

# Delimiting the Scope of Effects

Jon G. Riecke  
AT&T Bell Laboratories  
600 Mountain Avenue  
Murray Hill, NJ 07974 USA  
riecke@research.att.com

## Abstract

Program fragments in functional languages may be observationally congruent in a language without effects (continuations, state, exceptions) but *not* in an extension with effects. We give a generic way to preserve pure functional congruences by means of an *effects delimiter*. The effects delimiter is defined semantically using the retraction techniques of [14], but can also be given an operational semantics. We show that the effects delimiter restores observational congruences between purely functional pieces of code, thus achieving a modular separation between the purely functional language and its extensions.

## 1 Introduction

Functional programming is a powerful paradigm, but it has long been recognized that purely functional programs are often inefficient and cumbersome. Many modern functional languages, *e.g.*, SML [9], build in control and state features that strictly fall outside the functional paradigm. For example, SML of New Jersey includes a “call-with-current-continuation” operation `callcc`, an exception-handling mechanism, and the ability to manipulate state using references, assignment, and dereferencing operations. The extensions are *not* simple extensions of underlying semantics of the language: old equivalences between pieces of code *not involving the new constructs* may fail to be correct in the extension.

In order to make the discussion more concrete, consider a call-by-value version of the simply-typed programming language PCF. Call-by-value PCF—VPCF for short—is a functional language with numerals, basic arithmetic operations, recursion, and a conditional test for zero. Section 2 describes the full language for those unfamiliar with the syntax and operational semantics of VPCF. In essence, VPCF is a simplified form of the purely functional fragment of SML.

VPCF permits many optimizations. For instance,

$$M_1 = \lambda a. \lambda b. \lambda c. (\lambda x. (a\ c)\ x)\ (b\ c)$$

may be replaced by the simpler term

$$M_2 = \lambda a. \lambda b. \lambda c. (a\ c)\ (b\ c)$$

in any VPCF program. One less reduction step is needed when  $M_2$  is applied to arguments, but there is another important difference:  $M_1$  and  $M_2$  evaluate subterms in different orders, *viz.*, in  $M_1$  the application  $(b\ c)$  is evaluated first whereas in  $M_2$  the application  $(a\ c)$  is evaluated first. Nevertheless, it is impossible to detect the different orders of evaluation in VPCF, *i.e.*,  $M_1$  and  $M_2$  are **observationally congruent**. A formal definition appears in Section 2, but intuitively, two terms are observationally congruent if when used in any program of type *num*, either both return the same numeral or both diverge. Observational congruence thus captures the notion of a correct optimization: replacing a term by an observationally congruent term does not change the final answer of a program.

Extending VPCF with state or control can cause optimizations like “replace  $M_1$  by  $M_2$ ” to be incorrect. For instance, suppose we extend VPCF with SML-style references, *i.e.*, add the operations  $(\text{ref } M)$ ,  $(! M)$ , and  $(M := N)$  (*cf.* Section 4 below). Then the terms  $M_1$  and  $M_2$  are *not* observationally congruent in the extended language. The context

$$\begin{aligned} C[\cdot] = & (\lambda w. [\cdot] (\lambda x. (\text{if0 } (!\ w)\ (\lambda y. 2)\ (\lambda y. 1)))) \\ & (\lambda x. (w := 3); x) \\ & 4) (\text{ref } 0) \end{aligned}$$

where  $(M; N)$  is shorthand for  $((\lambda x. N)\ M)$  with  $x$  not free in  $N$ , distinguishes the two terms:  $C[M_1]$  returns 1 whereas  $C[M_2]$  returns 2. Similarly, if we extend VPCF with a `callcc` operation (*cf.* [17] and Section 5 below), the terms  $M_1$  and  $M_2$  can be distinguished. The context

$$C[\cdot] = \text{callcc } (\lambda k. [\cdot] (\lambda x. \Omega)\ (\lambda x. k\ 1)\ 2)$$

adapted from [6], where  $\Omega$  is a divergent term, forces  $C[M_1]$  to converge with result 1 but  $C[M_2]$  to diverge. If exceptions are added to the language, then again there

Table 1: Syntactic Formation Rules for VPCF.

Variables	$x^\sigma : \sigma$		
Numerals	$0, 1, 2, \dots : num$	Recursion	$Y : ((\sigma \rightarrow \sigma) \rightarrow \sigma), \quad \sigma = (\tau \rightarrow \nu)$
Abstraction	$\frac{M : \tau}{(\lambda x^\sigma. M) : (\sigma \rightarrow \tau)}$	Application	$\frac{M : (\sigma \rightarrow \tau) \quad N : \sigma}{(M N) : \tau}$
Successor	$\frac{M : num}{(succ M) : num}$	Predecessor	$\frac{M : num}{(pred M) : num}$
If-zero	$\frac{M : num \quad N : \sigma \quad P : \sigma}{(if0 M N P) : \sigma}$	Parallel If-zero	$\frac{M : num \quad N : num \quad P : num}{(pif0 M N P) : num}$

is a context distinguishing  $M_1$  and  $M_2$ . Such examples are familiar to compiler writers and programmers. The problem is, in essence, a lack of modularity in extensions of functional languages: purely functional fragments of a program may interact with the surrounding program in ways that are not purely functional. In algebra and logic, this lack of modularity is known as a failure of *conservative extension*; that is, the set of congruences (read equations) between terms in the extended language may not contain the set of congruences in the original language (cf. [1]).

In this paper we introduce a new linguistic mechanism, written  $\#$ , and a related “protected application” operation, written  $\@$ , to separate purely functional fragments of a program from those involving state, control, or exceptions. In the case of state,  $\#$  is related to the **pure** operation of [11]; in the case of control,  $\#$  is related to the **prompt** operation of [2]. The new mechanism satisfies two important properties: first, if  $M$  is purely functional and closed and  $C[\cdot]$  is a purely functional context, then  $C[M]$  and  $C[\# M]$  return the same result; and second, if  $M, N$  are closed VPCF terms and are observationally congruent, then  $(\# M)$  and  $(\# N)$  are observationally congruent in the *extended* language. The  $\#$ -mechanism is called an “effects delimiter”, and allows the programmer to enforce distinctions between purely functional parts of a program and other, less pure fragments. The properties above show that the  $\#$ -mechanism does not change the meaning of purely functional programs, and moreover declares and forces certain portions of a program to be “purely functional”. It is, in essence, a coercion function.

Effects delimiters have a direct use in compiler optimizations and for proving the correctness of compiler optimizations. For example, placing  $\#$ ’s around purely functional code—either by hand or by some means of analysis in the compiler—allows the compiler to perform more aggressive optimizations under the  $\#$ ’s. A programmer might also directly use  $\#$ ’s to enforce distinctions between functional and non-functional portions of a program. We will give a detailed example in Sec-

tion 4 of how to use the  $\#$ -mechanism, but the utility can also be appreciated from an anecdote. Suppose a programmer is working on a program with two modules, each implemented purely functionally. Later on, the programmer may want to optimize the code and replace the first module with one that contains imperative features. Functional principles are not sound for proving the correctness of the whole program in general, but with  $\#$ ’s the programmer can still use functional reasoning in proving correctness. Alternatively, placing  $\#$ ’s around the non-functional code *forces* the code to behave purely functionally.

In order to find a meaning for  $\#$  that satisfies the right *operational* properties, we take a long excursion through *denotational* semantics. The  $\#$  mechanism can be defined for those program features whose denotational semantics can be described using certain kinds of *monads* (cf. [10, 19]). Monads are only a technical device, allowing us to state theorems with some degree of generality. Section 3 defines “strict monads”, gives a few examples, and then defines the meaning of VPCF terms with respect to an arbitrary strict monad. The proof that  $\#$  satisfies the right operational properties relies upon a deep connection between a model of VPCF and models that are built using monads. Sections 4 and 5 discuss two representative extensions of VPCF and prove the operational properties of  $\#$ . Section 6 concludes the paper with a discussion of related and future work. We assume the reader is familiar with the basic definitions of a category and functors.

## 2 Call-by-value PCF

Each term in VPCF is assigned a *simple type* of the form  $num$  or  $(\sigma \rightarrow \tau)$  for simple types  $\sigma, \tau$ . We use the symbols  $\sigma, \tau, \nu$  to denote types. The set of terms, together with their corresponding types, are defined inductively in Table 1. Following standard conventions, we use  $M, N, P$  to denote terms, and  $V$  to denote **values**, *viz.*, terms that are either constants or  $\lambda$ -abstractions. Also, the notation  $[N/x]M$  denotes the substitution of  $N$  for

Table 2: Redex Rules and Evaluation Context Rewriting for VPCF.

---

$(\lambda x. M) V$	$\rightarrow$	$[V/x]M$
$\text{if0 } 0 \ M \ N$	$\rightarrow$	$M$
$Y \ V$	$\rightarrow$	$V (\lambda x. Y \ V \ x)$
$\text{if0 } (n + 1) \ M \ N$	$\rightarrow$	$N$
$\text{succ } n$	$\rightarrow$	$n + 1$
$\text{pred } 0$	$\rightarrow$	$0$
$\text{pred } (n + 1)$	$\rightarrow$	$n$
$E[M]$	$\rightarrow_v$	$E[M']$ , if $M \rightarrow M'$
$E[\text{pif0 } M \ N \ P]$	$\rightarrow_v$	$E[n]$ , if $M \rightarrow_v^* 0$ and $N \rightarrow_v^* n$
$E[\text{pif0 } M \ N \ P]$	$\rightarrow_v$	$E[n]$ , if $M \rightarrow_v^* (n + 1)$ and $P \rightarrow_v^* n$
$E[\text{pif0 } M \ N \ P]$	$\rightarrow_v$	$E[n]$ , if $N \rightarrow_v^* n$ and $P \rightarrow_v^* n$

---

$x$  in  $M$  that avoids capture of the free variables of  $N$ .

VPCF includes a parallel conditional `pif0` to facilitate connections with denotational semantics. The inclusion of a parallel facility is not ideal. In Section 6, we argue that effects delimiters can be found for VPCF without `pif0`. In fact, the definitions are the same as for extensions to VPCF! Our *general* theorems about monads are simply more difficult to state for sequential languages, because the categories needed to interpret sequential languages are less well understood. The techniques in proving the general theorems, though, carry over to the sequential case for *particular* extensions of VPCF–`pif0`. We give an example in Section 6.

Table 2 defines a rewriting operational semantics for VPCF. Three relations are defined. The first relation, written  $\rightarrow$ , reduces redexes; there are no rules for reducing general terms that do not fit one of the forms of the left hand sides. This reduction relation will be used in our interpreters for extensions of VPCF. The second relation, written  $\rightarrow_v$ , is a relation for reducing inside a term. It is parameterized by the set of **evaluation contexts**, which for VPCF are defined by

$$E ::= [\ ] \mid (E \ M) \mid (V \ E) \mid \\ (\text{succ } E) \mid (\text{pred } E) \mid (\text{if0 } E \ M \ N) \mid (Y \ E)$$

Rewriting semantics using evaluation contexts appears in [2]. Intuitively, the hole of the evaluation context specifies the next redex to be reduced using  $\rightarrow$  in a call-by-value reduction strategy. In the term

$$((\lambda x. M) (\text{succ } 8)),$$

for instance, the next term to be reduced is `(succ 8)`; the term can be parsed  $E[\text{succ } 8]$  where  $E[\ ] = ((\lambda x. M) [\ ])$ . It is therefore important to note that *any* closed term having type *num* can be parsed into an evaluation context together with a term in the hole that can be reduced by  $\rightarrow$ . The third relation, written  $\rightarrow_v^*$ , is the reflexive, transitive closure of  $\rightarrow_v$ , and gets used in the definition of the semantics of `pif0`.

One can show that  $\rightarrow_v^*$  defines an interpreter function, *i.e.*, there is at most one value  $V$  where  $M \rightarrow_v^* V$ . The interpreter generates the notion of equivalence between code.

**Definition 1** Two VPCF terms  $M$  and  $N$  are **observationally congruent**, written  $M \equiv_{obs}^{val} N$ , if in any context  $C[\ ]$  such that  $C[M]$  and  $C[N]$  are closed terms of type *num*,  $C[M] \rightarrow_v^* k$  iff  $C[N] \rightarrow_v^* k$ .

We will also need a good denotational semantics of VPCF for the proof of correctness of our effects delimiters. The denotational semantics of VPCF is built in the category **DCPO**, whose objects are directed complete partial orders (dcpos), *i.e.*, partial orders in which the least upper bounds (lubs) of all nonempty directed sets exist, and whose morphisms are continuous functions (*cf.* [3]). A dcpo may not have a least element (unlike Scott domains), because it need not have a lub for the empty set. If a dcpo does have a least element, we call it **pointed**.

Following the style of presentation of Moggi and Sieber [10, 15], we build the spaces of meanings using **values** and **computations**. Abstractly, the semantic values are the meanings of those expressions that return syntactic values; computations, on the other hand, are the meanings of arbitrary expressions. The equations

$$\mathcal{V}_\perp[num] = \mathbb{N} \\ \mathcal{V}_\perp[\sigma \rightarrow \tau] = [\mathcal{V}_\perp[\sigma] \rightarrow \mathcal{C}_\perp[\tau]] \\ \mathcal{C}_\perp[\sigma] = (\mathcal{V}_\perp[\sigma])_\perp$$

define the dcpos of values and computations for the model of VPCF, where  $[A \rightarrow B]$  denotes the dcpo of continuous functions from  $A$  to  $B$  ordered pointwise, and  $(A)_\perp$ , the *lift* of  $A$  (*cf.* [3]), denotes the dcpo formed from  $A$  and a new element  $\perp$  ordered below every element of  $A$ . The function  $\text{lift} : A \rightarrow A_\perp$  injects values into computations. Note that computations of functional type may not be applied directly to computations;

Table 3: Denotational Semantics for VPCF.

---

$\begin{aligned} \mathcal{C}_\perp[[x^\sigma]\rho] &= \text{lift}(\rho(x)) \\ \mathcal{C}_\perp[[n]\rho] &= \text{lift}(n) \end{aligned}$	$\begin{aligned} \mathcal{C}_\perp[[Y]\rho] &= \text{lift}(\text{fix}(\lambda F. \lambda f. f(\lambda x. \text{Ap}(F f)(\text{lift } x)))) \\ \mathcal{C}_\perp[[\lambda x. M]\rho] &= \text{lift}(\lambda d. \mathcal{C}_\perp[[M]\rho[x \mapsto d]]) \end{aligned}$
$\mathcal{C}_\perp[[M N]\rho] = (\text{Ap } \mathcal{C}_\perp[[M]\rho] \mathcal{C}_\perp[[N]\rho])$	$\mathcal{C}_\perp[[\text{if}0 M N P]\rho] = \begin{cases} \mathcal{C}_\perp[[N]\rho] & \text{if } \mathcal{C}_\perp[[M]\rho] = 0 \\ \mathcal{C}_\perp[[P]\rho] & \text{if } \mathcal{C}_\perp[[M]\rho] > 0 \\ - & \text{otherwise} \end{cases}$
$\mathcal{C}_\perp[[\text{succ } M]\rho] = \begin{cases} n+1 & \text{if } \mathcal{C}_\perp[[M]\rho] = n \\ - & \text{otherwise} \end{cases}$	$\mathcal{C}_\perp[[\text{pi}f0 M N P]\rho] = \begin{cases} \mathcal{C}_\perp[[N]\rho] & \text{if } \mathcal{C}_\perp[[M]\rho] = 0 \\ \mathcal{C}_\perp[[P]\rho] & \text{if } \mathcal{C}_\perp[[M]\rho] > 0 \\ \mathcal{C}_\perp[[P]\rho] & \text{if } \mathcal{C}_\perp[[N]\rho] = \mathcal{C}_\perp[[P]\rho \\ - & \text{otherwise} \end{cases}$
$\mathcal{C}_\perp[[\text{pred } M]\rho] = \begin{cases} 0 & \text{if } \mathcal{C}_\perp[[M]\rho] = 0 \\ n & \text{if } \mathcal{C}_\perp[[M]\rho] = n+1 \\ - & \text{otherwise} \end{cases}$	

---

instead, we use the operation

$$(\text{Ap } x y) = \begin{cases} g(d) & \text{if } x = \text{lift}(g) \text{ and } y = \text{lift}(d) \\ - & \text{otherwise.} \end{cases}$$

The clauses for giving meaning to VPCF terms appear in Table 3, where the meaning of  $Y$  is taken from [15]. In these equations,  $\rho$  denotes a map from variables to values such that  $\rho(x^\sigma) \in \mathcal{V}_\perp[[\sigma]]$ , and the notation  $\rho[x \mapsto d]$  denotes an environment which returns  $d$  when applied to  $x$  and otherwise acts like  $\rho$ . Abusing notation, an expression of the form  $(\lambda x. e)$  denotes a function mapping arguments  $d$  to  $[d/x]e$ . The connection between the operational and denotational semantics is captured by the following theorem from [15]:

**Theorem 2 (Sieber)** *The model  $\mathcal{C}_\perp$  satisfies the following properties:*

1. *Adequacy:* If  $M$  is closed VPCF term of type num, then  $M \rightarrow_v^* n$  iff  $\mathcal{C}_\perp[[M]] = n$ .
2. *Full Abstraction:* If  $M, N$  are VPCF terms of the same type, then  $M \equiv_{obs}^{val} N$  iff  $\mathcal{C}_\perp[[M]\rho] = \mathcal{C}_\perp[[N]\rho]$  for all environments  $\rho$ .

### 3 Strict Monads, Monadic Semantics, and Retractions

We begin by briefly reviewing the definition of monads and then define a particular kind of monad, a *strict monad* over the category **DCPO**. We then prove the main technical result: the model  $\mathcal{C}_\perp$  is a retract of the model of VPCF over an arbitrary strict monad. This denotational result will be important in developing the operational properties of  $\#$  in Sections 4 and 5.

**3.1 Strict monads:** A **monad** over a category  $\mathcal{C}$  is a triple  $(T, \eta, \mu)$  where  $T$  is a functor from  $\mathcal{C}$  to  $\mathcal{C}$ , and

$\eta$  and  $\mu$  are natural transformations from  $Id \rightarrow T$  and  $T^2 \rightarrow T$ , viz., for any deposes  $A, B$ , the diagrams

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \eta_A \downarrow & & \downarrow \eta_B \\ TA & \xrightarrow{Tf} & TB \end{array} \quad \begin{array}{ccc} T^2A & \xrightarrow{T^2f} & T^2B \\ \mu_A \downarrow & & \downarrow \mu_B \\ TA & \xrightarrow{Tf} & TB \end{array}$$

commute. Moreover, the following diagrams must commute:

$$\begin{array}{ccc} TA & \xrightarrow{\eta_{TA}} & T^2A & \xleftarrow{T\eta_A} & TA \\ & \searrow id_{TA} & \downarrow \mu_A & \swarrow id_{TA} & \\ & & TA & & \end{array} \quad \begin{array}{ccc} T^3A & \xrightarrow{T\mu_A} & T^2B \\ \mu_{TA} \downarrow & & \downarrow \mu_A \\ T^2A & \xrightarrow{\mu_A} & TA \end{array}$$

One should think of the functor  $T$  as creating an object of *computations* from an object of *values*; then  $\eta$  is the map that forces a value to be a computation, and  $\mu$  is the map that, given a computation of a computation, returns a computation.

Many examples of monads over the category of sets and total functions are given in [10]. Monads as defined here, though, still lack the structure needed to interpret some of the constructs of VPCF. In particular, interpreting the fixpoint combinator  $Y$  as a least fixpoint operator requires partial order structure in the objects, and lubs of directed sets. This naturally leads to the category **DCPO**. The monads over **DCPO** should also respect the structure of **DCPO** and the structure of the natural numbers, and hence we arrive at the following definition:

**Definition 3** A **strict monad**  $(T, \eta, \mu, \beta)$  is a monad over **DCPO** such that  $\beta : T(\mathbb{N}) \rightarrow \mathbb{N}_\perp$  and

1. The object  $TA$  is pointed with least element  $-$ ;
2.  $T(f)(-) = -$  and  $\mu(-) = -$ ;
3.  $(\eta x) \neq -$ ; and
4. The following diagram commutes:

$$\begin{array}{ccc}
\mathbb{N} & & \\
\eta_{\mathbb{N}} \downarrow & \searrow \text{lift} & \\
T(\mathbb{N}) & \xrightarrow{\beta} & \mathbb{N}_{\perp}
\end{array}$$

The final requirement states that the representation of numbers constitutes a flat dcpo within the dcpo  $T(\mathbb{N})$ .

Many of the examples of monads in [10] can be modified to yield strict monads over **DCPO**. For instance, the state monad  $\mathcal{S} = (S, \eta^S, \mu^S, \beta^S)$  is a strict monad, which is defined as follows. First, let *Store* be the dcpo of functions mapping locations to values or unused. (A more formal definition will appear later in Section 4.) The symbol  $\pi$  denotes a store. The functor and natural transformations are defined by

$$\begin{aligned}
S(A) &= [\text{Store} \rightarrow (A \times \text{Store})_{\perp}] \\
S(f)(x)(\pi) &= \begin{cases} - & \text{if } (x \pi) = - \\ \text{lift}(f y, \pi') & \text{if } (x \pi) = \text{lift}(y, \pi') \end{cases} \\
\eta_A^S(x) &= \underline{\lambda} \pi. \text{lift}(x, \pi) \\
\mu_B^S(x)(\pi) &= \begin{cases} - & \text{if } (x \pi) = - \\ (y \pi') & \text{if } (x \pi) = \text{lift}(y, \pi') \end{cases} \\
\beta^S(x) &= \begin{cases} - & \text{if } (x \text{init}) = - \\ \text{lift}(y) & \text{if } (x \text{init}) = \text{lift}(y, \pi) \end{cases}
\end{aligned}$$

where *init* is the store mapping every location to unused. One may easily check that  $\mathcal{S}$  is a strict monad; one need only verify that the requisite diagrams commute and properties hold. The continuation monad is another example. Define  $\mathcal{K} = (K, \eta^K, \mu^K, \beta^K)$  by

$$\begin{aligned}
K(A) &= [[A \rightarrow \mathbb{N}_{\perp}] \rightarrow \mathbb{N}_{\perp}] \\
K(f)(x)(\kappa) &= (x (\underline{\lambda} n. \kappa (f n))) \\
\eta_A^K(x)(\kappa) &= (\kappa x) \\
\mu_B^K(x)(\kappa) &= (x (\underline{\lambda} m. m \kappa)) \\
\beta^K(x) &= (x \text{lift}_{\mathbb{N}}).
\end{aligned}$$

**3.2 Monadic Semantics:** What is the purpose of monads? In denotational semantics monads play a structuring role in isolating common features of different semantics. Consider, for instance, the two different semantics of a functional language with state and a functional language with control. The meaning of a natural number  $n$  in the state semantics is  $(\underline{\lambda} s. (n, s))$ —the function that takes an intermediate state  $s$  and returns  $n$  together

with the state unchanged. Likewise, in the continuation semantics for the language with control, the meaning of  $n$  is  $(\underline{\lambda} \kappa. \kappa n)$ —the function that passes  $n$  onto the rest of the program. The commonality between the two semantics should be clear: these two meanings are  $(\eta^S n)$  and  $(\eta^K n)$ . Like the numerals, the meaning of *all* of the constructs of VPCF can be defined “parametrically” with respect to the operations of a strict monad. Of course, the special operations of the language, *e.g.*, dereferencing and assignment in the case of the language with state, must be given semantics that use the structure of the particular monad.

The model  $\mathcal{C}_{\perp}$  provides the inspiration to defining the meaning of VPCF terms over a strict monad. The model  $\mathcal{C}_{\perp}$  is built upon the simplest strict monad, called the “lifting” monad since the functor part of the monad is the functor that lifts a dcpo.<sup>1</sup> By analogy, if  $\mathcal{T} = (T, \eta, \mu, \beta)$  is a strict monad we define the underlying domains of the general model by

$$\begin{aligned}
\mathcal{V}_{\mathcal{T}}[\text{num}] &= \mathbb{N} \\
\mathcal{V}_{\mathcal{T}}[\sigma \rightarrow \tau] &= [\mathcal{V}[\sigma] \rightarrow \mathcal{C}[\tau]] \\
\mathcal{C}_{\mathcal{T}}[\sigma] &= T(\mathcal{V}[\sigma]).
\end{aligned}$$

The corresponding clauses for interpreting the expressions of VPCF appear in Table 4, where  $\alpha : \mathbb{N}_{\perp} \rightarrow T(\mathbb{N})$  is defined by

$$\alpha(x) = \begin{cases} - & \text{if } x = - \\ (\eta_{\mathbb{N}} x) & \text{otherwise} \end{cases}$$

*succ* and *pred* are the semantic successor and predecessor functions, and *if0* is the expected function. The reader may care to compare the semantics in Table 4 with those in Table 3.

**3.3 Retractions:** In a model built from an arbitrary strict monad, the meaning of two terms may be different from the meanings predicted by the model  $\mathcal{C}_{\perp}$ . For example, consider the state monad  $\mathcal{S}$  defined above and the terms  $M_1$  and  $M_2$ . Then  $\mathcal{C}_{\mathcal{S}}[M_1] \neq \mathcal{C}_{\mathcal{S}}[M_2]$ , *i.e.*, one can distinguish  $M_1$  and  $M_2$  on a *semantic* level. The appropriate values to which to apply  $M_1$  and  $M_2$  are not difficult to find. The problem lies in the fact that, while  $M_1$  and  $M_2$  cannot modify the state *directly*, arguments passed to  $M_1$  and  $M_2$  may lead  $M_1$  and  $M_2$  to modify the state. If we hope to restore the equivalence between the denotations of  $M_1$  and  $M_2$ , we must force the arguments of  $M_1$  and  $M_2$  to avoid modifying the state. In other words, we must force the arguments of  $M_1$  and  $M_2$  to be in the range of the meaning function on VPCF terms.

This is the key insight: we want to turn arbitrary elements in  $\mathcal{C}_{\mathcal{S}}$  into elements that are representations of the original meanings in  $\mathcal{C}_{\perp}$ . For elements in  $\mathcal{C}_{\mathcal{S}}[\text{num}]$ , this is relatively easy: an element in  $\mathcal{C}_{\mathcal{S}}[\text{num}]$  represents an element in  $\mathcal{C}_{\perp}[\text{num}]$  if it is  $-$  or has the form

<sup>1</sup>The natural transformation  $\eta_A : A \rightarrow A_{\perp}$  is the function *lift*, and the natural transformation  $\mu_A : (A_{\perp})_{\perp} \rightarrow A_{\perp}$  collapses the doubly-lifted domain into the singly-lifted domain.

Table 4: Semantics of VPCF Over a Strict Monad.

---

$\mathcal{C}_{\mathcal{T}}[x^\sigma]\rho$	$= (\eta \rho(x))$
$\mathcal{C}_{\mathcal{T}}[\lambda x^\sigma. M]\rho$	$= (\eta (\lambda d. \mathcal{C}_{\mathcal{T}}[M]\rho[x \mapsto d]))$
$\mathcal{C}_{\mathcal{T}}[M N]\rho$	$= \mu [T (\lambda f. \mu ((T f) \mathcal{C}_{\mathcal{T}}[N]\rho)) \mathcal{C}_{\mathcal{T}}[M]\rho]$
	$= (\text{Ap } \mathcal{C}_{\mathcal{T}}[M]\rho \mathcal{C}_{\mathcal{T}}[N]\rho)$
$\mathcal{C}_{\mathcal{T}}[n]\rho$	$= (\eta n)$
$\mathcal{C}_{\mathcal{T}}[Y]\rho$	$= \eta (\text{fix } (\lambda F. \lambda f. f (\lambda x. \text{Ap } (F f) (\eta x))))$
$\mathcal{C}_{\mathcal{T}}[\text{succ } M]\rho$	$= T(\text{succ}) (\mathcal{C}_{\mathcal{T}}[M]\rho)$
$\mathcal{C}_{\mathcal{T}}[\text{pred } M]\rho$	$= T(\text{pred}) (\mathcal{C}_{\mathcal{T}}[M]\rho)$
$\mathcal{C}_{\mathcal{T}}[\text{if0 } M N P]\rho$	$= \mu (T (\lambda b. \text{if0 } b \mathcal{C}_{\mathcal{T}}[N]\rho \mathcal{C}_{\mathcal{T}}[P]\rho) \mathcal{C}_{\mathcal{T}}[M]\rho)$
$\mathcal{C}_{\mathcal{T}}[\text{pif0 } M N P]\rho$	$= \begin{cases} \alpha (\beta (\mathcal{C}_{\mathcal{T}}[N]\rho)) & \text{if } \beta (\mathcal{C}_{\mathcal{T}}[M]\rho) = 0 \\ \alpha (\beta (\mathcal{C}_{\mathcal{T}}[P]\rho)) & \text{if } \beta (\mathcal{C}_{\mathcal{T}}[M]\rho) > 0 \\ \alpha (\beta (\mathcal{C}_{\mathcal{T}}[P]\rho)) & \text{if } \beta (\mathcal{C}_{\mathcal{T}}[N]\rho) = \beta (\mathcal{C}_{\mathcal{T}}[P]\rho) \\ - & \text{otherwise} \end{cases}$

---

$(\lambda s. (n, s))$ . Thus, given an element  $e \in \mathcal{C}_S[\text{num}]$  such that  $e$  is not  $-$ , the function  $(\lambda s. (e \text{ init}, s))$  represents an element of  $\mathcal{C}_\perp[\text{num}]$ . In other words, we apply  $e$  to the map  $\delta_c^{\text{num}} = (\alpha \circ \beta)$ . We may similarly define maps  $\delta_c^\sigma : \mathcal{C}_S[\sigma] \rightarrow \mathcal{C}_S[\sigma]$  for all  $\sigma$  that force computations of  $\mathcal{C}_S$  into “legal” computations, and maps  $\delta_v^\sigma : \mathcal{V}_S[\sigma] \rightarrow \mathcal{V}_S[\sigma]$  that force values into “legal” values. Each  $\delta_c^\sigma$  and  $\delta_v^\sigma$  turns out to be a **retraction**, *i.e.*,  $(\delta_c^\sigma \circ \delta_c^\sigma) = \delta_c^\sigma$  and  $(\delta_v^\sigma \circ \delta_v^\sigma) = \delta_v^\sigma$ . In other words, converting an element twice into a “legal” computation is the same as converting it once, and the same for values.

The  $\delta_c^\sigma$ 's and  $\delta_v^\sigma$ 's are built from  $\alpha$ ,  $\beta$ , and the operations of a strict monad by induction on types. Define  $\alpha_v^\sigma \in [\mathcal{V}_\perp[\sigma] \rightarrow \mathcal{V}_\mathcal{T}[\sigma]]$ ,  $\alpha_c^\sigma \in [\mathcal{C}_\perp[\sigma] \rightarrow \mathcal{C}_\mathcal{T}[\sigma]]$ ,  $\beta_v^\sigma \in [\mathcal{V}_\mathcal{T}[\sigma] \rightarrow \mathcal{V}_\perp[\sigma]]$ , and  $\beta_c^\sigma \in [\mathcal{C}_\mathcal{T}[\sigma] \rightarrow \mathcal{C}_\perp[\sigma]]$  by

$$\begin{aligned} \alpha_v^{\text{num}} &= \beta_v^{\text{num}} = I \\ \alpha_c^{\text{num}} &= \alpha \\ \beta_c^{\text{num}} &= \beta \end{aligned}$$

$$\begin{aligned} \alpha_v^{\sigma \rightarrow \tau} (x) &= \alpha_c^\tau \circ x \circ \beta_v^\sigma \circ \delta_v^\sigma \\ \beta_v^{\sigma \rightarrow \tau} (x) &= \beta_c^\tau \circ x \circ \alpha_v^\sigma \end{aligned}$$

$$\begin{aligned} \alpha_c^{\sigma \rightarrow \tau} (x) &= \begin{cases} - & \text{if } x = - \\ \eta (\alpha_v^{\sigma \rightarrow \tau} y) & \text{if } x = \text{lift}(y) \end{cases} \\ \beta_c^{\sigma \rightarrow \tau} (x) &= \begin{cases} - & \text{if } (\beta (\text{Ap } (\eta (\lambda z. \eta 0)) x)) = - \\ \text{lift } (\beta_v^{\sigma \rightarrow \tau} (\delta_v^{\sigma \rightarrow \tau} (\lambda y. \text{ProtAp } x (\eta y)))) & \\ \text{otherwise} & \end{cases} \end{aligned}$$

$$\begin{aligned} \delta_v^\sigma &= (\alpha_v^\sigma \circ \beta_v^\sigma) \\ \delta_c^\sigma &= (\alpha_c^\sigma \circ \beta_c^\sigma) \end{aligned}$$

where  $(\text{ProtAp } d e) = \mu [T (\lambda f. \mu ((T (\delta_v^{\sigma \rightarrow \tau} f)) e)) d]$ . The  $\text{ProtAp}$  operation is just like the meaning of application, except that the result of computing  $d$ , called  $f$ , is coerced into a “legal” value.

The  $\delta_c^\sigma$ 's give meaning to an effects delimiter. In order to add the effects delimiter to the syntax of VPCF, we add the following rules to the syntax of Table 1.

Delimiter	$\frac{M : \sigma}{(\#_\sigma M) : \sigma}$
Protected Application	$\frac{M : (\sigma \rightarrow \tau) \quad N : \sigma}{(@ M N) : \tau}$

The  $@$  operator is a “protected application” operator that will make it easier to define an operational semantics for  $\#$  in Sections 4 and 5. The meanings of the new constructs are determined by

$$\begin{aligned} [(\#_\sigma M)]\rho &= (\delta_c^\sigma [M]\rho) \\ [(@ M N)]\rho &= (\text{ProtAp } [M]\rho [N]\rho). \end{aligned}$$

Given these new constructs, we may state our main technical theorem.

**Theorem 4** *Suppose  $M, N$  are closed VPCF terms of type  $\sigma$  and  $\mathcal{T} = (T, \eta, \mu, \beta)$  is a strict monad. Then the following properties hold:*

1. *Adequacy:*  $\mathcal{C}_\perp[M] = n$  iff  $\beta(\mathcal{C}_\mathcal{T}[\#_{\text{num}} M]) = n$ .
2. *Preservation:* If  $D[M]$  is a closed VPCF term of type  $\text{num}$ , then  $\mathcal{C}_\mathcal{T}[D[M]] = \mathcal{C}_\mathcal{T}[D[\#_\sigma M]]$ .
3. *Full abstraction:* For any environments  $\rho$  and  $\rho'$ ,  $\mathcal{C}_\perp[M]\rho = \mathcal{C}_\perp[N]\rho$  iff  $\mathcal{C}_\mathcal{T}[\#_\sigma M]\rho' = \mathcal{C}_\mathcal{T}[\#_\sigma N]\rho'$ .

This theorem can be extended to open terms as well by placing  $\#$ 's on the free variables. The proof, similar to proofs in [12, 14, 15], goes by showing that the model  $\mathcal{C}_\perp$  is a retract of the model  $\mathcal{C}_\mathcal{T}$  using logical relations. First, we define relations that relate values and computations in the models  $\mathcal{C}_\perp$  and  $\mathcal{C}_\mathcal{T}$ , and show that the meanings  $\mathcal{C}_\perp[M]$  and  $\mathcal{C}_\mathcal{T}[N]$  are related; this is enough to guarantee the first condition holds. We then show that the relations are surjective on the range of  $\delta_v^\sigma$  and  $\delta_c^\sigma$ ; this is enough to prove the second and third conditions. We omit the complete argument here for lack of space.

Table 5: Rules for VPCF+S

---


$$\begin{array}{l}
(E[M], s) \rightarrow_s (E[M'], s), \text{ if } M \rightarrow M' \\
(E[\text{pif0 } M \ N \ P], s) \rightarrow_s (n, s), \text{ if } (M, \text{init}) \rightarrow_s^* (0, s') \text{ and } (N, \text{init}) \rightarrow_s^* (n, s'') \\
(E[\text{pif1 } M \ N \ P], s) \rightarrow_s (n, s), \text{ if } (M, \text{init}) \rightarrow_s^* (n+1, s') \text{ and } (P, \text{init}) \rightarrow_s^* (n, s'') \\
(E[\text{pif0 } M \ N \ P], s) \rightarrow_s (n, s), \text{ if } (N, \text{init}) \rightarrow_s^* (0, s') \text{ and } (P, \text{init}) \rightarrow_s^* (n, s'') \\
(E[\text{ref } V], s) \rightarrow_s \text{new}(V, s) \\
(E[l := V], s) \rightarrow_s (E[V], s[l \mapsto V]) \\
(E[!l], s) \rightarrow_s (E[s(l)], s), \text{ if } s(l^\sigma) \neq \text{unused} \\
(E[\#_{num} M], s) \rightarrow_s (n, s), \text{ if } (M, \text{init}) \rightarrow_s^* (n, s') \\
(E[\#_{\sigma \rightarrow \tau} M], s) \rightarrow_s (E[(\#_{num} M; 0); (\lambda y. \#_\tau (@ M y))], s) \\
(E[@ (\lambda x. M) V], s) \rightarrow_s (E[\#_\tau ((\lambda x. M) (\#_\sigma V))], s)
\end{array}$$


---

#### 4 Enhancing VPCF with State

One of the ways to enhance the power of VPCF is to add state with operations similar to those in SML. We give the syntax, operational, and denotational semantics of a language called VPCF+S in this section. We also describe how the effects delimiter can be used in programs and how a compiler might optimize  $\#$ .

*4.1 Syntax and Semantics:* To the grammar of types, we add ( $\text{ref } \sigma$ ). For creating references, accessing, and updating the values held at locations, we add

Location	$l^\sigma : (\text{ref } \sigma)$
Reference	$\frac{M : \sigma}{(\text{ref } M) : (\text{ref } \sigma)}$
Dereference	$\frac{M : \text{ref } \sigma}{(! M) : \sigma}$
Assignment	$\frac{M : \text{ref } \sigma \quad N : \sigma}{(M := N) : \sigma}$

to the syntax of Table 1, where  $l^\sigma$  denotes a location. The set of locations of all types is denoted by  $\text{Loc}$ . User programs will only create locations through the  $\text{ref}$  construct and should contain no locations explicitly; locations are needed only to describe the evaluation of terms. We finally add the  $\#$  and  $@$  operations from Section 3.3. The resultant language is called VPCF+S.

An operational semantics for VPCF+S appears in Table 5 which passes a state  $s$  as an argument. For VPCF+S, a **value** is a term in the syntax

$$V ::= n \mid l^\sigma \mid (\lambda x. M).$$

A **state** is a map from  $\text{Loc}$  to the set of closed VPCF+S values or  $\text{unused}$ . A state must further respect types, *i.e.*, if  $s$  is a state and  $s(l^\sigma) = V$ , then  $V$  is a closed value of type  $\sigma$ . We also need a function  $\text{new}$  which allocates fresh locations. Formally, if  $V$  is a value of type  $\sigma$ , the expression  $\text{new}(V, s)$  returns a pair  $(l^\sigma, s[l^\sigma \mapsto V])$  where  $s(l^\sigma) = \text{unused}$ .

The operational semantics of Table 5 involves the definition of two relations,  $\rightarrow_s$  and  $\rightarrow_s^*$ , using the  $\rightarrow$  relation defined before to perform VPCF-style reductions. The  $\rightarrow_s$  relation is parameterized by a set of VPCF+S evaluation contexts, defined by

$$\begin{aligned}
E ::= & [\cdot] \mid (E \ M) \mid (V \ E) \mid \\
& (\text{succ } E) \mid (\text{pred } E) \mid (\text{if0 } E \ M \ N) \mid (Y \ E) \mid \\
& (\text{ref } E) \mid (E := M) \mid (l := E) \mid (! E) \mid \\
& (\#_{num} E) \mid (@ E \ M) \mid (@ V \ E)
\end{aligned}$$

The relation  $\rightarrow_s^*$  is the reflexive, transitive closure of  $\rightarrow_s$ . The operational rules for  $\#$  are derived from the inductive definition of  $\delta_c^\sigma$  in a somewhat mechanical way. We use  $M \equiv_{obs}^{state} N$  to denote observational congruence.

Two features of the operational semantics are worth noting. First, evaluation of a term may get “stuck” when trying to access a location which has no value associated with it. This behavior cannot occur in a program that does not mention locations directly and has no  $\#$ 's. It *can* occur, however, in a program that has  $\#$ , because the evaluation of  $\#_{num}$  involves beginning in the *initial store* where no locations are yet bound to values. If an expression under a  $\#_{num}$  tries to access a previously-created location, the evaluation cannot proceed. This is similar to operational rules for the language  $\lambda_{var}$  in [11]. Second, the evaluation crucially depends on the types of the expressions involved; the evaluation rule for  $\#$  and  $@$  is different depending on the types.

The  $\#_{num}$  rule for VPCF+S changes the character of state quite a bit; the state is no longer single-threaded (*cf.* [11]). We shall shortly see, however, that a compiler may optimize certain code fragments in a way that restores single-threadedness.

A denotational semantics may be given by making the abstract model  $\mathcal{C}_S$  more concrete. The definition of stores, left unstated in Section 3.1 above, is defined to be the solution to the domain equations

$$\begin{aligned}
\text{Store} &= [\text{Loc} \rightarrow (\text{Value} + \{\text{unused}\})_\perp] \\
\text{Value} &= \text{Loc} + \text{Nat} + [\text{Value} \rightarrow \text{Comp}] \\
\text{Comp} &= [\text{Store} \rightarrow [\text{Value} \times \text{Store}]_\perp]
\end{aligned}$$

A solution exists in the category **DCPO** by techniques due to Scott (*cf.* [4]) and Plotkin [13]; the recursive domain equations are needed because, intuitively, stores need to model states and states may be circular.

Most of the constructs of VPCF+S already have a semantics determined by the semantics over the strict monad. It is also straightforward to interpret the reference, dereference, and assignment operations in the natural way in the model  $\mathcal{C}_S$ , *e.g.*,

$$\mathcal{C}_S[\![ M ]\!] \rho \pi = \begin{cases} (\pi'(l), \pi') & \text{if } \mathcal{C}_S[\![ M ]\!] \rho \pi = (l, \pi') \\ - & \text{otherwise} \end{cases}$$

The model is adequate for the operational semantics:

**Theorem 5** *If  $M$  is a closed VPCF+S term of type  $num$ , then  $(M, s) \rightarrow_s^* (n, s')$  iff  $\mathcal{C}_S[\![ M ]\!] \rho s = (n, s')$ .*

The proof uses an inclusive predicates argument (*cf.* [3]), with additional but unsurprising complications due to the recursive definition of Store. Theorem 5 allows us to prove the syntactic analog of Theorem 4.

**Theorem 6** *Suppose  $M, N$  are closed VPCF terms of the same type (and hence contain no locations, references, dereferences, or assignments). Then*

1.  $(M, s) \rightarrow_s^* (n, s')$  iff  $(\#_{num} M, s) \rightarrow_s^* (n, s')$ .
2. If  $D[M]$  is a closed VPCF term of type  $num$ , then  $(D[M], s) \rightarrow_s^* (n, s')$  iff  $(D[\#_\sigma M], s) \rightarrow_s^* (n, s')$ .
3.  $M \equiv_{obs}^{val} N$  iff  $(\#_\sigma M) \equiv_{obs}^{state} (\#_\sigma N)$ .

**Proof:** We only prove the ( $\Rightarrow$ ) direction of Part 3 and leave the others as exercises. Suppose  $M \equiv_{obs}^{val} N$ . It follows from Theorem 2 that  $\mathcal{C}_\perp[\![ M ]\!] = \mathcal{C}_\perp[\![ N ]\!]$  and hence, by Part 3 of Theorem 4,  $\mathcal{C}_S[\![ \#M ]\!] = \mathcal{C}_S[\![ \#N ]\!]$ . Since the semantics of terms in  $\mathcal{C}_S$  is defined by induction on the structure of terms, for any context  $D[\cdot]$  the equation  $\mathcal{C}_S[\![ D[\#M] ]\!] = \mathcal{C}_S[\![ D[\#N] ]\!]$  still holds. Thus, by Theorem 5, it follows that  $(D[\#M], s) \rightarrow_s^* (n, s')$  iff  $(D[\#N], s) \rightarrow_s^* (n, s')$ , and so  $(\#M) \equiv_{obs}^{state} (\#N)$ . ■

The proof only requires the compositionality and adequacy of the model  $\mathcal{C}_S$  for VPCF+S and *not* a full abstraction theorem. In fact, this model is not fully abstract for VPCF+S (*cf.* [7]).

One could imagine proving Theorem 6 directly and skip the long excursion through denotational semantics. The technical advantage to using models, though, is that one may reason about chains of elements instead of reasoning syntactically about fixed points.

*4.2 Pragmatics:* To see how to use the  $\#$  construct in the case of VPCF+S, we implement substitution in the  $\lambda$ -calculus. The renaming of bound variables is usually coded using a “gensym” operation. We can use  $\#$  to separate this operation from the part of the program that really performs the substitution.

Suppose we extend VPCF with strings and the set of parse trees described by the syntax

$$T ::= (\text{Var } c) \mid (\text{Appl } T T) \mid (\text{Fun } c T)$$

where  $c$  ranges over the set of string constants. Each term in the syntax of trees has type *tree*, and strings have type *str*. We also add the constant *cat* for appending two strings together, *numtostr* for converting a number into a string, *eq?* which returns 0 if its arguments are equal strings and 1 otherwise, *rator* and *rand* which return the operator and operand of an application, *bvar* and *body* which return the bound variable and body of an abstraction, and *var?*, *app?*, and *fun?* which return 0 if the tree is a variable, application, or function respectively and 1 otherwise. VPCF extended with these new constants and terms is called VPCF+T, and VPCF+T extended with state is called VPCF+T+S. There is no problem extending the models of VPCF and VPCF+S with these types and operations, and the analog of Theorem 6 holds for the new language.

With some sugar for defining recursive functions and pattern matching (which can be resolved using the constants above), the code for substitution is

```
fun rename x =
  let fun gensym = (\w. \d.w := (succ (!w)); w) (ref 0)
      fun replace (Var x) c
        = if0 (eq? x c) then (Var c) else (Var x)
        | replace (Appl x y) c
        = (Appl (replace x c) (replace y c))
        | replace (Fun x y) c
        = if0 (equal? x c)
          then (Fun x y)
          else (Fun x (replace y c))
      fun renameall (Var x) = (Var x)
        | renameall (Appl x y)
        = (Appl (renameall x) (renameall y))
        | renameall (Fun x y)
        = let t = (cat (numtostr (gensym 0)) x)
          in (Fun t (replace (renameall y) t))
        end
  in renameall x

let fun subst (Var x) y = y
    | subst (Appl x y) z
    = (Appl (subst x z) (subst y z))
    | subst (Fun x y) z
    = (Fun x (subst y z))
in fun sub x y = subst (rename x) y
```

The code assumes that no variable begins with a number, so that the renaming does not create a variable that already appears in the term. This restriction is typical, for instance, in implementing a  $\lambda$ -calculus reducer. The program naturally divides itself into two parts: the first function, which renames the bound variables of a term, depends crucially on the state. The second function, however, does not access the state at all: indeed, for most uses, two successive invocations of the *sub* function could rename bound variables starting at 0.

One way to use the  $\#$  in this code, therefore, is to wrap it around the definitions of the *sub* function and the corresponding call to *rename* in the definition of *sub*. Given the analog of Theorem 6, the compiler or programmer could optimize the code of *sub* using functional principles and obtain a correct program. The  $\#$ 's guarantee that principles like “identical calls to *sub* always return the same results” are sound.

One may ask whether such a program is efficient, especially if the rules for  $\#$  are implemented directly.

Table 6: Rules for VPCF+K

---

$E[\#_{num} E'[\text{callcc } M]]$	$\rightarrow_k$	$E[\#_{num} E'[M (\lambda x. \text{abort } E'[x])]]$ ,	$E'$ contains no $\#$ 's
$E[\#_{num} E'[\text{abort } n]]$	$\rightarrow_k$	$E[n]$ ,	$E'$ contains no $\#$ 's
$E[\#_{num} n]$	$\rightarrow_k$	$E[n]$	
$E[\#_{\sigma \rightarrow \tau} M]$	$\rightarrow_k$	$E[(\#_{num} M; 0) ; (\lambda y. \#_{\tau} (@ M y))]$	
$E[@ (\lambda x. M) V]$	$\rightarrow_k$	$E[\#_{\tau} ((\lambda x. M) (\#_{\sigma} V))]$	

---

However, many optimizations can be performed. For instance, the subterm  $((\# \text{ rename}) x)$  in the new version of the program can be replaced by  $(\# (\text{rename } x))$  because the variable `rename` is a value; the optimization can be formally justified by the denotational model. A similar optimization can be performed for the outer definition of `sub`, resulting in the code

```
let fun subst (Var x) y = y
    ...
in fun sub x y = #str (subst (#str (rename x)) y)
```

Since the  $\#_{str}$  operation starts a computation in the initial store, this essentially declares that the `(ref 0)` in `gensym` is a local variable that is deallocated after it is used. After these optimizations, the compiler can deduce that the program is single-threaded.

Instead of putting  $\#$  around purely functional code, one may also put  $\#$ 's around code that manipulates the state. For instance, in the above program, one can put a  $\#$  around the definition of the `rename` function. There are no precise theorems that govern the behavior of such  $\#$ 'ed terms, but for *particular* cases, one may again use the model to obtain some intuition on how things will behave. Putting a  $\#$  around the definition of `rename` and then optimizing results in the code

```
fun rename x =
  #(let fun gensym = (\lambda w. \lambda d. w := (succ (!w)); w) (ref 0)
      ...
    in renameall x)
```

The results returned by the program are the same as when  $\#$ 's are placed around `sub`.

## 5 Enhancing VPCF with Control

One can also add features to VPCF that manipulate the flow of control, and obtain a “control delimiter” by the same techniques used above. For instance, one can add `callcc` and `abort` operations to the language by adding

$$\text{Callcc} \quad \frac{M : (\sigma \rightarrow \text{num}) \rightarrow \sigma}{(\text{callcc } M) : \sigma}$$

$$\text{Abort} \quad \frac{M : \text{num}}{(\text{abort } M) : \tau}$$

and the formation rules for  $\#$  and  $@$  defined in Section 3.3 to the rules of Table 1. We call the resultant language VPCF+K. An operational semantics for

VPCF+K, following [17], appears in Table 6, where the evaluation contexts are defined by the grammar

$$E ::= [\cdot] \mid (E M) \mid (V E) \mid \\ (\text{succ } E) \mid (\text{pred } E) \mid (\text{if0 } E M N) \mid (Y E) \mid \\ (\text{abort } E) \mid (\#_{num} E) \mid (@ E M) \mid (@ V E)$$

and where each term to be evaluated begins with  $\#_{num}$ .

The model  $\mathcal{C}_{\mathcal{K}}$  built from the monad  $\mathcal{K}$  of Section 3.1 can be used to model the `callcc` and `abort` constructs in the standard way, *e.g.*,

$$\mathcal{C}_{\mathcal{K}}[\text{abort } M]\rho = \underline{\lambda} \kappa. \mathcal{C}_{\mathcal{K}}[M]\rho (\lambda x. \text{lift}(x))$$

It is not hard to check that  $\mathcal{C}_{\mathcal{K}}$  is adequate, so

**Theorem 7** *Suppose  $M, N$  are closed VPCF terms of type  $\sigma$ . Then*

1.  $M \rightarrow_k^* n$  iff  $(\#_{num} M) \rightarrow_k^* n$ .
2. If  $D[M]$  is a closed VPCF term of type  $\text{num}$ , then  $D[M] \rightarrow_k^* n$  iff  $D[\#_{\sigma} M] \rightarrow_k^* n$ .
3.  $M \equiv_{obs}^{val} N$  iff  $(\#_{\sigma} M) \equiv_{obs}^{cont} (\#_{\sigma} N)$ .

The control delimiter is similar to Felleisen’s prompt operation [2]: the semantics of  $\#_{num}$  and `prompt` are the same, but  $\#$  extends `prompt` to higher-types.

## 6 Conclusion

Others have investigated the problem of adding state and input-output to functional languages while retaining the same observational congruences. The Haskell programming language [5], for instance, incorporates continuation-based I/O to prevent functional equivalences of code from failing. Wadler [19] also advocates using monads to add continuations, state, and other features while preserving functional equivalences. Wadler’s method requires rewriting the entire program, whereas our methods do not. But closest to this work is a recent paper of Odersky, Rabin, and Hudak [11]. They describe a call-by-name language called  $\lambda_{var}$  with assignments and an operation called `pure` that works much the same way as  $\#$  for state. Observational congruence in the new language is a conservative extension of observational congruence in the untyped  $\lambda$ -calculus with constants. One distinction between  $\#$  and `pure` is that `pure` functions do not force their arguments to

be pure (even though the bodies are forced to be pure). Odersky, Rabin, and Hudak also speculate on, but do not formulate, a similar operation for an extension with `callcc`. Our extensions are also more familiar;  $\lambda_{var}$  is a complete redesign of a  $\lambda$ -calculus with state.

Much work remains to be done. On the theoretical side, one direction is to investigate other extensions of VPCF whose semantics can be described by monads, *e.g.*, exceptions. Recent work of Sieber [16] also suggests that an extension of VPCF with nondeterminism may be straightforward. Extensions to more flexible type systems and call-by-name languages are also important. But most importantly, the theorems should be adapted for *sequential* languages. In fact, the definitions of the  $\#$  operations for VPCF+K also work for VPCF-pif0+K:

**Theorem 8** *Suppose  $M$  and  $N$  are closed VPCF-pif0 terms of type  $\sigma$ . Then the operational semantics of Table 6 satisfies the three properties of Theorem 7.*

The proof techniques used in proving Theorem 4 work in a Milner-style model of VPCF-pif0 (*cf.* [8]). We are currently investigating whether such theorems can be extended to more general situation, *e.g.*, VPCF-pif0+S.

There is also a large amount of practical work left to be done. For instance, the optimizations described in Section 4 should have similar counterparts in VPCF+K and other language extensions. These optimizations are crucial in making the effects delimiters practical.

The development in this paper is in itself interesting. Other papers have pointed to the usefulness of denotational methods, *e.g.*, [17, 18] in extending languages with control features. This paper also shows how denotational semantics can play a role in the design of a useful programming language feature.

*Acknowledgements:* I thank Matthias Felleisen, Carl Gunter, and Ramesh Subrahmanyam for helpful conversations, and the referees, Tim Griffin, and Phil Wadler for comments on drafts of this paper. This work was partially supported by an NSF Graduate Fellowship, NSF grant number CCR-8912778, NRL N00014-91-J-2022, and NOSC 19-920123-31.

## References

- [1] V. Breazu-Tannen and A. R. Meyer. Computable values can be classical. In *Fourteenth Symposium on Principles of Programming Languages*, pages 238–245. ACM, 1987.
- [2] M. Felleisen. The theory and practice of first-class prompts. In *Fifteenth Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.
- [3] C. A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.
- [4] C. A. Gunter and D. S. Scott. Semantic domains. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. Elsevier, 1990.
- [5] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.
- [6] A. R. Meyer and J. G. Riecke. Continuations may be unreasonable (preliminary report). In *Proceedings of the 1988 Conference on Lisp and Functional Programming*, pages 63–71. ACM, 1988.
- [7] A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables. In *Fifteenth Symposium on Principles of Programming Languages*, pages 191–203. ACM, 1988.
- [8] R. Milner. Fully abstract models of the typed lambda calculus. *Theor. Comp. Sci.*, 4:1–22, 1977.
- [9] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [10] E. Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.
- [11] M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment, and the lambda calculus. In *Twentieth Symposium on Principles of Programming Languages*, pages 43–56. ACM, 1993.
- [12] G. D. Plotkin. Notes on completeness of the full continuous type hierarchy. Unpublished manuscript, MIT. November 1982.
- [13] G. D. Plotkin. Types and partial functions. Handwritten manuscript, February 1984.
- [14] J. G. Riecke. Fully abstract translations between functional languages (preliminary report). In *Eighteenth Symposium on Principles of Programming Languages*, pages 245–254. ACM, 1991.
- [15] K. Sieber. Relating full abstraction results for different programming languages. In *Foundations of Software Technology and Theoretical Computer Science*, volume 472 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1990.
- [16] K. Sieber. Call-by-value and nondeterminism. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1993.
- [17] D. Sitaram and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 Conference on Lisp and Functional Programming*, pages 161–175. ACM, 1990.
- [18] D. Sitaram and M. Felleisen. Modeling continuations without continuations. In *Eighteenth Symposium on Principles of Programming Languages*, pages 185–196. ACM, 1991.
- [19] P. Wadler. The essence of functional programming. In *Nineteenth Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.