with control facilities, whether first- or higher-order.

[4]With side-effects and an *eval* function, it is possible to simulate *prompt* in Scheme [23].

In conclusion, with the right extensions, a Lisp- or Scheme-like language not only increases its power as a programming language, but, just like purely functional languages, becomes a genuine mathematical reasoning system based on the fully abstract model.

Acknowledgment. We are grateful to Robert Cartwright and Albert Meyer for helpful discussions on this topic.

References

[1] Aiken A., J.H. Williams, and E.L. Wimmers. Program transformation in the presence of errors. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 210–217, 1990.

[2] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics 103. North-Holland, Amsterdam, 1984. Revised edition.

[3] V. Breazu-Tannen, C. Gunter, and A. Scedrov. Computing with coercions. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990, this volume.

[4] O. Danvy and A. Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, 1990, this volume.

[5] R.K. Dybvig and R. Hieb. Engines from continuations. *Journal of Computer Languages* (Pergamon Press), 14(2):109–124, 1989.

[6] M. Felleisen. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[7] M. Felleisen. λ-v-CS: An extended λ-calculus for Scheme. In *Proc. 1988 Conference on Lisp and Functional Programming*, pages 72–84, 1988.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[9] M. Felleisen. On the expressive power of programming languages. In *Proc. European Symposium on Programming.* Lecture Notes in Computer Science, 1990. To appear.

[10] M. Felleisen and R.S. Cartwright. Extended direct semantics. Technical Report 105, Rice University, January 1990.

[11] M. Felleisen, D.P. Friedman, E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theor. Comput. Sci.*, 52(3):205–237, 1987. Preliminary version: Reasoning with Continuations, *Proc. Symposium on Logic in Computer Science*, 1986, 131–141.

[12] D.P. Friedman, C.T. Haynes, and E. Kohlbecker. Programming with continuations. In P. Pepper, editor, *Program Transformations and Programming Environments*, pages 263–274. Springer-Verlag, Heidelberg, 1985.

[13] T. Griffin. A formulæ-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, 1990, to appear.

[14] C.T. Haynes and D.P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):245–254, 1987.

[15] C.T. Haynes, D.P. Friedman, and M. Wand. Obtaining coroutines from continuations. *Journal of Computer Languages* (Pergamon Press), 11(3/4):109–121, 1986.

[16] R. Hieb and R.K. Dybvig. Continuations and concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990.

[17] J. McCarthy et al. *Lisp 1.5 Programmer's Manual.* The MIT Press, Cambridge, second edition, 1965.

[18] A.R. Meyer and J.R. Riecke. Continuations may be unreasonable. In *Proc. 1988 Conference on Lisp and Functional Programming*, pages 63–71, 1988.

[19] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[20] G.D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.*, 1:125–159, 1975.

[21] G.D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5:223–255, 1977.

[22] J. Rees and W. Clinger. The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[23] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[24] G.L. Steele Jr. *Common Lisp—The Language.* Digital Press, 1984.

[25] C. Strachey and C.P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

[26] G.J. Sussman and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

[27] C. Talcott. *The Essence of Rum—A Theory of the Intensional and Extensional Aspects of Lisp-type Computation.* PhD thesis, Stanford University, 1985.

[28] C. Talcott. Rum: An intensional theory of function and control abstractions. In *Proc. 1987 Workshop on Foundations of Logic and Functional Programming.* Springer Lecture Notes **306**, 1988.

A Domains and their Properties

A domain is a partially ordered collection of elements. The ordering relation ($\sqsubseteq$) is generally a *complete partial order*. The following describes the domain constructions and properties used in the paper.

Domain constructions:

$\mathbb{N}_\perp$**:** This is the traditional cpo of natural numbers.

$\mathbb{A} \to \mathbb{B}$**:** This is a *function* cpo consisting of the *continuous* (i.e., lub-preserving) functions from the cpo $\mathbb{A}$ to the cpo $\mathbb{B}$. The ordering is that induced by pointwise ordering. We shall use "meta-lambda" ($\underline{\lambda}$) to notate elements in the function domains, e.g.,

$$\underline{\lambda} x : \mathbb{A}.1 = \{(x, 1) \mid x \in \mathbb{A}\}.$$

$\mathbb{A} \to_s \mathbb{B}$**:** This is the *strict* and *continuous* function cpo. We shall use $\underline{\lambda}^s$ to notate strict functions, e.g.,

$$\begin{aligned}\underline{\lambda}^s x : \mathbb{A}.1 = \{&(\perp_{\mathbb{A}}, \perp)\} \\ &\cup \{(x, 1) \mid x \in \mathbb{A} \setminus \{\perp_{\mathbb{A}}\}\}.\end{aligned}$$

$\mathbb{A} \oplus \mathbb{B}$**:** This is the *disjoint union* cpo of cpos $\mathbb{A}$ and $\mathbb{B}$, and is given by:

$$\begin{aligned}\mathbb{A} \oplus \mathbb{B} \quad = \quad & \{(a, 0) \mid a \in \mathbb{A} \setminus \{\perp_{\mathbb{A}}\}\} \\ & \cup \{(b, 1) \mid b \in \mathbb{B} \setminus \{\perp_{\mathbb{B}}\}\} \\ & \cup \{\perp_{\mathbb{A} \oplus \mathbb{B}}\}.\end{aligned}$$

The ordering is inherited from the constituent domains, with $\perp_{\mathbb{A} \oplus \mathbb{B}}$ being the new bottom element. We use $\mathsf{inL}(a)$ and $\mathsf{inL}(b)$ to refer to the elements $(a, 0)$ and $(b, 1)$ respectively.

$\mathbb{A}_\perp$**:** This is the *lift* of the cpo $\mathbb{A}$, given by:

$$\{(a, 0) \mid a \in \mathbb{A}\} \cup \{\perp\}.$$

Again, the ordering is inherited from the original domain, with the new bottom being the least element. We use $\mathsf{lift}(a)$ to refer to the non-$\perp$ element $(a, 0) \in \mathbb{A}_\perp$.

Domain properties

Finite elements: A domain element $d \in \mathbb{A}$ is *finite* if for all directed subsets $X \subseteq \mathbb{A}$, $d \sqsubseteq \sqcup X$ implies $d \sqsubseteq e$ for some $e \in X$.

Prefinite elements: A prefinite element $d \Rightarrow e$ in $\mathbb{X} \to \mathbb{Y}$ is an element such that:

$$(d \Rightarrow e)x = \begin{cases} e & \text{if } x \sqsupseteq d, \\ \perp & \text{otherwise.} \end{cases}$$

By Plotkin, the finite elements of a function domain are the lubs of finite sets of prefinite elements.

Algebraicity: A domain $\mathbb{A}$ is *algebraic* if for any $x \in \mathbb{A}$, the set $\{d \mid d \text{ is finite } \wedge d \sqsubseteq x\}$ is directed with x as its lub.

Consistent completeness: A domain is *consistently complete* if two elements with an upper bound also have a *least* upper bound.

The domain of natural numbers, $\mathbb{N}_\perp$, is consistently complete and algebraic. Further, the domains constructions for (strict) function cpo, disjoint union, and lifting preserve consistent completeness and algebraicity. Thus, all our domains are consistently complete algebraic cpos.