Modeling Continuations without Continuations

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

Abstract

Traditional denotational models of languages with control operators rely on Strachey and Wadsworth's continuation semantics. Such models represent the effects of control operations by tacking on an additional function argument, the continuation, to all denotations. In essence, a continuation semantics encodes a stack machine for the language where the continuation is a functional representation of the stack. As we have shown recently, a continuation model can accurately describe the behavior of a control language, provided the latter has a sufficiently strong control structure.

In this paper we investigate a new class of models for control operators. These models do not rely on the continuation-passing technique but build the required information for control operations upon demand. In contrast to the continuation framework, there is a pair of simple projection/injection functions between the direct model for the core language without control operators and the extended model for the full language. Like the continuation model, the new models provide accurate descriptions of languages with control operators *and* control delimiters. For the design of programming languages, our analysis points out that control operators need a cooperative exception-handling facility.

*Supported in part by NSF and DARPA.

1 Modeling control without special effects

Programming language operators for control manipulation provide expressive and efficient programming abstractions. These constructs grant access to (first-class) abstractions of the control state in the form of control objects. Virtually all programming languages are equipped with some form of control operators.

Although the study of denotational models is useful for understanding and reasoning about languages, these analysis techniques have not been exploited for languages with control.

The traditional semantics for control operators and control objects uses the Strachey and Wadsworth continuation model [25]. Our recent study of this model showed that it has the ability to restrict the reach of control operators. This capability is usually not available in modern languages. The language requires a control-delimiting operator to allow an accurate correspondence between model and language.

However, the continuation model, while popular (indeed, the control objects provided by the language are often dubbed "continuations"), differs radically from the basic direct model for languages without control. The denotations of language phrases demand an additional continuation argument, and it is difficult to relate them to the denotations in the direct model for the language without control [19, 21, 24].

In this work, we investigate the consequences of using the more uniform approach of having *extensions* of the basic direct semantics [8] as models for language extensions. Extending a direct model requires only an incremental alteration to the base model. In our specific case, languages with "continuations" have models *without special continuation functions.* Our study also shows that, like the continuation model, the extended direct model furnishes a perfect match with a language, provided the latter has a sufficiently expressive control structure.

The following section introduces the syntax and an informal description of the language PS and some of its control extensions. Section 3 describes the direct model for PS. The fourth section explains the techniques used to obtain extended models for PS enhanced with control facilities. Section 5 discusses related work, and the final section interprets our technical results for the design of programming languages.

2 PS and its control extensions

PS is the purely functional subset of Scheme. It is essentially the untyped call-by-value λ-calculus [16] with integers and some basic procedures. A control extension of PS is plain PS enhanced with one or more control constructs.

2.1 PS

Figure 1 describes the syntax of PS. PS terms are either values such as numbers, basic procedures, and λ-abstractions, or non-values such as variables and (procedure) applications. The basic procedures include a conditional (with 0 serving as *false* and anything else as *true*), arithmetic functions and the predicate int?. The λ-abstractions introduce new by-value procedures.

A *program* is a PS term without free variables. On evaluation, a program either converges to an *answer* value or diverges. In the evaluation process, basic procedures have their expected behavior; applications are evaluated from left to right before the argument is passed to the procedure.

2.2 Control extensions

Evaluating subterms in a PS program involves keeping track of the rest of the evaluation, or the *evaluation context.* In a machine, this information is usually represented as a stack; in rewriting systems, it is the textual context surrounding the current redex. Control operators exploit an abstraction of the evaluation context to allow manipulation of the control flow in a program. We shall consider **abort**, call/cc [26], $\mathcal{C}$ [10], and control [9] as additions to PS:

abort stops the normal evaluation of a program, and replaces its evaluation context with its subexpression.

call/cc is a procedure that applies its argument to a procedure that is an abstracted form of the surrounding evaluation context. It leaves the evaluation context intact. In analogy to denotational semantics, Scheme refers to the abstracted context as a *continuation.* (Indeed, call/cc abbreviates "call-with-current-continuation".) On application, this continuation replaces its current evaluation context in favor of the one that was captured, sending its argument to the old context.

$\mathcal{C}$ is similar to call/cc except that a $\mathcal{C}$-application does not implicitly invoke its continuation. $\mathcal{C}$ and the combination **abort** and call/cc can express [7] each other: see Figure 2.

control, in contrast to call/cc and $\mathcal{C}$, provides a *functional* continuation to its argument. Upon invocation, the functional continuation installs its context without throwing away the current one. As shown in Figure 2, control can express all the other constructs.

A different kind of control construct is the control delimiter or *prompt* (#) [6, 22]:

# *constrains* control manipulation occurring within its dynamic extent. In other words, a *prompt*-expression treats its subexpression as an independent program, insofar as control action is concerned.

Prompts are an important component of the full abstraction result for cps models [23]; we shall find them useful here, too.

These operators are sufficiently abstract to capture other popular variations such as **escape** [18], Iswim's J [13], GL's state [12], spawn [11], **shift** and **reset** [5].

2.3 A small library

Many elementary procedures and forms are easy derivations from the core of PS. Here we list a couple that are useful below.

- A conditional form **if** is the syntactic extension ($d \notin N, P$):

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

  Here, $\lambda d.N$ is the *thunk* form of N, and is discharged by applying it to a dummy value, $\ulcorner 0 \urcorner$.

- A typical PS program that fails to *converge* is

  $$\Omega \equiv (\lambda x.xx)(\lambda x.xx).$$

- The fixpoint combinator Y provides a tool for defining recursive procedures:

  $$Y \equiv \lambda f.(\lambda w.ww)(\lambda x.f(\lambda z.(xx)z)).$$

$$
\begin{array}{rcll}
M & := & V \mid x \mid MM & \\
V & := & \mathsf{c} \mid \mathsf{f} \mid \lambda x.M & \text{(values)} \\
\mathsf{c} & := & n & \text{(integers)} \\
\mathsf{f} & := & 1{+} \mid 1{-} \mid \mathsf{int?} \mid \mathsf{ef} & \text{(basic procedures)}
\end{array}
$$

Control additions to PS

$$M \;:=\; \mathbf{abort}\ M \mid \mathsf{call/cc}\ M \mid \mathcal{C}\ M \mid \mathsf{control}\ M \mid \#\ M$$

Figure 1: Syntax of PS.

$$
\begin{array}{rcl}
\mathbf{abort}\ M & \equiv & \mathcal{C}(\lambda d.M) \\
\mathsf{call/cc} & \equiv & \lambda f.\mathcal{C}(\lambda k.k(fk)) \\
\mathcal{C} & \equiv & \lambda f.\mathsf{call/cc}\ (\lambda k.\mathbf{abort}\ (fk)) \\
\mathcal{C} & \equiv & \lambda f.\mathsf{control}(\lambda k.f(\lambda v.\mathsf{control}\ (\lambda d.kv)))
\end{array}
$$

Figure 2: The expressiveness of **abort**, call/cc, $\mathcal{C}$, and control.

- Any value that is not a number is a procedure:

$$\mathsf{proc?} \equiv \lambda v.\mathbf{if}\ (\mathsf{int?}\ v)\ \ulcorner 0 \urcorner\ \ulcorner 1 \urcorner.$$

3 Modeling PS

A model for a language consists of a structure, the *domain*, and an interpretation or *meaning function* that maps phrases from the language to values in the structure. This section summarizes the tools for building models, the direct model for PS, and some of its properties.

3.1 Information systems

We use Scott's [20] information systems approach to construct the domains for the basic direct model and its extensions. The appendix summarizes the salient features of information systems.

An information system domain is a collection of consistent, deductively closed sets of propositions. A *finite* element is (the deductive closure of) a finite set of propositions. The complete domain is isomorphic to the ideal closure of its finite elements, and hence the latter suffice for studying the domain. The subset relation on the sets composing the elements gives an ordering $\sqsubseteq$ on the domain. Domain constructions, e.g., disjoint sums ($\oplus$), strict function spaces ($\rightarrow_s$), reflexive domains, etc., consist in enumerating the finite elements in terms of the finite elements of the constituent domains.

3.2 The direct model for PS

The direct model for PS is a reflexive domain $\mathbb{D}$ that contains integers and strict functions on itself:

$$\mathbb{D} = \underbrace{\mathbb{O}}_{\text{integers}} \oplus \underbrace{(\mathbb{D} \rightarrow_s \mathbb{D})}_{\text{procedures}} .$$

The appendix shows how to build such a domain as an information system. The tags inO and inP refer to elements in the subdomains for atomic *O*bservables and *P*rocedures, respectively. E.g., the elements for the number 9 and the procedure that maps 1 to 1 (and everything else to $\bot$) are respectively $\mathsf{inO}(9)$ and $\mathsf{inP}(\overline{\{\langle \mathsf{inO}(1), \mathsf{inO}(1)\rangle\}})$. In the latter case, the set containing the pair denotes a finite consistent proposition in the information system for $\mathbb{D} \rightarrow \mathbb{D}$ and the overline denotes its deductive closure. This notation rapidly becomes unwieldy for larger procedures. We subsequently use the more concise notation $\mathsf{inP}(\mathsf{inO}(1) \Rightarrow \mathsf{inO}(1))$ to denote the same procedure. To avoid clutter, we can further omit the tags when there is no ambiguity.

The function $\mathfrak{A}$ (Figure 3) defines the meanings of PS terms. *Env* is a set of environments or mappings from variables to domain values (*other than* $\bot$). The meaning of a PS program P is simply $\mathfrak{A}'[\![P]\!] = \mathfrak{A}[\![P]\!]\bot$, where $\bot$ is the empty environment.

In the following, the functions *freeze* and *thaw* : $\mathbb{D} \rightarrow \mathbb{D}$ are the semantic counterpart of forming and discharging a thunk:

$$\begin{aligned} \mathit{freeze}(v) &= \mathsf{inP}(\underline{\lambda} d.v) \\ \mathit{thaw}(f) &= \mathit{apply}(f, \mathsf{inO}(0)) \end{aligned}$$

3.3 Semantic equivalence relations

The model determines two natural equivalence relations on the language terms [14]. First, there is the relation based on their meanings in the model.

Definition 3.1 Two terms M and N are *denotationally equivalent*, $M \equiv N$, if $\mathfrak{A}[\![M]\!] = \mathfrak{A}[\![N]\!]$.

The second relation describes the behavior of terms as it appears to the programmer. For the latter, the only way to witness a term's effect is by using it as a subterm in a program, and then observing the *program*'s behavior (meaning). In this view, two terms are indistinguishable if one can be substituted for the other in any program without affecting that program's meaning. For our purposes, it is enough to measure the termination behavior.[1] In the following, a context $C[\ ]$ is a PS term with a "hole" where a subterm should be; a *program* context for a term is a context that becomes a program when filled with that term.

Definition 3.2 Two terms M and N are *observationally equivalent*, $M \simeq N$, if for all $C[\ ]$ that are program contexts for M and N, $\mathfrak{A}'[\![C[M]]\!] = \bot$ iff $\mathfrak{A}'[\![C[N]]\!] = \bot$.

For example, $\lambda f.\Omega \simeq \lambda f.(f\ulcorner 1 \urcorner)\Omega$: no program context can distinguish them. On the other hand, $\lambda x.\Omega x \not\simeq \Omega$: the first term converges in the empty context, whereas the latter diverges.

Owing to the compositional definition of $\mathfrak{A}$, it follows that $M \equiv N$ implies $C[M] \equiv C[N]$ for all program contexts $C[\,]$, i.e., $M \simeq N$. The converse property is more interesting, viz., a model should not give different meanings to two language terms, if it cannot distinguish their behavior through any program context. This property is called *full abstraction* [14, 17].

Definition 3.3 A model is *fully abstract* if for any two terms M and N in the language, $M \simeq N$ iff $M \equiv N$.

3.4 Full abstraction of the direct model for PS+pif

Full abstraction fails for a model when the latter has some capability that cannot be mimicked in the language. For PS, this capability is the deterministic parallel conditional [17]. The model uses this conditional to distinguish the terms M_u ($u = 0, 1$):

$$\begin{aligned} M_u = \lambda x.\mathbf{if}\ &(x(\lambda d.\ulcorner 1\urcorner)(\lambda d.\Omega)) \\ &(\mathbf{if}\ (x(\lambda d.\Omega)(\lambda d.\ulcorner 1\urcorner)) \\ &\quad (\mathbf{if}\ (x(\lambda d.\ulcorner 0\urcorner)(\lambda d.\ulcorner 0\urcorner))\ \Omega\ \ulcorner u\urcorner) \\ &\quad \Omega) \\ &\Omega. \end{aligned}$$

Although $M \simeq N$, $M \not\equiv N$, for applying their denotations to the denotation for parallel disjunction on thunks:

$$\underline{por_\theta} = \mathsf{inP}\underline{\lambda} m, n. \begin{cases} \mathsf{inO}(0) & \text{if both } \mathit{thaw}(m) \text{ and } \mathit{thaw}(n) \text{ are zero} \\ \bot & \text{if both are } \bot \\ \mathsf{inO}(1) & \text{otherwise} \end{cases}$$

yields 0 and 1 respectively. In other words, $\mathfrak{A}[\![M_0]\!]\rho$ and $\mathfrak{A}[\![M_1]\!]\rho$ are different functions in $\mathbb{D} \to_s \mathbb{D}$.

To rectify this, PS is enhanced with the operator **pif**, which is similar to **if**, but can yield a result even when the test fails to converge. For convenience we shall use the thunk form pif_θ where:

$$\mathbf{pif}\ M\ N\ P \equiv \mathsf{pif}_\theta\ (\lambda d.M)\ (\lambda d.N)\ (\lambda d.P).$$

The semantics of pif_θ is given by:

$\mathfrak{A}[\![\mathsf{pif}_\theta]\!]\rho = \mathsf{inP}(\underline{pif_\theta})$
where $\underline{pif_\theta} =$

$$\underline{\lambda} b, t, e. \begin{cases} \mathit{thaw}(t) & \text{if } \mathit{thaw}(b) \text{ is neither zero nor } \bot \\ \mathit{thaw}(e) & \text{if } \mathit{thaw}(b) \text{ is zero} \\ \mathsf{inO}(i) & \text{if } \mathit{thaw}(t) = \mathit{thaw}(e) = \mathsf{inO}(i), \text{ a number} \\ \mathsf{inP}(\underline{\lambda} v.\underline{pif_\theta}\ t\ (\mathit{freeze}(nv))\ (\mathit{freeze}(pv))) & \text{if } \mathfrak{A}[\![N]\!]\rho = \mathsf{inP}(n), \mathfrak{A}[\![P]\!]\rho = \mathsf{inP}(p), \text{ i.e., both are procedures} \end{cases}$$

The form **pif** defines parallel-or on thunks as follows:

$$\begin{aligned} \mathsf{por} = \lambda m, n.\mathbf{pif}\ &(m\ulcorner 0\urcorner)\ \ulcorner 1\urcorner \\ &(\mathbf{pif}\ (n\ulcorner 0\urcorner)\ \ulcorner 1\urcorner\ \ulcorner 0\urcorner) \end{aligned}$$

It is easy to verify that $\mathfrak{A}[\![\mathsf{por}]\!]\bot = \underline{por_\theta}$, and that by using the context $[\ ]\mathsf{por}$ in PS+**pif**, $M_0 \not\simeq M_1$.

Unlike the typed languages PCF and PCF$_{\text{v}}$ [17, 23], the PS form of **pif** has to deal with procedures in the branches of the conditional, in

[1]For a better understanding of higher-order data, we would have to use a more sophisticated denotational framework [4].

$$\begin{array}{rcl}
\mathbb{D} & = & \mathbb{O} \oplus (\mathbb{D} \rightarrow_s \mathbb{D}) \\
\\
\mathfrak{A} & : & \mathit{Terms} \rightarrow \mathit{Env} \rightarrow \mathbb{D} \\
\\
\mathfrak{A}[\![\ulcorner n \urcorner]\!]\rho & = & \mathsf{inO}(n) \\
\mathfrak{A}[\![\mathsf{f}]\!]\rho & = & \mathsf{inP}(f) \\
\mathfrak{A}[\![x]\!]\rho & = & \rho[\![x]\!] \\
\mathfrak{A}[\![\lambda x.M]\!]\rho & = & \mathsf{inP}(\underline{\lambda} v : \mathbb{D}.\mathfrak{A}[\![M]\!]\rho[x/v]) \\
\mathfrak{A}[\![MN]\!]\rho & = & \mathit{apply}(\mathfrak{A}[\![M]\!]\rho, \mathfrak{A}[\![N]\!]\rho) \\
\\
\mathit{apply} & : & \mathbb{D} \times \mathbb{D} \rightarrow_s \mathbb{D} \\
\\
\mathit{apply}(\mathsf{inP}(f), a) & = & f(a) \\
\mathit{apply}(f, a) & = & \bot, \text{ for other domain values}
\end{array}$$

Figure 3: The semantic functions for PS.

which case it postpones the decision until the procedures are applied. Call-by-name versions of PS without constants require similar treatment [1, 15].

The proof of full abstraction for PS+**pif** requires that all finite elements of the model be *definable* in the language.

Lemma 3.4 For all finite elements e, f,

1. e is definable;
2. $\mathsf{inP}(e \Rightarrow 1)$, if it exists, is definable (this is a procedure that takes e and anything above it to true, others to $\bot$); and
3. $\mathsf{inP}((e \Rightarrow 1) \sqcup (f \Rightarrow 0))$, if it exists, is definable (this is a procedure that takes anything dominating e to true, anything dominating f to false, and the rest to $\bot$).

Proof sketch. The proof of the lemma basically follows Plotkin's proof for PCF [17], with two exceptions. First, there are no strong types that allow an induction on the type structure. Second, since the dynamic type of an application is unpredictable, taking lubs does not translate to checking the equality of integer values as in the PCF proofs. An induction on the *"size"* (based on the constituent propositions) of a finite element and the generalized **pif** address these issues. ∎

The proof of full abstraction of the model follows from the lemma.

Theorem 3.5 The direct model is fully abstract for PS+**pif**.

Proof sketch. Assume that $M \not\sqsubseteq N$. We now need to show that $M \not\preceq N$, i.e., there is a program context distinguishing the terms. Both $\mathfrak{A}[\![M]\!]\rho$ and $\mathfrak{A}[\![N]\!]\rho$ (for any ρ) cannot be $\bot$. If one of them is $\bot$, a context that merely closes the terms differentiates them. If one of them is a number and the other a procedure, the procedure int? tells them apart. If both are numbers, simple arithmetic operations suffice. And if both are procedures, then applying them to a finite number of finite elements produces a number or $\bot$ for one and something else for the other, which can then be distinguished as above. ∎

4 Modeling PS's control extensions

Extended direct models [8] accommodate the language extensions of PS. The basic domain $\mathbb{D}$ satisfies:

$$\begin{array}{rcl}
\mathbb{D}' & = & \overbrace{\mathbb{O}}^{\text{integers}} \oplus \overbrace{\mathbb{D} \rightarrow_s \mathbb{D}'}^{\text{procedures}} \\
\mathbb{D} & = & \mathbb{D}'
\end{array}$$

An extended domain introduces an additional subdomain for the denotations introduced by the language extension:

$$\begin{array}{rcl}
\mathbb{D}'_e & = & \overbrace{\mathbb{O}}^{\text{integers}} \oplus \overbrace{(\mathbb{D}'_e \rightarrow_s \mathbb{D}_e)}^{\text{procedures}} \\
\mathbb{D}_e & = & \mathbb{D}'_e \oplus \underbrace{\Gamma(\mathbb{D}'_e, \mathbb{D}_e)}_{\text{extension values}}
\end{array}$$

We use tags inL and inR to refer to the left and right subdomains of $\mathbb{D}_e$.

Additional clauses for the semantic function $\mathfrak{A}$ define the meanings associated with the language extensions:

$$\mathfrak{A}_e : PS_e \rightarrow \mathit{Env} \rightarrow \mathbb{D}_a.$$

The environments map variables to *values that are neither $\bot$ nor the extended values.* A modified version of the semantic function *apply* shows the denotational effect of the new domain values. Finally, a function *strip* transforms the meanings of programs into denotations that are the counterparts of values from the original subdomains.

Extending a domain may modify some existing subdomains that depend reflexively on the whole domain, but it does so in an easily predictable manner. More precisely, a pair of functions $\Phi : \mathbb{D} \longrightarrow \mathbb{D}_e$ and $\Psi : \mathbb{D}_e \longrightarrow \mathbb{D}$ connect the domain $\mathbb{D}$ and its extended counterpart $\mathbb{D}_e$. Φ injects values from the unextended domain $\mathbb{D}$ to their counterparts in the larger domain $\mathbb{D}_e$; Ψ projects values from the extended domain $\mathbb{D}_e$ to the smaller $\mathbb{D}$:

$$\Phi(v) = \begin{cases} \mathsf{inL}(v) & \text{if } v = \mathsf{inO}(n) \\ \mathsf{inL}(\mathsf{inP}(\Phi \circ f \circ \Psi)) & \text{if } v = \mathsf{inP}(f) \end{cases}$$

$$\Psi(w) = \begin{cases} w & \text{if } w = \mathsf{inL}(\mathsf{inO}(n)) \\ \mathsf{inP}(\Psi \circ f \circ \Phi) & \text{if } w = \mathsf{inL}(\mathsf{inP}(f)) \\ \bot & \text{if } w = \mathsf{inR}(u) \end{cases}$$

In short, the two mappings are a pair of projections. With these projections, it is possible to approximate the extended denotations of phrases in the core language using their old denotations, and to recover the old meanings of such phrases using their new meanings.

Theorem 4.1 ([8]) If M is a PS expression, and ρ and ρ' are environments such that $\rho : \mathit{Vars} \longrightarrow \mathbb{D}$ and $\rho' : \mathit{Vars} \longrightarrow \mathbb{D}'_e$, then

$$\mathfrak{A}[\![M]\!]\rho = \Psi(\mathfrak{A}_e[\![M]\!](\mathsf{outl} \circ \Phi \circ \rho))$$

and

$$\Phi(\mathfrak{A}[\![M]\!](\Psi \circ \mathsf{inL} \circ \rho')) \sqsubseteq \mathfrak{A}_e[\![M]\!]\rho'$$

where outl removes the inL tag of its argument.

4.1 Modeling PS+abort

The extended model for PS+**abort** illustrates the technique of extending a direct semantics to include a new language feature. The extended domain $\mathbb{D}_a$ satisfies the following equations:

$$\begin{aligned} \mathbb{D}'_a &= \overbrace{\mathbb{O}}^{\text{integers}} \oplus \overbrace{(\mathbb{D}'_a \longrightarrow_s \mathbb{D}_a)}^{\text{procedures}} \\ \mathbb{D}_a &= \mathbb{D}'_a \oplus \underbrace{\mathbb{D}_a}_{\text{abort values}} \end{aligned}$$

The tag inA (same as inR) refers to elements in the subdomain for *A*bort values. The semantic function $\mathfrak{A}_a$ maps terms in PS+**abort** and environments to values in $\mathbb{D}_a$, and is similar to $\mathfrak{A}$ for terms that do not include **abort**.

The additional semantic clause for **abort** is:

$$\mathfrak{A}_a[\![\textbf{abort } M]\!]\rho = \underline{abort}(\mathfrak{A}_a[\![M]\!]\rho),$$
$$\text{where } \underline{abort} = \mathsf{inL} \circ \mathit{strip}.$$

Figure 4 defines *apply* and *strip*. The new function *apply* describes how **abort**-values ignore their surrounding context. Unlike in the cps model, it is trivial to specify left-to-right or right-to-left evaluation for applications involving **abort**s. The function *strip* removes the tags associated with **abort** values to yield the result of a program. The denotation of a program P is $\mathfrak{A}'_a[\![P]\!] = \mathit{strip}(\mathfrak{A}_a[\![P]\!]\bot)$.

The parallel-**if** construct for PS+**abort** differs slightly from the one for plain PS, since it must accommodate control action in the branches. The semantics of the new **pif** is:

$$\mathfrak{A}[\![\mathsf{pif}_\theta]\!]\rho = \begin{cases} \ldots \text{ as for plain PS} \\ \underline{abort}(\mathit{pif}_\theta\ b\ (\mathit{freeze}(p))\ (\mathit{freeze}(q))) \\ \quad \text{if } \mathit{thaw}(t) = \mathsf{inA}(p) \text{ and} \\ \quad \mathit{thaw}(e) = \mathsf{inA}(q) \end{cases}$$

Full abstraction for PS+abort

Models for control usually have the ability to delimit control [23]. This is also case for the extended direct model, e.g., $\mathbb{D}'_a \longrightarrow \mathbb{D}'_a$ contains the function $\underline{\lambda}\theta.\mathit{strip}(\mathit{thaw}(\theta))$ that constrains the **abort**s in its argument thunk. This function can distinguish the denotations of the terms

$$\begin{aligned} M_u \equiv \lambda z.&\mathsf{equal?} \\ &(z(\lambda d.\textbf{abort}\ulcorner 1 \urcorner)) \\ &(z(\lambda d.\textbf{abort}\ulcorner u \urcorner)), \end{aligned}$$

where $u = 1, 2$, and $\mathsf{equal?}$ has the usual definition. Differentiating the terms in a program context requires that the context should both invoke the **abort**'s *and* perform the $\mathsf{equal?}$ test. An **abort**-delimiter in the language solves this problem by allowing the procedure z to prevent the **abort**'s from escaping past the $\mathsf{equal?}$.

Thus, to restore full abstraction to the extended direct model for control, the language requires a further extension in the form of a control delimiting construct. We choose the *prompt* [6].

The denotational semantics of *prompt* is an abstraction of the *strip* function:

$$\mathfrak{A}_a[\![\#M]\!]\rho = \mathsf{inL}(\mathit{strip}(\mathfrak{A}_a[\![M]\!]\rho)).$$

$$\begin{array}{rcll} \mathit{apply} & : & \mathbb{D}_a \times \mathbb{D}_a \to_s \mathbb{D}_a & \\ \mathit{apply}(\mathsf{inP}(\mathsf{inL}(f)), \mathsf{inL}(a)) & = & f a & \\ \mathit{apply}(\mathsf{inA}(a), b) & = & \mathsf{inA}(a) & \\ \mathit{apply}(a, \mathsf{inA}(b)) & = & \mathsf{inA}(b) & \\ \mathit{apply}(a, b) & = & \bot & \text{for other domain values} \\ & & & \\ \mathit{strip} & : & \mathbb{D}_a \to_s \mathbb{D}'_a & \\ \mathit{strip}(\mathsf{inA}(a)) & = & a & \\ \mathit{strip}(\mathsf{inL}(a)) & = & a & \end{array}$$

Figure 4: The semantic functions *apply* and *strip* for PS+**abort**.

With **pif** and *prompt* added to the language, the extended model for PS+**abort** becomes fully abstract.

Theorem 4.2 The extended direct model (with domain $\mathbb{D}_a$) is fully abstract for PS+**abort**+**pif**+**#**.

Proof sketch. As a slight variation on Lemma 3.4, we now need to show that for all finite elements e, f in $\mathbb{D}_a$:

1. e is definable;
2. $\mathsf{inP}((0 \Rightarrow e) \Rightarrow 1)$,[2] if it exists, is definable (this is a procedure that takes the thunk form of e, $\underline{\lambda} d.e$ (discharged *only* with 0), and anything above it to true, others to $\bot$); and
3. $\mathsf{inP}(((0 \Rightarrow e) \Rightarrow 1) \sqcup ((0 \Rightarrow f) \Rightarrow 0))$, if it exists, is definable (this is a procedure that takes anything dominating the thunk $\underline{\lambda} d.e$ to true, anything dominating the thunk $\underline{\lambda} d.f$ to false, and all else to $\bot$).

The proof that the extended direct model is fully abstract follows from the above lemma, using a strategy similar to that of the last section. In addition to int? and proc?, the proof needs a predicate that tests if its argument thunk **aborts**:

$$\begin{array}{l} \mathsf{abort?} \equiv \lambda f.\mathsf{sameint?} \\ \qquad (\#((\lambda d.\ulcorner 0 \urcorner)(f \ulcorner 0 \urcorner))) \\ \qquad (\#((\lambda d.\ulcorner 1 \urcorner)(f \ulcorner 0 \urcorner))). \end{array}$$

where sameint? returns false only if its arguments are unequal numbers:

$$\begin{array}{l} \mathsf{sameint?} \equiv \lambda x, y.\mathbf{if}\ (\mathbf{and}\ (\mathsf{int?}\ x)\ (\mathsf{int?}\ y)) \\ \qquad\qquad (\mathsf{equal?}\ x\ y) \\ \qquad\qquad \ulcorner 1 \urcorner \qquad \blacksquare \end{array}$$

[2]Actually, $\mathsf{inP}(\mathsf{inP}(\mathsf{inO}(0) \Rightarrow e) \Rightarrow \mathsf{inL}(\mathsf{inO}(1)))$!

4.2 Modeling PS with control

As for PS+**abort**, the extended direct model for PS with a higher-order control operators such as call/cc, $\mathcal{C}$, or control requires an additional subdomain, this time a domain of procedures called "receivers". A receiver denotes the argument procedure of a control construct, and *receives* the latter's continuation object as *its* argument.

Taking X to be any one of $\{\mathsf{call/cc}, \mathcal{C}, \mathsf{control}\}$, the specifications of the corresponding extended domain $\mathbb{D}_x$ are as follows:

$$\begin{array}{rcl} \mathbb{D}'_x & = & \overbrace{\mathbb{O}}^{\text{integers}} \oplus \overbrace{(\mathbb{D}'_x \to_s \mathbb{D}_x)}^{\text{procedures}} \\ \mathbb{D}_x & = & \mathbb{D}'_x \oplus \underbrace{\mathbb{D}_x}_{X\text{-receivers}} \end{array}$$

The tag inX indicates elements belonging to the subdomain for X-receivers.

The semantic clause for the X's are :

$$\mathfrak{A}_x[\![X\ M]\!]\rho \;=\; \mathsf{inX}(\mathfrak{A}_x[\![M]\!]\rho).$$

Figure 5 shows the semantic function for PS+control along with the auxiliaries *apply* and *strip*. The function *apply* deals with receiver arguments by expanding them to include progressively more surrounding context through the pending applications. The function *strip* simulates the final discharge of the accumulated receiver procedure with an identity continuation.

Both *apply* and *strip* change based on the control operator, and furthermore, call/cc requires an additional subdomain for **abort**-values as for PS+**abort**. Figure 7 describes the modified functions.

Full abstraction

The programming languages PS+X need control delimiters too, to make their models fully abstract.

$$
\begin{aligned}
\mathbb{D}'_c &= \mathbb{O} \oplus (\mathbb{D}'_c \to_s \mathbb{D}_c) \\
\mathbb{D}_c &= \mathbb{D}'_c \oplus \mathbb{D}_c \\[1em]
\mathfrak{A}'_c &: \mathit{Terms} \to \mathbb{D}'_c \\
\mathfrak{A}'_c[\![M]\!] &= \mathit{strip}(\mathfrak{A}_c[\![M]\!]\bot, \mathsf{inL}(\mathsf{inP}(\underline{\lambda} x.\mathsf{inL}(x)))) \\[1em]
\mathfrak{A}_c &: \mathit{Terms} \to \mathit{Env} \to \mathbb{D}_c \\
\mathfrak{A}_c[\![\ulcorner n \urcorner]\!]\rho &= \mathsf{inL}(\mathsf{inO}(n)) \\
\mathfrak{A}_c[\![\mathsf{f}]\!]\rho &= \mathsf{inL}(\mathsf{inP}(f)) \\
\mathfrak{A}_c[\![x]\!]\rho &= \mathsf{inL}(\rho[\![x]\!]) \\
\mathfrak{A}_c[\![\lambda x.M]\!]\rho &= \mathsf{inL}(\mathsf{inP}(\underline{\lambda} v : \mathbb{D}'_c.\mathfrak{A}_c[\![M]\!]\rho[x/v])) \\
\mathfrak{A}_c[\![M N]\!]\rho &= \mathit{apply}(\mathfrak{A}_c[\![M]\!]\rho, \mathfrak{A}_c[\![N]\!]\rho) \\
\mathfrak{A}_c[\![\mathsf{control}\ M]\!]\rho &= \mathsf{inC}(\mathfrak{A}[\![M]\!]\rho) \\
\mathfrak{A}_c[\![\%\ M\ H]\!]\rho &= \mathsf{inL}(\mathit{strip}(\mathfrak{A}_c[\![M]\!]\rho, \mathfrak{A}_c[\![H]\!]\rho)) \\[1em]
\mathit{apply} &: \mathbb{D}_c \times \mathbb{D}_c \to_s \mathbb{D}_c \\
\mathit{apply}(\mathsf{inL}(\mathsf{inP}(f)), \mathsf{inL}(a)) &= f a \\
\mathit{apply}(\mathsf{inC}(a), b) &= \mathsf{inC}(\mathsf{inL}(\mathsf{inP}(\underline{\lambda} k : \mathbb{D}'_c. \\
&\qquad \mathit{apply}(a, \mathsf{inL}(\mathsf{inP}(\underline{\lambda} v : \mathbb{D}'_c.\mathit{apply}(\mathsf{inL}(k), \mathit{apply}(\mathsf{inL}(v), b))))))) \\
\mathit{apply}(\mathsf{inL}(a), \mathsf{inC}(b)) &= \mathsf{inC}(\mathsf{inL}(\mathsf{inP}(\underline{\lambda} k : \mathbb{D}'_c. \\
&\qquad \mathit{apply}(b, \mathsf{inL}(\mathsf{inP}(\underline{\lambda} v : \mathbb{D}'_c.\mathit{apply}(\mathsf{inL}(k), \mathit{apply}(\mathsf{inL}(a), \mathsf{inL}(v)))))))) \\
\mathit{apply}(a, b) &= \bot \qquad \text{for other domain values} \\[1em]
\mathit{strip} &: \mathbb{D}_c \times \mathbb{D}_c \to_s \mathbb{D}'_c \\
\mathit{strip}(a, \mathsf{inC}(f)) &= \bot \\
\mathit{strip}(\mathsf{inC}(f), h) &= \mathit{strip}(\mathit{apply}(\mathit{apply}(h, f), \mathsf{inL}(\mathsf{inP}(\underline{\lambda} x : \mathbb{D}'_c.\mathsf{inL}(x)))), h) \\
\mathit{strip}(\mathsf{inL}(a), h) &= a
\end{aligned}
$$

Figure 5: The semantic functions for PS+control.

$$
\begin{aligned}
\mathit{apply}(\mathsf{in}\mathcal{C}(a), b) &= \text{as for PS+control} \\
\mathit{apply}(\mathsf{inL}(a), \mathsf{in}\mathcal{C}(b)) &= \text{as for PS+control} \\[1em]
\mathit{strip}\mathsf{in}\mathcal{C}(f), h) &= \mathit{strip}(\mathit{apply}(\mathit{apply}(h, f), \mathsf{inL}(\mathsf{inP}(\underline{\lambda} x.\mathsf{in}\mathcal{C}(\mathsf{inL}(\mathsf{inP}(\underline{\lambda} d.\mathsf{inL}(x))))))))
\end{aligned}
$$

Figure 6: Modified *apply* and *strip* clauses for PS+$\mathcal{C}$.

$$
\begin{aligned}
\mathit{apply}(\mathsf{inK}(a), b) &= \mathsf{inK}(\mathsf{inL}(\mathsf{inP}(\underline{\lambda} k : \mathbb{D}'_k. \\
&\qquad \mathit{apply}(\mathit{apply}(a, \mathsf{inL}(\mathsf{inP}(\underline{\lambda} v : \mathbb{D}'_k.\mathit{apply}(\mathsf{inL}(k), \mathit{apply}(\mathsf{inL}(v), b))))), b)))) \\
\mathit{apply}(\mathsf{inL}(a), \mathsf{inK}(b)) &= \mathsf{inK}(\mathsf{inL}(\mathsf{inP}(\underline{\lambda} k : \mathbb{D}'_k. \\
&\qquad \mathit{apply}(\mathsf{inL}(a), \mathit{apply}(b, \mathsf{inL}(\mathsf{inP}(\underline{\lambda} v : \mathbb{D}'_k.\mathit{apply}(\mathsf{inL}(k), \mathit{apply}(\mathsf{inL}(a), \mathsf{inL}(v)))))))))) \\[1em]
\mathit{strip}(\mathsf{inK}(f), h) &= \mathit{strip}(\mathit{apply}(\mathit{apply}(h, f), \mathsf{inL}(\mathsf{inP}(\underline{\lambda} x : \mathbb{D}'_k.\mathsf{inA}(x)))), h) \\
\mathit{strip}(\mathsf{inA}(x), h) &= \mathit{apply}(\mathit{apply}(h, \mathsf{inL}(\mathsf{inP}(\underline{\lambda} v : \mathbb{D}'_k.\mathsf{inL}(v)))), \mathsf{inL}(x))
\end{aligned}
$$

Figure 7: Modified *apply* and *strip* clauses for PS+call/cc.

However, this control delimiter should not only constrain control but also identify the *number of invocations* of the control operator in its dynamic extent. Here, we shall consider the most general of these languages, viz., PS+control; the others allow a similar treatment.

For example, the terms control $(\lambda k.\ulcorner 3 \urcorner)$ and control $(\lambda k.\text{control } (\lambda k.\ulcorner 3 \urcorner))$, though indistiguishable with the plain *prompt*, have different denotations. The first term invokes control twice, whereas the second does it just once.

A new form of *prompt*, $(\%\ M\ M)$, that takes a handler expression as an additional parameter solves this mismatch.[3] The additional subexpression is a *handler.* If the first subexpression returns without any control incident, the *prompt* does nothing. However, a control-application inside a *prompt* sends both receiver and continuation to the handler, which is then invoked on them in a fresh *prompt* with the same handler. A program is always run within an initial *prompt* with the identity handler, $\lambda f, k.fk$.

This extended *prompt* can distinguish the two terms above as follows. Letting $H \equiv Y(\lambda h.\lambda f, k.1{+}(\%(fk)h))$, the programs

$$\%\ (\text{control } (\lambda k.\ulcorner 3 \urcorner))\ H$$

and

$$\%\ (\text{control } (\lambda k.\text{control } (\lambda k.\ulcorner 3 \urcorner)))\ H$$

yield 4 and 5 respectively.

The function *strip* in Figure 5 reflects the effect of the handler in addition to the *prompt*'s control delimiting aspect. On encountering a receiver as an argument, *strip* supplies the identity function as the functional continuation to the composition of the handler and the receiver, and awaits the result for further receivers that may turn up.

The semantic clause for a *prompt*-expression and the semantic function $\mathfrak{A}'_c$ for complete programs is shown in Figure 5.

The parallel-**if** construct for PS+control undergoes the same change as for PS+**abort** to address control action in its branches. Further, a predicate control? is defined along the same lines as abort?.

The model for PS+control+**pif** is fully abstract, provided the language is enhanced with the *prompt*-with-a-handler. The related languages PS+call/cc and PS+$\mathcal{C}$ go through in like fashion.

Theorem 4.3 1. The extended direct model with domain $\mathbb{D}_c$ is fully abstract for PS+control+**pif**+%.

2. An extended direct model is fully abstract for PS+call/cc+**pif**+%.

3. An extended direct model is fully abstract for PS+$\mathcal{C}$+**pif**+%.

[3]Bruce Duba suggested prompts with handlers.

The proof is as for PS and PS+**abort**, with special care taken for receiver denotations. One interesting consequence of having the enhanced prompt is that it dissolves the differences between the control operators. Each pair of prompt and control operator can simulate any other pair. Thus, the extended direct model demands a delimiter that compensates for the drawbacks of the particular control operator in the language.

5 Related Work

Previous studies on the relationship between direct and continuation semantics [19, 21, 24] rely on *retraction* functions, and do not furnish the projection/injection maps supported by extensible direct models. — The Vienna School of denotational semantics [2, 3] treats **gotos** in a *first-order* imperative setting using an extension of a direct semantics, but contains no investigation of the formal aspects of these models or their relationship to direct models. — Felleisen and Cartwright's work on extended direct semantics [8] introduces the extension technique as a generalized mechanism to accommodate a variety of language extensions and studies the projection relations between their models. However, they do not seek extensions with a view to full abstraction. — The local reduction rules for $\mathcal{C}$ and control [9, 10] motivated the work on the denotational framework of extended direct models. Indeed, they are the operational counterpart of the *apply* clauses for the denotations of receivers in the extended model for PS+X.

6 Conclusions

In this study, we investigated fully abstract, extended direct models for the purely functional subset of Scheme and its control extensions. In particular, studying the correspondence between the meanings of language phrases in the model and the observable effect that these phrases have in complete programs both pinpoints and suggests remedies for deficiencies in the language design.

The direct models for the extended languages are the extensions of the basic direct model for pure PS. Projections connect the basic model and its extensions. All the models are fully abstract for their respective languages, provided the latter contain parallel-**if** and, in the case of the control extensions, a suitable control delimiter.

The new form of the control delimiter not only constrains control action but also invokes a handler procedure for every control-application in its dynamic extent. Such prompts allow easy implementations and form a powerful generalization of existing exception handling mechanisms.

Acknowledgment. We gratefully acknowledge discussions with Robert Cartwright and Bruce Duba. Mitch Wand's Semantic Prototyping System helped in type-checking the denotational semantics.

A Domains as information systems

This work approaches the construction of reflexive domains using Scott's information systems [20]. The following outlines the definition of such domains from finite sets of propositions, constructions of domains from other domains, and their salient properties.

Information systems

An *information system* is a structure

$$\langle \mathcal{D}, \Delta, \mathsf{Con}, \vdash \rangle$$

where

- $\mathcal{D}$ is a set of *data objects* or *propositions*;
- Δ is the *least informative member* of $\mathcal{D}$;
- Con is a set of finite subsets of $\mathcal{D}$, the finite *consistent* sets of propositions; and
- $\vdash$, *entailment*, is a binary relation on Con.

The set of consistent propositions and the entailment relation satisfy the following properties:

Properties of Con:

- if $u \subseteq v \in \mathsf{Con}$, then $u \in \mathsf{Con}$;
- if $X \in \mathcal{D}$, then $\{X\} \in \mathsf{Con}$;
- if $u \vdash v$, then $u \cup v \in \mathsf{Con}$;

Properties of $\vdash$:

- if $u \in \mathsf{Con}$, then $u \vdash \{\Delta\}$;
- if $u \in \mathsf{Con}$ and $u \supseteq v$, then $u \vdash v$;
- if $u \vdash v$ and $v \vdash w$, then $u \vdash w$.

The *elements* of the information system

$$\mathbb{A} = \langle \mathcal{D}_{\mathbb{A}}, \Delta_{\mathbb{A}}, \mathsf{Con}_{\mathbb{A}}, \vdash_{\mathbb{A}} \rangle$$

are all those subsets x of $\mathcal{D}_{\mathbb{A}}$ such that:

- all finite subsets of x are in $\mathsf{Con}_{\mathbb{A}}$;
- if a finite (i.e., $\in \mathsf{Con}_{\mathbb{A}}$) $u \subseteq x$ and $u \vdash v$, then $v \subseteq x$;

The deductive closure $\overline{u}$ of an element $u \in \mathsf{Con}_{\mathbb{A}}$ is the union of all $v \in \mathsf{Con}_{\mathbb{A}}$ such that $u \vdash_{\mathbb{A}} v$: From the definition of the elements of $\mathbb{A}$, if $u \in \mathsf{Con}_{\mathbb{A}}$, then $\overline{u} \in \mathbb{A}$.

The class of domains studied here are exclusively information systems. A domain element is thus a deductively closed (closed under $\vdash$) and consistent (all its finite subsets are in Con) set of propositions. The subset relation forms an ordering relation ($\sqsubseteq$) between the domain elements. The information-system-as-domain with its ordering relation $\sqsubseteq$ is a *complete partial order*.

Domain properties

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

Alternately, an element is finite if it is the deductive closure (d.c.) $\overline{u}$ for some $u \in \mathsf{Con}_{\mathbb{A}}$. The two definitions are equivalent.

Minimal representations of finite elements: *A minimal representation* (it need not be unique) of a finite element d is a finite consistent set $u \in \mathsf{Con}_{\mathbb{A}}$ such that $d = \overline{u}$ and for any $v \in \mathsf{Con}_{\mathbb{A}}$, if $d = \overline{v}$ and $u \vdash v$, then $u \subseteq v$. For function domains, minimal representations form a succinct finite description of the finite elements. Most proofs involving finite elements need only consider their minimal representations.

Algebraicity: A domain $\mathbb{A}$ is *algebraic* if for any $x \in \mathbb{A}$, the set $\{d \mid d \text{ is finite and } d \sqsubseteq x\}$ 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.

Domain constructions

Information systems provide a succinct description of various domain constructions, including *lifting*, *disjoint unions*, *function spaces*, and *reflexive domains.*

$\mathbb{A}_\perp$: *Lifting* a domain $\mathbb{A}$ adds a new bottom to the space. The set of propositions $\mathcal{D}_{\mathbb{A}_\perp}$ is $\mathcal{D}_\mathbb{A} \cup \{\Delta_{\mathbb{A}_\perp}\}$, where the new l.i.m. $\Delta_{\mathbb{A}_\perp}$ is consistent with all the original propositions.

$\mathbb{N}$: The traditional cpo of natural numbers. The set of propositions corresponding to $\mathbb{N}$ treated as an information system is $\mathcal{D}_\mathbb{N} = \{\Delta_\mathbb{N}, 0, 1, 2, \ldots\}$, with no two elements being consistent unless one of them is $\Delta_\mathbb{N}$.

$\mathbb{A} \oplus \mathbb{B}$: This is the *disjoint union domain* of $\mathbb{A}$ and $\mathbb{B}$. The propositions in $\mathcal{D}_{\mathbb{A}\oplus\mathbb{B}}$ are of the form

$$\mathcal{D}_{\mathbb{A}\oplus\mathbb{B}} = \left\{ \begin{array}{l} \{\mathsf{inL}(X) \mid X \in \mathcal{D}_\mathbb{A}\} \\ \qquad \text{the left subdomain} \\ \cup \{\mathsf{inR}(Y) \mid Y \in \mathcal{D}_\mathbb{B}\} \\ \qquad \text{the right subdomain} \\ \cup \{\perp_{\mathbb{A}\oplus\mathbb{B}}\} \\ \qquad \text{a new least inform.} \\ \qquad \text{member} \end{array} \right.$$

A proposition in the left subdomain is always inconsistent with a proposition in the right subdomain. Within the propositions stemming from one of the constituent domains, the entailment relation $\vdash_{\mathbb{A}\oplus\mathbb{B}}$ is exactly that domain's entailment relation, viz., $\vdash_\mathbb{A}$ or $\vdash_\mathbb{B}$ respectively.

$\mathbb{A} \to \mathbb{B}$: This is the *function domain* from $\mathbb{A}$ to $\mathbb{B}$. The set of propositions $\mathcal{D}_{\mathbb{A}\to\mathbb{B}}$ consists of pairs (u, v) where $u \in \mathsf{Con}_\mathbb{A}$ and $v \in \mathsf{Con}_\mathbb{B}$. The least informative member $\Delta_{\mathbb{A}\to\mathbb{B}}$ is $(\varnothing, \varnothing)$.

The set $w = \{\ldots, (u_i, v_i), \ldots\}$ belongs to $\mathsf{Con}_{\mathbb{A}\to\mathbb{B}}$ iff

$$\left(\bigcup_{\text{for some } i\text{'s}} \{u_i\} \right) \in \mathsf{Con}_\mathbb{A}$$

implies

$$\left(\bigcup_{\text{for the } \textit{same } i\text{'s}} \{v_i\} \right) \in \mathsf{Con}_\mathbb{B}.$$

Also, $w \vdash_{\mathbb{A}\to\mathbb{B}} \{(u, v)\}$ iff $\bigcup_i \{v_i \mid u \vdash_\mathbb{A} u_i\} \vdash_\mathbb{B} v$.

A finite function $c \in \mathbb{A} \to \mathbb{B}$ has a minimal representation w, where $w \in \mathsf{Con}_{\mathbb{A}\to\mathbb{B}}$ and $\overline{w} = c$. A finite *step* function $(a \Rightarrow b) \in \mathbb{A} \to \mathbb{B}$, where $a \in \mathbb{A}$ and $b \in \mathbb{B}$ are finite elements, is the function that takes anything dominating a to b and everything else to $\perp$. A function $c \in \mathbb{A} \to \mathbb{B}$ can thus be represented more directly as the lub of a finite number of step functions (rather than data objects) approximating it:

$$c = \ldots \sqcup a_i \Rightarrow b_i \sqcup \ldots.$$

$\mathbb{A} \to_s \mathbb{B}$: This is the *strict* function domain from $\mathbb{A}$ to $\mathbb{B}$. The construction is as above with the restriction that if $w = \{\ldots, (u_i, v_i), \ldots\} \in \mathsf{Con}_{\mathbb{A}\to_s\mathbb{B}}$, then $\varnothing \vdash_\mathbb{A} u_i$ implies $\varnothing \vdash_\mathbb{B} v_i$.

Reflexive constructions: A reflexive domain is one that satisfies an equation such as, e.g.,

$$\mathbb{D} \simeq \mathbb{A} \oplus (\mathbb{D} \to \mathbb{D}),$$

where $\mathbb{D}$ contains some *atomic* values and all the functions on itself. To construct the information system $\mathbb{D}$, given $\mathbb{A}$, the usual methods for $\oplus$ and $\to$ are followed, *but the constructions of sets of propositions and the set of consistent propositions interlace as follows*:

1. $\mathsf{inR}((u, v)) \in \mathcal{D}_\mathbb{D}$ if $u, v \in \mathsf{Con}_\mathbb{D}$;
2. $\{\ldots, \mathsf{inR}(u_i, v_i), \ldots\} \in \mathsf{Con}_\mathbb{D}$ if $\mathsf{inR}(u_i, v_i) \in \mathcal{D}_\mathbb{D}$ and

$$\left(\bigcup_{\text{for some } i\text{'s}} \{u_i\} \right) \in \mathsf{Con}_\mathbb{A}$$

implies

$$\left(\bigcup_{\text{for the } \textit{same } i\text{'s}} \{v_i\} \right) \in \mathsf{Con}_\mathbb{B}.$$

All domain constructions preserve consistent completeness and algebraicity.

Environments

The semantic function $\mathfrak{A}$ that maps language phrases to denotations uses an auxiliary argument, the *environment*. An environment is a map from a finite set of variables to domain values. The notation $\perp$ denotes the empty environment. The lookup $\rho[\![x]\!]$ denotes the image of x in the environment ρ. Extending an environment ρ to include the map $x \mapsto v$ gives the environment $\rho[x/v]$. Both these operations are strict, and hence, in particular, it is impossible to extend an environment with the map $x \mapsto \perp$.

References

[1] S. Abramsky. The lazy λ-calculus. In D. Turner, editor, *Declarative Programming*, pages 65–116. Addison Wesley, 1988.

[2] D. Bjørner and C. Jones. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[3] A. Blikle and A. Tarlecki. Naive denotational semantics. In *Proc. IFIP 9th World Computer Congress: Information Processing 83*, pages 345–355, Amsterdam, 1983. North Holland.

[4] R. Cartwright and A. Demers. The topology of program termination. In *Proc. Symposium on Logic in Computer Science*, pages 296–308, 1988.

[5] O. Danvy and A. Filinski. Abstracting control. In *Proc. 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

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

[7] M. Felleisen. On the expressive power of programming languages. In N. Jones, editor, *Proc. 1990 European Symposium on Programming*, pages 134–151. Lecture Notes in Computer Science 432, 1990.

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

[9] M. Felleisen and D.P. Friedman. A reduction semantics for imperative higher-order languages. In *Proc. Conference on Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, pages 206–223, Heidelberg, 1987. Lecture Notes in Computer Science 259, Springer Verlag.

[10] 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.

[11] 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.

[12] G.F. Johnson and D. Duggan. Stores and partial continuations as first-class objects in a language and its environment. *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 158–168, 1988.

[13] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM*, 8(2):89–101; 158–165, 1965.

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

[15] L. C.-H. Ong. Fully abstract models of the lazy lambda-calculus. In *Proc. 29th Symposium on Foundations of Computer Science*, pages 368–376, 1988.

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

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

[18] J.C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. ACM Conference*, pages 717–740, 1972.

[19] J.C. Reynolds. On the relation between direct and continuation semantics. In *Proc. International Conference on Automata, Languages and Programming*, pages 141–156, 1974.

[20] D.S. Scott. Domains for denotational semantics. ICALP, July 1982.

[21] R. Sethi and A. Tang. Constructing call-by-value continuation semantics. *J. ACM*, 27(3):580–597, 1980.

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

[23] D. Sitaram and M. Felleisen. Reasoning with continuations II: How to get full abstraction for models of control. In *Proc. 1990 Conference on Lisp and Functional Programming*, pages 161–175, 1990.

[24] J.E. Stoy. The congruence of two programming language definitions. *Theor. Comput. Sci.*, 13:151–174, 1981.

[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.