Isolating Side Effects in Sequential Languages

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

Ramesh Viswanathan*
Stanford University
Department of Computer Science
Stanford, CA 94305, USA
Email: vramesh@theory.stanford.edu

Abstract

It is well known that adding side effects to functional languages changes the operational equivalences of the language. We develop a new language construct, encap, that forces imperative pieces of code to behave purely functionally, *i.e.*, without any visible side effects. The coercion operator encap provides a means of extending the simple reasoning principles for equivalences of code in a functional language to a language with side effects. In earlier work [36], similar coercion operators were developed, but their correctness required the underlying functional language to include parallel operations. The coercion operators developed here are simpler and are proven correct for *purely sequential* languages. The sequential setting requires the construction of fully abstract models for sequential call-by-value languages and the formulation of a weak form of "monad" suitable for expressing the semantics of call-by-value languages with side effects.

1 Introduction

Two pieces of code are "operationally equivalent" if in any program, one can be replaced by the other without affecting the observable outcome of the program. A good understanding of operational equivalence can facilitate modular development of programs and make it possible to prove the soundness of various optimizations. It is therefore an advantage of a *purely* functional language, such as Haskell [13], that it supports some well-understood principles for reasoning about operational equivalence. Purely functional languages, however, lack features that make it possible to write more efficient programs. For this reason, many functional languages, *e.g.*, Standard ML [22] and Scheme [5], include control operations and ways of manipulating memory. Such side-effecting features do not come without cost: simple reasoning principles may fail to be sound, as two purely functional pieces of code may be equivalent in purely functional contexts but not in contexts with side effects.

*Supported in part by NSF Grant CCR-9303099 and the Powell Foundation. This work was begun while visiting AT&T Bell Laboratories during the summer of 1993.

To appear in the Twenty-Second ACM Symposium on Principles of Programming Languages, January 1995, San Francisco, California.

To illustrate the problem, consider a call-by-value version of PCF—here called VPCF (*cf.* [9, 36, 38])—which is formally defined in Section 3.1. VPCF is a purely functional language, but we can add operations ref, !, := for manipulating the state to obtain another language called VPCF+S. Briefly, ref x allocates a new cell initialized to x, $(r := x)$ updates the contents of the cell r to x, and $!r$ returns the current contents of the cell r. (The typing rules and formal operational semantics for VPCF+S appear in Section 3.3.) Two terms may be operationally equivalent in VPCF but not in the extended language. For example, consider the expressions

$$M_1 = \lambda x^{\mathrm{num}}.\, \lambda f^{\mathrm{num}\to\mathrm{num}}.\, \lambda g^{\mathrm{num}\to\mathrm{num}}.\, (f\ x); (g\ x)$$
$$M_2 = \lambda x^{\mathrm{num}}.\, \lambda f^{\mathrm{num}\to\mathrm{num}}.\, \lambda g^{\mathrm{num}\to\mathrm{num}}.\, (f\ x); (f\ x); (g\ x)$$

where ; is the usual sequencing operator (definable by the equation $(M; N) = ((\lambda d.\, N)\ M)$ for $d \notin FV(N)$ in VPCF). M_1 and M_2 are operationally indistinguishable in VPCF. However, in VPCF+S, they can be distinguished by the context

$$C[\cdot] = (\lambda r.\, [\cdot]\ 0\ (\lambda d.\, r := \mathsf{succ}\ (!r))\ (\lambda d.\, !r))\ (\mathsf{ref}\ 0).$$

$C[M_1]$ returns 1 whereas $C[M_2]$ returns 2.

This example, which shows how adding assignments fundamentally changes the character of VPCF, runs counter to a vague intuition that the VPCF is somehow "embedded" in VPCF+S, and that reasoning principles on the purely functional core extend easily to the language with state. In this paper we show how to recover that intuition. More precisely, we show that VPCF is embedded in an extension with side effects, *but mediated by the application of an additional operation* encap. The encap operation "encapsulates" side effects: the application of encap to some part of a program prevents its side effects from being visible to the rest of the program, and conversely, encap prevents any potential side effects of the surrounding program from affecting the behaviour of the code under encap. However, it does not change the "functional" behaviour of a term; formally, if M is a purely functional piece of code, then for any purely functional context $C[\cdot]$, the programs $C[M]$ and $C[\mathsf{encap}\ M]$ yield the same observable behaviour. As a consequence, the encap operation restores old operational equivalences, *i.e.*, two terms M and N are operationally equivalent in VPCF iff $(\mathsf{encap}\ M)$ and $(\mathsf{encap}\ N)$ are operationally equivalent in VPCF+S+encap. Finally, the operation encap forces non-functional code to behave purely functionally; we give a theorem illustrating this in Section 5.

We can now give some intuition for the behaviour of encap when applied to a term M of VPCF+S+encap. In executing (encap M), the current state of the memory is saved, say on disk, and M is executed with the contents of all memory locations flushed. If M is of base type, any resulting value is returned. If M is of function type, the resulting closure is returned, with its argument and body coerced by encap. In either case, memory is then restored to the saved state. Thus the execution of (encap M) does not change the state of memory and its result does not depend on the state in which it is executed.

The uses of encap are twofold. First, one may use encap to enforce certain portions of code to be oblivious to side effects caused by other modules; this would allow a programmer to aggressively optimize the code under a encap later since functional reasoning principles remain valid for purely functional code under an encap. Second, one may use encap to establish facts about code. If one proves that adding encap to a subterm M does not change the meaning of the whole term, then any side effects of M are purely local and so the proof of correctness of the whole term can be carried out using functional reasoning principles. Section 6 below outlines an example of proving program correctness using encap.

The idea of using coercion operations to encapsulate side effects is not new; in [36], Riecke adds a similar operation for functional languages extended by imperative features. In broad outline, the design and proof of correctness of encap in [36] relies on a connection between a model of the underlying functional language and a model of its extension with a side-effecting operation. This connection between the two models is easiest to establish in a general semantic framework of "strict monads" over a category, which are monads with additional structure (see [23, 43] for a discussion of monads). The proofs in [36] then require that the model of the extension must be definable from a strict monad. Moreover, the proofs rely crucially on two properties: the model of the functional language must be fully abstract, *i.e.*, operational equivalence must match denotational equivalence, and the model of the extension must be adequate (*cf.* [9]). In order to use the most natural semantic models, the full abstraction requirement forces the languages to include a "parallel if" (*cf.* [9, 31]). This operation, found in no practical programming language, is the main obstacle to applying the results.

The main contribution of this paper is the design and proof of correctness of encap for *purely sequential* programming languages. Our operational semantics for encap is also simpler than that of [36], making reasoning with encap and implementing it much easier. As applications, we describe the operational rules for encap when added to a language with state and to one with continuations. The operational rules for isolating other side effects, such as exceptions and non-determinism, can also be immediately obtained from our general framework.

Adapting the proofs of [36] for the sequential setting is not straightforward. There are three key insights. First, since general categories for sequential languages are not well understood, it is essential to work over a *call-by-value model* instead of a category. Section 2 develops an appropriate definition of a "call-by-value language" and "call-by-value model". Second, the naïve extension of "strict monads" to a call-by-value model does not work because a call-by-value model itself arises from a lifting monad over a category C, and the natural monads for side effects defined over C fail to be monads when extended to the lifted category. We develop the definition of a *call-by-value monad* in Section 4; these call-by-value monads are weak enough to express all the side effects of interest; yet, has enough structure to entail the desired semantic connections. Third, a close examination of the proofs in [36] reveals that the fully abstract model for the base language VPCF need only be constructed out of continuous, extensional functions; other mathematically attractive properties of the model are simply not needed. In a companion paper [38], we show how to construct—using techniques of Milner [21]—fully abstract models of VPCF and FPC (a sequential language with recursive types [9], in which the semantics of VPCF+S can be encoded), called $\mathcal{V}$ and $\mathcal{S}$. While our proofs only require the model $\mathcal{V}$ to be fully abstract, the intimate connection between $\mathcal{V}$ and $\mathcal{S}$ requires that the model $\mathcal{S}$ also not contain "parallel" functions which are missing from the fully abstract model $\mathcal{V}$ (*cf.* [31]). The existence of continuous, extensional, fully abstract models of VPCF and FPC is thus crucial. However, because our framework is applicable to any call-by-value model, the details of the construction in [38] play no role in the proofs.

2 Call-by-value Languages

2.1 Syntax and Operational Semantics

The type signature of a general functional language is a set Σ that is closed under $\rightarrow$, *viz.*, if $\sigma, \tau \in \Sigma$, then $(\sigma \rightarrow \tau) \in \Sigma$. A set of typing rules for functional languages appears in Table 1; all variables come tagged with types. The typing rules are parameterized by a set C of **term constructors**, which are tuples of the form $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma \rangle$ where $n \geq 0$ and $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma$. We chose term constructors, as opposed to term constants, to allow the possibility of non-strict operators in the call-by-value language.

We use a parameterized rewriting semantics, with two basic relations, to specify operational behaviour. The first relation, $\rightsquigarrow$, reduces redexes with $M \rightsquigarrow M'$ indicating that M is a redex reducing directly to M'. The second relation, $\rightarrow$, is the basic one-step evaluation relation for the particular language, which is defined using the notion of an **evaluation context** to impose an order on the evaluation of subexpressions [7]. Table 1 specifies a general rewriting framework for call-by-value languages. The only redex rule common to all call-by-value languages is the β-reduction rule; here, V ranges over a set of **values**—expressions that cannot be further evaluated—which varies among languages but always includes λ-abstractions. The evaluation contexts specify the next redex using a left-to-right order of evaluation for applications. To give the semantics of a particular language, one must add rules for $\rightsquigarrow$ to specify the behaviour of term constructors, and add new productions to the grammar of evaluation contexts to reflect the order of evaluation.

2.2 Call-by-value Models

A denotational model for a call-by-value language is based on a collection of sets, indexed by type, suitable for giving meaning to terms. The following definition is taken from [35] and resembles definitions from [1, 28].

Definition 1 A **call-by-value type frame** over a type signature Σ is a tuple

$$\mathcal{D} = (\{D^\sigma \mid \sigma \in \Sigma\}, \{\bot^\sigma \mid \sigma \in \Sigma\}, \{App^{\tau,\nu} \mid \tau, \nu \in \Sigma\})$$

Table 1: Typing and Evaluation Rules for Call-by-Value Functional Languages.

Terms	(var)	$x^\sigma : \sigma$	$(const)$	$\dfrac{\langle f;\sigma_1,\ldots,\sigma_n \Rightarrow \sigma\rangle \in C \quad M_i : \sigma_i}{f(M_1,\ldots,M_n) : \sigma}$
	(abs)	$\dfrac{M : \tau}{(\lambda x^\sigma.\, M) : (\sigma \to \tau)}$	(app)	$\dfrac{M : (\sigma \to \tau) \quad N : \sigma}{(M\ N) : \tau}$
Evaluation	$(\lambda x.\, M)\ V \ \rightsquigarrow\ [V/x]M$ $E \ ::= \ [\cdot] \mid (E\ M) \mid (V\ E) \mid \ldots$			

where each D^σ is a poset, $\bot^\sigma \in D^\sigma$, and $App^{\tau,\nu} : D^{\tau\to\nu} \times D^\tau \to D^\nu$ such that

1. For all $d \in D^\sigma$, $\bot^\sigma \leq d$.
2. If either $d = \bot^{\tau\to\nu}$ or $e = \bot^\tau$, then $App^{\tau,\nu}(d,e) = \bot^\nu$.
3. $f \leq_{D^{\tau\to\nu}} g$ iff ($f \neq \bot^{\tau\to\nu}$ implies $g \neq \bot^{\tau\to\nu}$) and for all $d \leq_{D^\tau} e$, $App^{\tau,\nu}(f,d) \leq_{D^\nu} App^{\tau,\nu}(g,e)$.

We use $\Sigma(\mathcal{D})$ to denote the type signature over which the type frame $\mathcal{D}$ is defined. As notational conventions, we omit the type superscripts on $\bot$ and App when obvious from the context, and write $(f\ x)$ for $App(f,x)$ when clear . Also, we write $d\!\uparrow$ whenever $d = \bot$ and $d\!\downarrow$ whenever $d \neq \bot$.

- The type frame $\mathcal{D}$ is **flat** at type $\sigma \in \Sigma(\mathcal{D})$ if for any $d,e \in D^\sigma$, $d\!\downarrow$ and $d \leq e$ implies $d = e$.
- The type frame $\mathcal{D}$ is **continuous** if each D^σ is a cpo, and for any $f \in D^{\tau\to\nu}$ and directed set $S \subseteq D^\tau$, $App(f, \bigsqcup S) = \bigsqcup\{App(f,e) \mid e \in S\}$.

The element $\bot$ is the semantic representation of a divergent computation. The conditions on a call-by-value frame capture the call-by-value parameter passing mechanism and the fact that non-divergent elements of a function type behave monotonically and extensionally.

Let $\mathcal{D}$ be a type frame over Σ. A **$\mathcal{D}$-environment** is a type-respecting map from variables to elements of $\mathcal{D}$, *i.e.*, $\rho(x^\sigma) \in D^\sigma$. A **call-by-value model** for a language with types Σ and term constructors C is then a type frame $\mathcal{D}$ together with functions $\mathcal{D}[\![f]\!] : D^{\sigma_1} \times \cdots \times D^{\sigma_n} \to D^\sigma$ for each $\langle f;\sigma_1,\ldots,\sigma_n \Rightarrow \sigma\rangle \in C$, with enough elements to interpret every term of the language, *i.e.*, there is a meaning function $\mathcal{D}[\![\cdot]\!]$ satisfying

$$\begin{array}{rcl}
\mathcal{D}[\![x^\sigma]\!]\rho & = & \rho(x^\sigma) \\
\mathcal{D}[\![f M_1 \ldots M_k]\!]\rho & = & \mathcal{D}[\![f]\!](\mathcal{D}[\![M_1]\!]\rho,\ldots,\mathcal{D}[\![M_k]\!]\rho) \\
\mathcal{D}[\![M\ N]\!]\rho & = & App(\mathcal{D}[\![M]\!]\rho, \mathcal{D}[\![N]\!]\rho) \\
\mathcal{D}[\![\lambda x^\sigma.\, M]\!]\rho & = & g, \text{ where } g\!\downarrow \text{ and for any } d\!\downarrow, \\
 & & \quad App(g,d) = \mathcal{D}[\![M]\!]\rho[x^\sigma \mapsto d]
\end{array}$$

and such that $\mathcal{D}[\![M]\!]\rho \in D^\sigma$ for any term M of type σ and any $\mathcal{D}$-environment ρ.

3 VPCF and its Extensions

In this section, we define our basic functional language, VPCF, and two extensions. The operational semantics of each language specifies an interpreter function $Eval_L$ which is a partial function from programs, *i.e.*, closed terms of base type num to numerals. The evaluation function determines the operational equivalence relation of the language.

Definition 2 Two terms M and N in L are **operationally equivalent**, written $M \equiv_L N$, if in any context $C[\cdot]$ such that $C[M]$ and $C[N]$ are programs, $Eval_L(C[M]) = n$ iff $Eval_L(C[N]) = n$.

3.1 VPCF

VPCF is a purely functional language with basic arithmetic and a mechanism for recursive function definitions. Its types are given by the grammar

$$\sigma \ ::= \ \mathsf{num} \mid (\sigma \to \sigma)$$

and the term constructors are

$$\begin{array}{l}
\langle n;\mathsf{num}\rangle \ \text{ for all } n \geq 0 \\
\langle \mathsf{succ};\mathsf{num} \Rightarrow \mathsf{num}\rangle \\
\langle \mathsf{pred};\mathsf{num} \Rightarrow \mathsf{num}\rangle \\
\langle \mathsf{if0}^\sigma;\mathsf{num},\sigma,\sigma \Rightarrow \sigma\rangle \\
\langle \mathsf{Y}_\sigma;(\sigma\to\sigma)\to\sigma\rangle \ \text{ where } \sigma \neq \mathsf{num}.
\end{array}$$

The syntax of VPCF is now completely defined by Table 1.

In addition to the redex rule in Table 1, VPCF has the redexes given in Table 2 which reflect the behaviour of its term constructors. The additional productions for evaluation contexts (beyond those of Table 1) are also given in Table 2, where a value V is a numeral n, a λ-abstraction, or Y_σ. One can verify that every closed non-value can be written uniquely as $E[R]$ with R a redex. The basic one-step evaluation relation $\to_v$ for closed VPCF terms is defined by

$$E[M] \ \to_v \ E[M'], \ \text{ if } M \rightsquigarrow M'.$$

Define $Eval_v(M) = n$ iff $M \twoheadrightarrow_v n$, where $\twoheadrightarrow_v$ is the reflexive, transitive closure of $\to_v$.

3.2 VPCF+K

Control operations can be added to VPCF by adding the term constructors $\langle \mathsf{abort};\mathsf{num} \Rightarrow \sigma\rangle$ for terminating computations and $\langle \mathsf{callcc};((\sigma \to \mathsf{num}) \to \sigma) \Rightarrow \sigma\rangle$ for capturing continuations. The resulting language is called VPCF+K (for "control").

Table 2 specifies the one-step evaluation relation $\to_c$ and set of evaluation contexts for VPCF+K. The last two rules for $\to_c$ give the formal semantics of the new operations: callcc captures the current evaluation context, makes it into

Table 2: Operational Semantics of VPCF, VPCF+K, and VPCF+S

VPCF	$\mathsf{succ}(n) \rightsquigarrow n+1 \qquad \mathsf{if0}(0,M,N) \rightsquigarrow M$ $\mathsf{pred}(0) \rightsquigarrow 0 \qquad \mathsf{if0}(n+1,M,N) \rightsquigarrow N$ $\mathsf{pred}(n+1) \rightsquigarrow n \qquad (\mathsf{Y}\ V) \rightsquigarrow (V\ (\lambda x.\ \mathsf{Y}\ V\ x))$ $E ::= \ldots \mid \mathsf{succ}(E) \mid \mathsf{pred}(E) \mid \mathsf{if0}(E,M,N) \mid (\mathsf{Y}\ E) \mid \ldots$
VPCF+K	$E[M] \rightarrow_c E[M'] \quad \text{if } M \rightsquigarrow M'$ $E[\mathsf{callcc}(M)] \rightarrow_c E[M\ (\lambda x.\ \mathsf{abort}(E[x]))]$ $E[\mathsf{abort}(n)] \rightarrow_c n$ $E ::= \ldots \mid \mathsf{abort}(E)$
VPCF+S	$(E[M],s) \rightarrow_s (E[M'],s) \quad \text{if } M \rightsquigarrow M'$ $(E[\mathsf{ref}\ V],s) \rightarrow_s (E[l],s[l \mapsto V]) \quad \text{if } l \notin dom(s)$ $(E[!l],s) \rightarrow_s (E[s(l)],s)$ $(E[l := V],s) \rightarrow_s (E[V],s[l \mapsto V])$ $E ::= \ldots \mid \mathsf{ref}\ (E) \mid !(E) \mid (E := M) \mid (V := E)$

a λ-abstraction, and passes it to the argument of callcc; abort discards the evaluation context. The interpreter function $Eval_k$ is defined as for VPCF.

3.3 VPCF+S

One may add Standard ML-style references and assignments to VPCF (*cf.* [22]). The enhanced language, called VPCF+S (for "state"), has types generated by the grammar

$$\sigma ::= \mathsf{num} \mid (\sigma \to \sigma) \mid (\mathsf{ref}\ \sigma)$$

where $(\mathsf{ref}\ \sigma)$ is the type of locations holding values of type σ. In addition to the VPCF term constructors, VPCF+S has term constructors $\langle l^\sigma; (\mathsf{ref}\ \sigma)\rangle$, where l is drawn from an infinite set of location constants; $\langle \mathsf{ref}\ ; \sigma \Rightarrow (\mathsf{ref}\ \sigma)\rangle$ for creating references; $\langle !; (\mathsf{ref}\ \sigma) \Rightarrow \sigma\rangle$ for dereferencing; and $\langle :=; (\mathsf{ref}\ \sigma), \sigma \Rightarrow \sigma\rangle$ for assignment, written in infix notation. The location constants l^σ do not appear in user programs (this would be tantamount to allowing the user to pick addresses), but help in defining the operational semantics.

The evaluation relation $\rightarrow_s$ and the additional evaluation contexts of VPCF+S appear in Table 2. Values in VPCF+S include locations l in addition to the values of VPCF. The operational rules manipulate a state s in addition to terms; a **state** is simply a finite map from a set of locations to values. The operational rules for the assignments and dereferencing are standard (*cf.* [9]), and the evaluation function $Eval_s$ is defined as $Eval_s(M) = n$ iff $(M, \emptyset) \twoheadrightarrow_s (n, s')$ for some state s', where $\emptyset$ is the empty map denoting the initial store.

4 Call-by-value Monads and Coercions

We design and prove the correctness of a coercion operator for an extension of VPCF in two steps. First, we show that a continuous model $\mathcal{D}$ for VPCF can be regarded as a retract of a model $\mathcal{T}$ for an extension of VPCF. The retractions act as semantic coercion operators, *viz.*, they coerce any element of $\mathcal{T}$ into one with the same functional behaviour but which has no side effects. Second, we use the semantic coercions to design the operational rules for the coercion operators encap. The operational facts are proven through model-theoretic facts: if the model $\mathcal{D}$ is fully abstract for VPCF, and the model $\mathcal{T}$ is adequate for the extension of VPCF together with the encap operator, then the properties of the retractions imply the desired operational correctness properties of encap.

In this section we define a general framework in which the semantics of extensions of VPCF can be described using a "call-by-value monad" and the model that it defines, and prove that any call-by-value model $\mathcal{D}$ of VPCF is a retraction of the model determined by a call-by-value monad. We defer the application of the results to Section 5.

4.1 Call-by-value Monads

Ideally, monads are built over categories with some abstract structure, *e.g.*, the notion of a "strict monad" over **DCPO** defined in [36]. Call-by-value monads are, however, defined over *call-by-value models*, not over categories. There is a good reason for this: constructing categories suitable for *sequential* languages like VPCF (see, *e.g.*, [3, 4, 6]) has been open for many years (although recent announcements [2, 15, 26] suggest that a categorical framework can be constructed). Moreover, by defining call-by-value monads over models, we can state and prove our results independent of the details of the construction of any particular model.

Our definition of a call-by-value monad is inspired by the notion of a "strict monad" defined in [36] ("strict monads" are monads [23] with some additional structure).

Definition 3 Let $\mathcal{D}$ be a **least fixpoint model**, *i.e.*, a continuous call-by-value model with $\mathcal{D}[\![\mathsf{Y}_\sigma]\!] = \bigsqcup_n F^n_\sigma$, where

$$F^0_\sigma = \bot^{((\sigma\to\sigma)\to\sigma)}$$
$$F^{n+1}_\sigma = \mathcal{D}[\![\lambda f^{(\sigma\to\sigma)}.\ f(\lambda x.\ G\ f\ x)]\!][G \mapsto F^n_\sigma],$$

and where $\mathcal{D}$ is flat at a type $\mathsf{num} \in \Sigma(\mathcal{D})$. A **call-by-value monad** over $\mathcal{D}$ is a tuple $\langle T, \eta, \mu, \beta\rangle$ with $T : \Sigma(\mathcal{D}) \to \Sigma(\mathcal{D})$ and a family of functions $T_{\tau,\nu} : D^{\tau\to\nu} \to D^{T(\tau)\to T(\nu)}$, $\eta_\sigma : D^\sigma \to D^{T(\sigma)}$, and $\mu_\sigma : D^{T(T(\sigma))} \to D^{T(\sigma)}$ for each type $\sigma, \tau, \nu \in \Sigma(\mathcal{D})$ satisfying the following properties:

1. The functions $T_{\tau,\nu}, \eta_\sigma, \mu_\sigma$ are **interpretable** in $\mathcal{D}$. That is, when the meaning function of $\mathcal{D}$ is extended to interpret new constructors

$$\langle T_{\tau,\nu}; (\tau \to \nu) \Rightarrow (T(\tau) \to T(\nu)) \rangle,$$
$$\langle \eta_\sigma; \sigma \Rightarrow T(\sigma) \rangle, \text{ and}$$
$$\langle \mu_\sigma; T(T(\sigma)) \Rightarrow T(\sigma) \rangle$$

using the functions, the meaning of any term over the new constructors still exists in the model.

2. For any $x \in D^\sigma$ and $f \in D^{\sigma\to\tau}$, if $f\!\downarrow$, then $\eta_\tau(f\ x) = T_{\sigma,\tau}(f)\ (\eta_\sigma\ x)$.

3. For any $x \in D^{T(\sigma)}$,
   - If $x\!\downarrow$, then $\mu_\sigma(\eta_{T(\sigma)}\ x) = x$.
   - If $x\!\uparrow$, then $\mu_\sigma(\eta_{T(\sigma)}\ x) = \eta_\sigma(\bot^\sigma)$.

4. For any $x, y \in D^\sigma$, if $\eta_\sigma(x) \le \eta_\sigma(y)$, then $x \le y$.

5. For any $x \in D^{\mathrm{num}}$, $\beta(\eta_{\mathrm{num}}(x)) = x$.

If one takes the elements of a call-by-value model as defining the morphisms of a category, then a call-by-value monad differs from a monad over this category in two respects. Firstly, we do not require the maps η_σ and μ_σ to be elements of the model. In particular, they do not have to be strict functions. Secondly, the equations that they need to satisfy are weaker forms of the naturality and unit laws of a monad. Indeed, all the side effects of interest would fail to satisfy the stronger monad laws over a call-by-value model. For example, in the monads given in Section 5, η is not strict and they only satisfy the modified monad laws expressed in this definition. The weaker properties of call-by-value monads will, however, be strong enough to prove the semantic connections established in Section 4.3.

Monads capture the notion of "computations" that consist of results and side effects [23]. However, unlike categorical monads, the notion of divergence is already represented in results; this is the reason for the departure from the definition of a categorical monad. Thus computations only incorporate the possibility of side effects. A call-by-value monad provides basic operations for manipulating computations: the type $T(\sigma)$ represents the set of computations whose results are of type σ; the function $T_{\tau,\nu}(f)$ transforms the result of a computation by f without changing its side effect ; η maps a result d to a computation with no side effect; μ composes two computations to give the cumulative computation; and β extracts the result from a computation of type num. The conditions in the definition of a call-by-value monad simply express some natural and general properties of side effects.

4.2 Monadic Semantics

A call-by-value monad over a model of VPCF generates a new interpretation of application and the VPCF constructs—one which passes along side effects. The meaning of a VPCF term thus never generates any side effects itself. The type constructor T of the call-by-value monad is used to promote the types. More precisely, for any VPCF type σ, let $\sigma', \sigma^* \in \Sigma(\mathcal{D})$ be defined by induction on the structure of σ as follows.

$$\mathsf{num}' = \mathsf{num}$$
$$(\sigma \to \tau)' = (\sigma' \to \tau^*)$$
$$\sigma^* = T(\sigma')$$

Let the domains of results and computations in $\mathcal{T}$ be given by $\mathcal{R}_T[\![\sigma]\!] = D^{\sigma'}$ and $\mathcal{C}_T[\![\sigma]\!] = D^{\sigma^*}$.

We can now give the meaning of VPCF terms. For any VPCF term M of type σ, $\mathcal{T}[\![M]\!]\rho_\mathcal{T}$ is an element of $\mathcal{C}_T[\![\sigma]\!]$, where $\rho_\mathcal{T}$ is a **$\mathcal{T}$-environment**, *i.e.*, a map from variables to $\mathcal{D}$ such that $\rho_\mathcal{T}(x^\sigma) \in \mathcal{R}_T[\![\sigma]\!]$. The clauses for interpreting the VPCF constructs are given in Table 3, which use the functions of the monad and the semantic elements $\mathsf{Ap}^{\sigma,\tau}$ defined as

$$\mathsf{Ap}^{\sigma,\tau} = \underline{\lambda} f \in D^{(\sigma\to\tau)^*}.\ \underline{\lambda} x \in D^{\sigma^*}.$$
$$\mu\ [T\ (\underline{\lambda} g \in D^{(\sigma\to\tau)'}.\ \mu\ ((T\ g)\ x))\ f]$$

and where $(\underline{\lambda} x.\ e)$ is a meta-notation to denote elements of the model. Requirement (1), that the call-by-value monad operations be "interpretable", ensures that these definitions are good definitions.

4.3 Semantic Coercions

We are now ready to establish the key semantic result—the model $\mathcal{D}$ is a retract of the model $\mathcal{T}$. Let $\langle T, \eta, \mu, \beta \rangle$ be a call-by-value monad over the model $\mathcal{D}$. For each VPCF type σ, Table 4 defines elements $\alpha_r^\sigma \in D^{\sigma\to\sigma'}, \beta_r^\sigma \in D^{\sigma'\to\sigma}$, and functions $\alpha_c^\sigma : D^\sigma \to \mathcal{C}_T[\![\sigma]\!], \beta_c^\sigma : \mathcal{C}_T[\![\sigma]\!] \to D^\sigma$. (Note: The elements $\alpha_r^\sigma, \beta_r^\sigma$ are thus strict functions.) Intuitively, the function α_c^σ embeds elements of D^σ into computations in $\mathcal{C}_T[\![\sigma]\!]$ that have identical functional behaviour but no side effects; the function β_c^σ extracts the result of the computation and maps it to the element that has the same functional behaviour. One can show that $(\beta_c^\sigma \circ \alpha_c^\sigma)$ is the identity function, *i.e.*, D^σ is a **retract** of $\mathcal{C}_T[\![\sigma]\!]$. We thus define the retraction function $\delta_c^\sigma = (\alpha_c^\sigma \circ \beta_c^\sigma)$; the retractions δ_c^σ coerce a computation into one with identical functional behaviour but no side effects. Add the term constructors $\langle \mathsf{encap}_\sigma; \sigma \Rightarrow \sigma \rangle$ for each VPCF type σ to VPCF; $\mathcal{T}$ is a model of this extended language with $\mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]\rho_\mathcal{T} = \delta_c^\sigma(\mathcal{T}[\![M]\!]\rho_\mathcal{T})$. The following theorem then captures the behaviour of the retractions as semantic coercions.

Theorem 4 *Suppose M, N are closed VPCF terms and $\mathcal{T}$ is a call-by-value monad over $\mathcal{D}$. Then*

1. *Adequacy:* $\mathcal{D}[\![M]\!] = \beta(\mathcal{T}[\![\mathsf{encap}_{\mathsf{num}}\ M]\!])$.
2. *Preservation: If $C[\cdot]$ is a closed VPCF context of type* num, *then* $\mathcal{T}[\![C[M]]\!] = \mathcal{T}[\![C[\mathsf{encap}_\sigma\ M]]\!]$.
3. *Full abstraction:* $\mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!] = \mathcal{T}[\![\mathsf{encap}_\sigma\ N]\!]$ *if and only if* $\mathcal{D}[\![M]\!] = \mathcal{D}[\![N]\!]$.

We use logical relations to prove Theorem 4. First, define the following family of binary relations that are used to relate elements of D^σ with computations in $\mathcal{C}_T[\![\sigma]\!]$ whose results have the same functional behaviour.

Definition 5 Define the relations $R_r^\sigma : D^\sigma \times \mathcal{R}_T[\![\sigma]\!]$ and $R_c^\sigma : D^\sigma \times \mathcal{C}_T[\![\sigma]\!]$ by

- $d\ R_r^{\mathrm{num}}\ e$ iff $d = e$.
- $d\ R_r^{\sigma\to\tau}\ e$ iff ($d\!\uparrow$ and $e\!\uparrow$), or
  ($d\!\downarrow$ and $e\!\downarrow$) and
  $\forall d'\!\downarrow, e'\!\downarrow .\ d'\ R_r^\sigma\ e' \supset (d\ d')\ R_c^\tau\ (e\ e')$
- $d\ R_c^\sigma\ e$ iff there is an e' such that $e = (\eta\ e')$ and $d\ R_r^\sigma\ e'$.

Table 3: Semantics of VPCF Over a Call-by-value Monad.

$$
\begin{array}{rcl}
\mathcal{T}[\![x^\sigma]\!]\rho_\mathcal{T} & = & \eta\,(\rho_\mathcal{T}(x^\sigma)) \\
\mathcal{T}[\![(\lambda x^\sigma.\, M)]\!]\rho_\mathcal{T} & = & \eta\,(\underline{\lambda} d \in D^{\sigma'}.\, \mathcal{T}[\![M]\!]\rho_\mathcal{T}[x^\sigma \mapsto d]) \\
\mathcal{T}[\![(M\ N)]\!]\rho_\mathcal{T} & = & \mathsf{Ap}\ \mathcal{T}[\![M]\!]\rho_\mathcal{T}\ \mathcal{T}[\![N]\!]\rho_\mathcal{T} \\
\mathcal{T}[\![n]\!]\rho_\mathcal{T} & = & \eta\,(\mathcal{D}[\![n]\!]) \\
\mathcal{T}[\![\mathsf{succ}\,(M)]\!]\rho_\mathcal{T} & = & T\,(\underline{\lambda} x \in D^{\mathrm{num}}.\, \mathsf{succ}\,(x))\ \mathcal{T}[\![M]\!]\rho_\mathcal{T} \\
\mathcal{T}[\![\mathsf{pred}\,(M)]\!]\rho_\mathcal{T} & = & T\,(\underline{\lambda} x \in D^{\mathrm{num}}.\, \mathsf{pred}\,(x))\ \mathcal{T}[\![M]\!]\rho_\mathcal{T} \\
\mathcal{T}[\![\mathsf{if0}(M,N,P)]\!]\rho_\mathcal{T} & = & \mu[T\,(\underline{\lambda} b \in D^{\mathrm{num}}.\, \mathsf{if0}(b, \mathcal{T}[\![N]\!]\rho_\mathcal{T}, \mathcal{T}[\![P]\!]\rho_\mathcal{T}))\ \mathcal{T}[\![M]\!]\rho_\mathcal{T}] \\
\mathcal{T}[\![\mathsf{Y}^\sigma]\!]\rho_\mathcal{T} & = & \eta\,(\bigsqcup_n F^n_\sigma),\ \text{where} \\
 & & F^0_\sigma \;=\; \bot^{((\sigma\to\sigma)\to\sigma)'} \\
 & & F^{n+1}_\sigma \;=\; \underline{\lambda} f \in D^{(\sigma\to\sigma)'}.\, f\,(\underline{\lambda} x \in D^{\tau'}.\, \mathsf{Ap}\,(F^n_\sigma\ f)\,(\eta_{\tau'}\ x))
\end{array}
$$

Table 4: Embedding $\mathcal{D}$ into $\mathcal{T}$.

$$
\begin{array}{rcl@{\qquad}rcl}
\alpha^{\mathrm{num}}_r & = & \underline{\lambda} x \in D^{\mathrm{num}}.\, x & \alpha^{\sigma\to\tau}_r & = & \underline{\lambda} x \in D^{\sigma\to\tau}.\, \underline{\lambda} y \in D^{\sigma'}.\, \alpha^\tau_c\,(x\,(\beta^\sigma_r\ y)) \\
\beta^{\mathrm{num}}_r & = & \underline{\lambda} x \in D^{\mathrm{num}}.\, x & \beta^{\sigma\to\tau}_r & = & \underline{\lambda} x \in D^{(\sigma\to\tau)'}.\, \underline{\lambda} y \in D^{\sigma}.\, \beta^\tau_c\,(x\,(\alpha^\sigma_r\ y)) \\
\alpha^\sigma_c(x) & = & \eta\,(\alpha^\sigma_r\ x) & & & \\
\beta^{\mathrm{num}}_c & = & \beta & \beta^{\sigma\to\tau}_c(x) & = & \begin{cases} \bot^{\sigma\to\tau} & \text{if } \beta\,(\mathsf{Ap}\,(\eta\,(\underline{\lambda} z \in D^{(\sigma\to\tau)'}.\, \eta\ 0))\ x)\uparrow \\ \beta^{\sigma\to\tau}_r\,(\underline{\lambda} y \in D^{\sigma'}.\, \mathsf{Ap}\ x\ (\eta\ y)) & \text{otherwise} \end{cases}
\end{array}
$$

The relations R^σ_c can be shown to be directed complete, which together with the fact that the relations R^σ_c are "logical" helps establish the following lemma, that relates the meaning of a term in the two models.

Lemma 6 *Suppose that $\rho_\mathcal{D}$ is a $\mathcal{D}$-environment and $\rho_\mathcal{T}$ is a $\mathcal{T}$-environment such that $\rho_\mathcal{D}(x^\sigma)\ R^\sigma_r\ \rho_\mathcal{T}(x^\sigma)$ for all x^σ. Then for any PCF-term $M : \sigma$, $\mathcal{D}[\![M]\!]\rho_\mathcal{D}\ R^\sigma_c\ \mathcal{T}[\![M]\!]\rho_\mathcal{T}$.*

To prove Theorem 4, we need a few more simple facts:

Lemma 7

1. *For any $x \in D^\sigma$, $\beta^\sigma_c\,(\alpha^\sigma_c\ x) = x$.*
2. *If $d\ R^\sigma_c\ e$ then $d\ R^\sigma_c\ (\delta^\sigma_c\ e)$.*
3. *If $d\ R^\sigma_c\ (\delta^\sigma_c\ e)$ then $\alpha^\sigma_c\ d = \delta^\sigma_c\ e$.*

The proof, which we omit, uses a simple induction on the structure of types. Part 1 of Lemma 7 states the previously mentioned property that $\beta_c \circ \alpha_c$ is the identity function. An important consequence of this property is that the functions α^σ_c are injective. It therefore follows from the last two statements in Lemma 7 that the inverse of the logical relations are injective on the range of the coercions δ^σ_c.

Proof of Theorem 4:

1. By Lemma 6, $\mathcal{D}[\![M]\!]\ R^{\mathrm{num}}_c\ \mathcal{T}[\![M]\!]$. Hence, by the second part of Lemma 7,

   $$\mathcal{D}[\![M]\!]\ R^{\mathrm{num}}_c\ \mathcal{T}[\![\mathsf{encap}_{\mathrm{num}}\ M]\!].$$

   Using the definition of R^{num}_c, this means that

   $$\mathcal{T}[\![\mathsf{encap}_{\mathrm{num}}\ M]\!] = \eta_{\mathrm{num}}(\mathcal{D}[\![M]\!])$$

   By Property 5 in the definition of a call-by-value monad, the adequacy statement now follows.

2. By Lemma 6,

   $$\mathcal{D}[\![M]\!]\ R^\sigma_c\ \mathcal{T}[\![M]\!] \tag{1}$$

   Using the definition of R^σ_c, this means that

   $$\mathcal{T}[\![M]\!] = (\eta\ e) \text{ and } \mathcal{D}[\![M]\!]\ R^\sigma_r\ e.$$

   Thus, if x^σ is a variable that does not appear in the context $C[\cdot]$, then the environments $\rho_\mathcal{D}$ and $\rho_\mathcal{T}$ such that $\rho_\mathcal{D}(x^\sigma) = \mathcal{D}[\![M]\!]$ and $\rho_\mathcal{T}(x^\sigma) = e$ satisfy the conditions of Lemma 6. It therefore follows from Lemma 6 that $\mathcal{D}[\![C[x^\sigma]]\!]\rho_\mathcal{D}\ R^{\mathrm{num}}_c\ \mathcal{T}[\![C[x^\sigma]]\!]\rho_\mathcal{T}$ and so

   $$\mathcal{D}[\![C[M]]\!]\ R^{\mathrm{num}}_c\ \mathcal{T}[\![C[M]]\!].$$

   Now, using the definition of R^{num}_c,

   $$\mathcal{T}[\![C[M]]\!] = \eta(\mathcal{D}[\![C[M]]\!]) \tag{2}$$

   Using the second part of Lemma 7 on (1),

   $$\mathcal{D}[\![M]\!]\ R^\sigma_c\ \mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]$$

   Then by a similar argument as before,

   $$\mathcal{T}[\![C[\mathsf{encap}_\sigma\ M]]\!] = \eta(\mathcal{D}[\![C[M]]\!]) \tag{3}$$

   Hence we get the required statement from (2) and (3).

3. By Lemma 6 and the second part of Lemma 7, we have

   $$\mathcal{D}[\![M]\!]\ R^\sigma_c\ \mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]$$
   $$\mathcal{D}[\![N]\!]\ R^\sigma_c\ \mathcal{T}[\![\mathsf{encap}_\sigma\ N]\!]$$

   Using the third part of Lemma 7, we have

   $$\alpha^\sigma_c\ (\mathcal{D}[\![M]\!]) = \mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]$$
   $$\alpha^\sigma_c\ (\mathcal{D}[\![N]\!]) = \mathcal{T}[\![\mathsf{encap}_\sigma\ N]\!]$$

   It follows that

   $$\begin{array}{lll} \mathcal{D}[\![M]\!] = \mathcal{D}[\![N]\!] & \text{iff} & \alpha^\sigma_c\ (\mathcal{D}[\![M]\!]) = \alpha^\sigma_c\ (\mathcal{D}[\![N]\!]) \\ & \text{iff} & \mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!] = \mathcal{T}[\![\mathsf{encap}_\sigma\ N]\!] \end{array}$$

   where the first equivalence follows from the injectivity of α^σ_c.

This completes the proof of the theorem. ■

5 Syntactic Coercion Operators

We now use the framework of call-by-value monads to interpret extensions of VPCF with side effects. We first show how to add coercion operation encap to extensions of VPCF, with the operational rules for the coercion operation determined by the monad, and state a correctness theorem for the resulting coercions given some properties of the monadic semantics of the extension. We then apply the theorem to our two example extensions, VPCF+K and VPCF+S.

5.1 General extensions

Let L be an extension of VPCF and $\langle T, \eta, \mu, \beta\rangle$ be a call-by-value monad over a model $\mathcal{D}$ of VPCF. As can be seen from the examples in Section 3, a general way to specify the syntax of terms in L is to possibly add type and term constructors to VPCF. A denotational semantics $\mathcal{T}[\![\cdot]\!]$, over the monad $\mathcal{T}$, for L is given by extending the definition of σ' to the additional types σ of L and specifying functions $\mathcal{T}[\![f]\!] : \mathcal{C}_{\mathcal{T}}[\![\sigma_1]\!] \times \ldots \times \mathcal{C}_{\mathcal{T}}[\![\sigma_n]\!] \to \mathcal{C}_{\mathcal{T}}[\![\sigma]\!]$ for each new term constructor $\langle f; \sigma_1, \ldots, \sigma_n \Rightarrow \sigma\rangle$ in L, such that $\mathcal{T}[\![f]\!]$ is interpretable in $\mathcal{D}$. Together with Table 3, the clause $\mathcal{T}[\![f(M_1, \ldots, M_n)]\!]\rho_{\mathcal{T}} = \mathcal{T}[\![f]\!](\mathcal{T}[\![M_1]\!]\rho_{\mathcal{T}}, \ldots, \mathcal{T}[\![M_n]\!]\rho_{\mathcal{T}})$ defines the meaning of all terms of L. We add term constructors $\langle \mathsf{encap}_\sigma; \sigma \Rightarrow \sigma\rangle$ for each VPCF type σ to L; call the resulting language $L + \mathsf{encap}$. The meaning function $\mathcal{T}[\![\cdot]\!]$ can be extended to $L + \mathsf{encap}$ with $\mathcal{T}[\![\mathsf{encap}_\sigma\ M]\!]\rho_{\mathcal{T}} = \delta_c^\sigma(\mathcal{T}[\![M]\!]\rho_{\mathcal{T}})$. The following lemma, proved by expanding out the definitions of the retractions δ_c, gives us operational rules for the encap operators.

Lemma 8 *Let L be an extension of call-by-value PCF whose model $\mathcal{T}[\![\cdot]\!]$ is defined over a call-by-value monad. Then, for any $L + \mathsf{encap}$ term M and $\mathcal{T}$-environment ρ,*

- $\mathcal{T}[\![\mathsf{encap}_{\mathrm{num}}\ M]\!]\rho = \eta\,(\beta\,(\mathcal{T}[\![M]\!]\rho))$.
- *If y^σ is not free in M, then $\mathcal{T}[\![\mathsf{encap}_{\sigma\to\tau}\ M]\!]\rho$ equals $\mathcal{T}[\![\mathsf{encap}_{\mathrm{num}}(M\,;0)\,;\lambda y^\sigma.\ \mathsf{encap}_\tau(M(\mathsf{encap}_\sigma y))]\!]\rho$.*

The rewriting operational semantics framework makes it particularly easy to describe the operational rules for encap. The operational semantics of L+encap is obtained by adding the redexes in Table 5 to those of VPCF. Note that the second redex is independent of the language extension or the particular monad. Call the denotational semantics $\mathcal{T}[\![\cdot]\!]$ of L, over the monad T, **adequate**, if for all programs, *i.e.*, closed terms M of type num, $Eval_L(M) = n$ iff $\beta(\mathcal{T}[\![M]\!]) = \mathcal{T}[\![n]\!]$, and similarly for $L + \mathsf{encap}$. Using Lemma 8, we can prove the following lemma which shows that the operational rules for encap are faithful to its denotational semantics.

Lemma 9 *Let $\langle T, \eta, \mu, \beta\rangle$ be a call-by-value monad over a call-by-value model $\mathcal{D}$ and $\mathcal{T}[\![\cdot]\!]$ be some denotational semantics over the monad T that is adequate for L. Then $\mathcal{T}[\![\cdot]\!]$ is also adequate for $L + \mathsf{encap}$.*

Using Lemma 9 and Theorem 4, we can now prove that under certain conditions encap accurately recovers the operational behaviour of VPCF:

Theorem 10 *Let $\mathcal{D}$ be a fully abstract, least fixpoint model for VPCF. Let $\langle T, \eta, \mu, \beta\rangle$ be a call-by-value monad over $\mathcal{D}$ and $\mathcal{T}[\![\cdot]\!]$ be some denotational semantics over the monad T that is adequate for L. Then, for any closed VPCF terms M, N of the same type,*

1. *Adequacy: $Eval_{L+\mathsf{encap}}(\mathsf{encap}_{\mathrm{num}} M) = n$ if and only if $Eval_v(M) = n$.*
2. *Preservation: Let $C[\cdot]$ be any closed VPCF context of type num. Then $Eval_{L+\mathsf{encap}}(C[M]) = n$ if and only if $Eval_{L+\mathsf{encap}}(C[\mathsf{encap}_\sigma\ M]) = n$.*
3. *Full abstraction: $(\mathsf{encap}_\sigma\ M) \equiv_{L+\mathsf{encap}} (\mathsf{encap}_\sigma\ N)$ if and only if $M \equiv_{\mathrm{VPCF}} N$.*

While the above theorem describes the behaviour of encap on purely functional terms, one can also use the semantic theorems to describe the behaviour of encap on arbitrary terms. For instance, one can prove that encap produces observationally congruent terms from "functionally" equivalent terms. Define a closed term M of arbitrary type σ to **halt** iff $Eval(M\,;0)$ is defined. Then we can define two closed terms M and N of type σ to be functionally equivalent, $M \equiv^\sigma_{pure} N$, as follows:

- $M \equiv^{\mathrm{num}}_{pure} N$ iff $Eval(M) \simeq Eval(N)$
- $M \equiv^{\tau\to\nu}_{pure} N$ if M halts iff N halts and for all VPCF values V of type τ, $M\ V \equiv^\nu_{pure} N\ V$.

Inuitively, M and N are functionally equivalent if they have the same input-output behaviour. Then, the following theorem shows that functionally equivalent pieces of code are interchangeable within the scope of encap.

Theorem 11 *Let $\mathcal{D}$ be a fully abstract, least fixpoint model for VPCF. Let $\langle T, \eta, \mu, \beta\rangle$ be a call-by-value monad over $\mathcal{D}$ and $\mathcal{T}[\![\cdot]\!]$ be some denotational semantics over the monad T that is adequate for L. Suppose M, N are closed $L + \mathsf{encap}$ terms of PCF type such that $M \equiv_{pure} N$. Then*

$$(\mathsf{encap}\ M) \equiv_{L+\mathsf{encap}} (\mathsf{encap}\ N).$$

Since the logical relations establish a bijective correspondence between the call-by-value model of VPCF and the range of the retractions, we conjecture that the theory of the call-by-value model (when the logic is suitably defined) will be the same as for coerced expressions.

5.2 Adding encap to VPCF+K

Let $\mathcal{V}$ be any least fixpoint model of VPCF. We can define the continuation monad $\mathcal{K} = \langle K, \eta^{\mathcal{K}}, \mu^{\mathcal{K}}, \beta^{\mathcal{K}}\rangle$ over $\mathcal{V}$ by

$$\begin{aligned}
K(\sigma) &= (\sigma \to \mathsf{num}) \to \mathsf{num}\\
K_{\tau,\nu} &= \underline{\lambda} f \in V^{\tau\to\nu}.\ \underline{\lambda} x \in V^{K(\tau)}.\\
&\qquad \underline{\lambda}\kappa \in V^{\nu\to\mathrm{num}}.\ x\ (\underline{\lambda} v \in V^{\tau}.\ \kappa\ (f\ v))\\
\eta^{\mathcal{K}}_\sigma\ (x) &= \underline{\lambda}\kappa \in V^{\sigma\to\mathrm{num}}.\ \kappa\ x\\
\mu^{\mathcal{K}}_\sigma &= \underline{\lambda} x \in V^{K(K(\sigma))}.\\
&\qquad \underline{\lambda}\kappa \in V^{\sigma\to\mathrm{num}}.\ x\ (\underline{\lambda} m \in V^{K(\sigma)}.\ m\ \kappa)\\
\beta^{\mathcal{K}} &= \underline{\lambda} x \in V^{K(\mathrm{num})}.\ x\ (\underline{\lambda} y \in V^{\mathrm{num}}.\ y)
\end{aligned}$$

One may verify that this definition gives a call-by-value monad. We can obtain a semantics $\mathcal{K}$ for VPCF+K by

$$\begin{aligned}
\mathcal{K}[\![\mathsf{callcc}]\!](f) &= \underline{\lambda}\kappa.\ f\ (\underline{\lambda} m.\ m\ \kappa\ \kappa)\\
\mathcal{K}[\![\mathsf{abort}]\!](f) &= \underline{\lambda}\kappa \in V^{\mathrm{num}\to\mathrm{num}}.\ f\ (\underline{\lambda} x \in V^{\mathrm{num}}.\ x)
\end{aligned}$$

Since $\beta^{\mathcal{K}}$ corresponds to evaluating in the empty context, we thus get the redex defined in Table 5 for VPCF+K. Note

Table 5: Operational Rules for encap.

General call-by-value monad	$(\mathsf{encap}_{\mathrm{num}}\ M) \rightsquigarrow n$, if $\beta(\mathcal{T}[\![M]\!]) = \mathcal{D}[\![n]\!]$ $(\mathsf{encap}_{\sigma\to\tau}\ M) \rightsquigarrow \mathsf{encap}_{\mathrm{num}}(M;0)\ ;\ \lambda y^{\sigma}.\ \mathsf{encap}_{\tau}\ (M\ (\mathsf{encap}_{\sigma}\ y))$
VPCF+K	$(\mathsf{encap}_{\mathrm{num}}\ M) \rightsquigarrow n$, if $M \twoheadrightarrow_c n$
VPCF+S	$(\mathsf{encap}_{\mathrm{num}}\ M) \rightsquigarrow n$, if $(M, \emptyset) \twoheadrightarrow_s (n, s')$

that the semantics of $\mathsf{encap}_{\mathrm{num}}$ coincides with that of first-class prompts [8]. The semantics $\mathcal{K}[\![\cdot]\!]$ corresponds directly to usual cps (continuation-passing style) transforms [30, 39]. It is thus easy to check that if $\mathcal{V}$ is an adequate model for VPCF, then the semantics $\mathcal{K}[\![\cdot]\!]$ is adequate for VPCF+K. Since any fully abstract model is also adequate, we can apply Theorem 10 to the case of VPCF+K, if the model $\mathcal{V}$ is fully abstract. Thus, to establish the correctness of encap for VPCF+K, all that remains is the construction of such a model, *i.e.*, prove the following theorem:

Theorem 12 *There exists a fully abstract, least fixpoint model $\mathcal{V}$ of VPCF.*

One obvious way to construct a fully abstract model of $\mathcal{V}$ is to take the term model consisting of equivalence classes of all closed terms. Unfortunately the term model for VPCF, while trivially fully abstract, is *not continuous*. However, by a suitable "context lemma" (*cf.* [21]), the term model for a purely functional call-by-value language is always monotone. Thus, if all types in the term model are finite, then it is also *continuous*. We rely upon this insight to prove Theorem 12. Instead of VPCF, we consider a language FVPCF in which all types are finite and which will serve as an approximation to VPCF. FVPCF has base types num_i, whose elements are $\{0, \ldots, i\}$, that serve as finite approximations of the infinite type num. FVPCF does not have fixed point operators, but includes suitable "finite" approximations of the other term constructors of VPCF modified to operate on the num_i's. More importantly, FVPCF needs to include constants representing divergence explicitly in its syntax to reflect the call-by-value operational equivalence. The continuous term model of FVPCF contains a lattice of "partial types" related by embedding-projection maps approximating each type of VPCF. We obtain a least fixed point model $\mathcal{V}$ for VPCF by taking the meaning of each type to be the (inverse) limit of its "partial type" approximations. In fact, each type is an algebraic cpo with its finite elements corresponding to the definable terms of FVPCF; definability of the finite elements is enough to prove full abstraction. The precise details of the construction appear in [38].

5.3 Adding encap to VPCF+S

To give a denotational semantics for VPCF+S, we need to model states. Since states may be circular, the appropriate representation of states has to be recursive. A suitable framework for building states is the language FPC defined in [9], which includes a one-element type unit, product types, sum types, and recursive types. We can thus define a store call-by-value monad over $\mathcal{F}$ as follows. Using the recursive types of FPC, we first define types $\mathsf{Loc} = (\mu t.\ \mathsf{unit} + t)$ and Store to be the appropriate recursive type given by the equations

$$\begin{aligned} \mathsf{Store} &= \mathsf{Loc} \to (\mathsf{Res} + \mathsf{unit}) \\ \mathsf{Res} &= \mathsf{num} + \mathsf{Loc} + (\mathsf{Res} \to \mathsf{Comp}) \\ \mathsf{Comp} &= \mathsf{Store} \to (\mathsf{Res} \times \mathsf{Store}) \end{aligned}$$

(*cf.* [36]). Let init be the initial store mapping every location to the right portion of the sum, *i.e.*, declaring each cell to be "unused". Let the call-by-value monad $\mathcal{S} = \langle S, \eta^{\mathcal{S}}, \mu^{\mathcal{S}}, \beta^{\mathcal{S}} \rangle$ be defined by

$$\begin{aligned} S(\sigma) &= \mathsf{Store} \to (\sigma \times \mathsf{Store}) \\ S_{\tau,\nu} &= \underline{\lambda} f \in F^{\tau\to\nu}.\ \underline{\lambda} x \in F^{S(\tau)}. \\ &\qquad \underline{\lambda} s.\ \mathsf{pair}(f(\mathsf{fst}(x\ s)), \mathsf{snd}(x\ s)) \\ \eta^{\mathcal{S}}_{\sigma}(x) &= \underline{\lambda} s.\ \mathsf{pair}(x, s) \\ \mu^{\mathcal{S}}_{\sigma} &= \underline{\lambda} x \in F^{S(S(\sigma))}.\ \underline{\lambda} s.\ \mathsf{fst}(x\ s)\ \mathsf{snd}(x\ s) \\ \beta^{\mathcal{S}} &= \underline{\lambda} x \in F^{S(\mathrm{num})}.\ \mathsf{fst}(x\ \mathsf{init}) \end{aligned}$$

The semantics $\mathcal{S}$ of types is determined by setting $(\mathsf{ref}\ \sigma)' = \mathsf{Loc}$; the semantics of the additional term constructors in VPCF+S can be written as meanings of the appropriate terms of FPC. Since $\beta^{\mathcal{S}}$ corresponds operationally to evaluating in the initial store, the operational rule in Table 5 is an implementation of $\beta^{\mathcal{S}}$. This breaks the usual "single-threaded" property of state; see below for more discussion.

One can show that if $\mathcal{F}$ is an adequate model for FPC then the resulting semantics $\mathcal{S}[\![\cdot]\!]$ of VPCF+S is also adequate. Moreover, if the model $\mathcal{F}$ of FPC is fully abstract, then the submodel of $\mathcal{F}$ for VPCF obtained by taking $\mathsf{num} = (\mu t.\ \mathsf{unit} + t)$ is also fully abstract. Thus, by Theorem 10, we can obtain the correctness result for VPCF+S+encap if we can prove

Theorem 13 *There exists a continuous, fully abstract model $\mathcal{F}$ of FPC.*

We follow the construction for VPCF; the details also appear in [38]. We define a finite language FFPC that includes an "empty" base type void, the one-element type unit, products and sums, but not recursive types. The main difficulty in constructing the model of FPC from the term model of FFPC is in defining the types of FFPC that serve as approximations to the recursive types of FPC and suitable embedding-projection maps between them. Intuitively, we think of a recursive type $\mu t.\ \sigma$ as the limit of its finite unwindings $\mu^n t.\ \sigma$, where $\mu^0 t.\ \sigma = \mathsf{void}$, $\mu^{i+1} t.\ \sigma = [\mu^i t.\ \sigma/t]\sigma$. The partial types approximating a recursive type $\mu t.\ \sigma$ are those approximating each unwinding $\mu^n t.\ \sigma$.

6 Example of encap with State

Let us consider an extended example of how to use encap for reasoning in a language with state. We use the syntax of Standard ML [22] instead of VPCF in this example, partly because we will need a richer set of data types, and partly to improve readability.

The example is unification (also considered as a test case in [40] for the Imperative Lambda Calculus). Ordinary (first-order) unification admits two elegant solutions: the first, Robinson's algorithm using substitutions, can be implemented in a purely functional manner; the second, using directed acyclic graphs (dags), is much more efficient but requires state. The following is an encoding of the latter algorithm; we leave it to the reader to find the purely functional encoding.

```
datatype term = Num | Typevar of string
                    | Fnsp of term*term
local
  datatype tag = Unbound | Instance of dagterm
  and dagterm  = DNum | DTypevar of string*(ref tag)
                      | DFnsp of dagterm*dagterm
  fun find (x,[]) = None
    | find (x,(y,z)::tl) =
      if x=y then Some z else find (x,tl)
  fun convertToDags (t1,t2) =
    let val table = ref []
        fun convert Num = DNum
          | convert (Typevar x) =
            (case find (x,!table)
               of Some r => DTypevar (x,r)
                | _ => let val p = (x,ref Unbound)
                       in  table := p::(!table);
                           DTypevar p
                       end)
          | convert (Fnsp(t,u)) =
            DFnsp(convert t,convert u)
    in (convert t1, convert t2)
    end
  fun occur x DNum = false
    | occur x (DTypevar (y,r)) =
      (case !r
         of Unbound => (x=y)
          | Instance t => occur x t)
    | occur x (DFnsp (t1,t2)) =
      (occur x t1) orelse (occur x t2)
  fun unifyDags (DNum, DNum) = true
    | unifyDags (DTypevar v1, DTypevar v2) =
      if (v1=v2) then true else unifyVs(v1, v2)
    | unifyDags (DTypevar v1, t2) = unifyVT (v1, t2)
    | unifyDags (t1, DTypevar v2) = unifyVT (v2, t1)
    | unifyDags (DFnsp(t1, t2),DFnsp(t3, t4)) =
      (unifyDags (t1, t3)) andalso
      (unifyDags (t2, t4))
    | unifyDags _ = false
  and unifyVs (v1 as (x,r), v2 as (x',r')) =
      (case (!r, !r')
         of (Instance ty1, _) => unifyVT(v2, ty1)
          | (_, Instance ty2) => unifyVT(v1, ty2)
          | (Unbound,Unbound) =>
            (r := (Instance (DTypevar v2));true))
  and unifyVT (v1 as (x,r), t) =
      (case !r of Unbound =>
                  if occur x t
                     then false
                     else (r := Instance t; true)
                | (Instance t') => unifyDags(t,t'))
in fun unify p = unifyDags (convertToDags p)
end
```

Terms are encoded by the datatype `term` and the function `unify` returns `true` if the terms are unifiable and `false` otherwise. The datatype `dagterm` changes only the representation of variables—variables can now point to other terms as an "instance", which serves to make unification of terms with variables a simple matter. Two auxiliary functions in this code use assignments. The function `convertToDags`, which converts elements of the datatype `term` into dags, uses a reference cell to store type variables that have already been assigned references; the function `unifyVT` also uses assignment to link a variable to a new instance.

We can use the two encodings of `unify` to illustrate a general programming methodology of modular refinement where a programmer first implements a module purely functionally and then refines it using state to produce a more efficient version. In this context, one may hope to use the imperative version of `unify` in place of the functional version in any program since they have the same input-output behaviour. However, as the example in the introduction showed, in the presence of side-effects, such a refinement may not always preserve the behaviour of the whole program. If the language includes encap, we can guarantee the correctness of this process of modular refinement by wrapping the code of `unify` with encap. By Theorem 11, having the same input-output behaviour is then enough to guarantee the correctness of this refinement, since the refinement is within the scope of encap.

If we do not wish to include encap in the language, we can still use encap as a useful reasoning tool to establish the correctness of replacing the functional version with the imperative version directly (*i.e.*, without wrapping it with encap). First, observe that the code accesses no reference cell that is not allocated during a call to `unify`, and that none of the reference cells can be accessed after a call to `unify`. Also observe that the declaration yields a value, and that the arguments to `unify` cannot themselves contain any reference cells (due to the limited structure of the datatype `term`). From these facts and the denotational or operational semantics of encap, one may prove that wrapping encap around the entire expression yields a term that is operationally equivalent to the expression itself. Similarly, since the functional version of `unify` also cannot access any reference cells from outside, one may show that wrapping encap around it gives an operationally equivalent expression. Since Theorem 11 gives us the operational equivalence of the two pieces of code wrapped with encap, it follows that the two pieces of code are equivalent even without encap around them.

This example and the example in [36] illustrate a common technique for proofs of correctness. If one can show that state is already encapsulated (by showing that applying encap results in an operationally equivalent term), one can then apply Theorem 11 to deduce that functional reasoning principles are sufficient. This is an important reasoning principle and well-supported by programmer's intuition.

7 Related Work

This approach in this paper is certainly not the first attempt to control side effects in functional languages; there is a long tradition of language designs that attempt to control side effects. Such efforts have mainly focused on simplifying the reasoning principles required for proving properties of programs and on automatically determining opportunities

for parallel evaluation. Broadly speaking, two distinct and seemingly unrelated camps have emerged, divided along the traditional call-by-name versus call-by-value line.

Most of the work has followed the call-by-name tradition, beginning with Algol 60. Reynolds' more contemporary work on syntactic control of interference [32, 34], Idealized Algol [33], and Forsythe [34] have carried forth this tradition. Each of these languages incorporates restrictions on both types and terms so as to force assignments and references to be well-behaved. For example, in Idealized Algol, the basis for the other two programming language designs, terms that write to the state are grouped into the type of *commands*; the other two types of basic terms, locations and expressions, may read the state but not modify the state. This greatly simplifies reasoning about programs: with the call-by-name parameter passing, programs of command type may be unwound into a simple sequence of imperative statements—in other words, user-defined functions can be completely removed from the program by partial evaluation [34]. The imperative lambda calculus (ILC) of [40] has a similar design philosophy in separating the terms that can modify the state from those that may not. The lazy functional programming community has also been active in searching for new, clean ways of adding state to programming languages; just a few of these efforts are dialogue-based I/O in Haskell [13], monads [29, 43], continuation-based mutable abstract datatypes [12], and lazy functional state threads [18] (whose type-theoretic encapsulation of state is remarkably similar to the work in semantics of [27]). In each case, the design preserves some functional character of the language—the equation (β) is sound for proving operational equivalences, for example, so terms of functional type do not yield side effects when applied.

Another language in the call-by-name tradition, and possibly the closest design to VPCF+S+encap, is the language λ_{var} of [25]. Syntactically, λ_{var} is an extension of the untyped λ-calculus with assignments and some additional operations for encapsulating state. λ_{var} enjoys a property similar to our main theorem—two pieces of code M, N without assignments are operationally equivalent in the full language iff they M, N are operationally equivalent in the language without assignments; in contrast to our main theorem, no operations are required to coerce M and N. λ_{var} enforces this purity of assignments via two new term constructors, pure and return. Using the syntax of this paper, the operational semantics for the special operations of λ_{var} are

$$(\mathsf{pure}\ (\mathsf{return}\ n)) \rightsquigarrow n$$
$$(\mathsf{pure}\ (\mathsf{return}\ (\lambda x.\ M))) \rightsquigarrow \lambda x.\ \mathsf{pure}\ (\mathsf{return}\ M)$$

As with the semantics of encap, pure applied to functional abstractions forces the value returned by the function to be pure; the argument, on the other hand, is not forced to be pure (probably because of the call-by-name reduction semantics). Unlike encap, the operation return is needed in the language to mark which subexpressions may be marked as side-effect free. Those expressions that are not returned, *e.g.*, (pure 0), are "stuck" expressions that cannot be reduced further; the counterpart in VPCF+S+encap is an expression like (encap l) for some location l. The advantage of λ_{var} over VPCF+S+encap is that it can be implemented with a single-threaded store; the disadvantage is that it is manifestly based on call-by-name semantics (as with ILC and Idealized Algol), and that the syntax of assignments, pure, and return are somewhat restrictive.

Far less attention has been paid to controlling side effects in the call-by-value realm. Some of the most well-developed approaches are the type and effect systems for assignments [17, 19, 20, 41, 42], continuations [16], and exceptions [10]. In each of these languages, a static system assigns both types and possible effects to expressions; the type and effect can then be used in the generation of code for parallel architectures and other applications. For example, in the type and effect system for state, one envisages memory as divided into a set of *regions*, or blocks of locations. The type and effect system calculates a conservative approximation of dividing the cells into regions. Consider a simple example, the "gensym" function that returns a new integer each time it is called:

$$(\lambda x^{\mathsf{ref\ num}}.\ \lambda d^{\mathsf{num}}.\ (x := \mathsf{succ}\ (!x)); (!x))\ (\mathsf{ref}\ 0)$$

In the type and effect system, the type is more complicated than (num $\rightarrow$ num) (the type in VPCF+S). This expression creates a new cell, so the effect is init(ρ) for some region ρ, and when given an argument of type num the expression returns a num and reads in writes in region ρ, so the type is

$$(\mathsf{num} \overset{\mathsf{read}(\rho),\mathsf{write}(\rho)}{\rightarrow} \mathsf{num}).$$

The effects above the $\rightarrow$ is called a *latent* effect, since they are effects poised to fire during a function call but do not occur during the evaluation of the expression.

Automatic parallelization of code with side effects is one of the primary uses of effect systems (see, *e.g.*, [14]). In the case of state, for example, if the manifest effects of two subterms of a term occur in different regions, the two subterms may be safely reduced in parallel. The operations encap may also be used for this purpose, albeit at a much less refined level: for example, the subterms (encap M) and (encap N) may be reduced in parallel in evaluating ((encap M) (encap N)); the correctness of this parallelization follows from Theorem 10 and a suitable standardization theorem for VPCF (*cf.* [30]).

The type and effect system has at least two advantages over the encap operation in VPCF+S+encap. First, using ideas from type inference, the compiler can determine the type and effect of an expression automatically [42]; the programmer need not worry about the type and effects of expressions initially. Second, the language can be implemented using a single-threaded state: type and effect annotations give hints to where parallelization can happen safely without write-after-read or write-after-write conflicts, *i.e.*, where parallelization can occur without violating the sequential, single-threaded semantics; encap breaks the single-threaded semantics of state. The automatic character of type and effect systems also has drawbacks, primarily because it requires a kind of conservativity. For instance, the expression

$$(\mathsf{if0}\ 0\ \mathsf{then}\ 3\ \mathsf{else}\ x := 0; 8)$$

has a type num and effect write(ρ) if x is bound to a location in the region ρ, even though the expression has no effect. Indeed, there may be a way to meld the two approaches together by redefining the semantics of VPCF+S to use computations of the form

$$\mathsf{Comp} = (\mathsf{Region} \rightarrow \mathsf{Store}) \rightarrow (\mathsf{Res} \times (\mathsf{Region} \rightarrow \mathsf{Store}))$$

so that, in effect, the store is divided into regions. Defining β in this setting is the only difficult part, along with tying it into a general theorem like Theorem 10.

8 Conclusion

We have shown how to construct encapsulation operations that force portions of a program to behave purely functionally. The coercion operations rely on a few deep denotational theorems. The framework developed here, in contrast to the framework of [36], encompasses sequential programming languages, and hence is applicable to a wider variety of practical languages.

It is worth reexamining the stated design goals of encap from the introduction. The first design goal, to allow the programmer to protect pieces of code, requires an efficient implementation of encap. This seems to be very unclear in the case of state. Morrisett [24] has shown how one can implement first-class stores in Standard ML, with operations for naming the current store and returning back to a previously named store. It is not hard to implement encap at base type: one could name the initial store, and when entering a encap, name the current store and "snap-back" to the initial store. The higher-type encap could be implemented by evaluating the term inside the encap in the initial store, then using the closure if the evaluation converges to a λ-abstraction. In the case of polymorphic languages, some variant of the typeseq operation of [11] is essential in implementing encap. Without single-threading of state, however, the efficiency of a language with state and encap may discourage anyone from using it. There may be similar efficiency issues with encap in the language with control.

On the other hand, we feel that the second design goal—to encapsulate state for proving properties of programs—is met. We feel that the example of Section 6 is quite convincing, and it would be worth pursuing making a formal logic to reflect the reasoning of the example.

The techniques developed in this paper are useful beyond the context of the particular problem studied here. The fully abstract models of VPCF and FPC can be used in establishing other general results about sequential languages. For example, we can use them to prove that translations between programming languages are fully abstract (*cf.* [37]). More importantly, this work brought out the shortcoming of monads as a framework for expressing the semantics of side effects when applied to models arising from other monads (such as other side effects). Since the definition of a call-by-value monad formulated here had to bypass precisely this problem, it may prove to be more appropriate, for example, in analyzing languages with more than one side effecting feature.

Acknowledgements: We thank the members of the program committee for useful comments. The unification example is borrowed from an example due to John Reppy.

References

[1] S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[2] S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF: preliminary announcement. Unpublished, 1993.

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

[4] R. Cartwright and M. Felleisen. Observable sequentiality and full abstraction. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 328–342. ACM, 1992.

[5] W. Clinger and J. Rees. The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4:1–55, 1991.

[6] P.-L. Curien. Observable sequential algorithms on concrete data structures. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 432–443, 1992.

[7] M. Felleisen. λ-V-CS: An extended λ-calculus for Scheme. In *Proc. of Conf. LISP and Functional Programming*, pages 72–85. ACM, July 1988.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[9] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[10] J. C. Guzmán and A. Suárez. A type system for exceptions. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 127–135, 1994. Available as INRIA Technical Report 2265.

[11] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages.* ACM, 1995.

[12] P. Hudak. Continuation-based mutable abstract datatypes, or how to have your state and munge it too. Technical Report Research Report YALEU/DCS/RR-914, Yale University, 1992.

[13] P. Hudak, S. L. Peyton Jones, P. L. Wadler, Arvind, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the functional programming language Haskell, Version 1.2. *ACM SIGPLAN Notices*, May 1992.

[14] L. F. Huelsbergen. *Dynamic Language Parallelization.* PhD thesis, University of Wisconsin, 1993. Available as Computer Sciences Department Technical Report Number 1178.

[15] M. Hyland and L. Ong. Dialogue games and innocent strategies: An approach to intensional full abstraction to PCF (preliminary announcement). Unpublished, 1993.

[16] P. Jouvelot and D. K. Gifford. Reasoning about continuations with control effects. In *Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 1989.

[17] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 303–310. ACM, 1991.

[18] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35. ACM, 1994.

[19] J. M. Lucassen. *Type-checking Fluent Languages.* PhD thesis, Dept. Electrical Engineering & Computer Sci., Massachusetts Institute of Technology, 1987. Available as technical report MIT/LCS/TR-408 (MIT Laboratory for Computer Science).

[20] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–57. ACM, 1988.

[21] R. Milner. Fully abstract models of the typed lambda calculus. *Theoretical Computer Sci.*, 4:1–22, 1977.

[22] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[23] E. Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[24] G. Morrisett. Generalizing first-class stores. In *ACM SIGLPLAN Workshop on State in Programming Languages*, pages 73–87, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.

[25] M. Odersky, D. Rabin, and P. Hudak. Call by name, assignment, and the lambda calculus. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 43–56. ACM, 1993.

[26] P. W. O'Hearn and J. G. Riecke. Kripke relations and PCF. *Information and Computation*, 1995. To appear.

[27] P. W. O'Hearn and R. D. Tennent. Relational parametricity and local variables. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 171–184. ACM, 1993.

[28] C.-H. L. Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, Imperial College, University of London, 1988.

[29] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 71–84. ACM, 1993.

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

[31] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[32] J. C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM, 1978.

[33] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[34] J. C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Automata, Languages and Programming: 16th International Colloquium*, volume 372 of *Lect. Notes in Computer Sci.*, pages 704–722. Springer-Verlag, 1989.

[35] J. G. Riecke. *The Logic and Expressibility of Simply-Typed Call-by-Value and Lazy Languages.* PhD thesis, Massachusetts Institute of Technology, 1991. Available as technical report MIT/LCS/TR-523 (MIT Laboratory for Computer Science).

[36] J. G. Riecke. Delimiting the scope of effects. In *Proceedings of the 1993 Conference on Functional Programming and Computer Architecture*, pages 146–158. ACM, 1993.

[37] J. G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 3:387–415, 1993. Preliminary version appears in *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, ACM, 1991.

[38] J. G. Riecke and R. Viswanathan. Full abstraction for call-by-value sequential languages. Unpublished manuscript, 1993.

[39] D. Sitaram and M. Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 161–175. ACM, 1990.

[40] V. Swarup, U. Reddy, and E. Ireland. Assignments for applicative languages. In J. Hughes, editor, *Conference Proceedings of the Fifth International Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lect. Notes in Computer Sci.*, pages 192–214. Springer-Verlag, 1991.

[41] J.-P. Talpin. *Theoretical and Practical Aspects of Type and Effect Inference.* PhD thesis, University of Paris VI, 1993.

[42] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.

[43] P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.