Call-By-Push-Value

Paul Blain Levy, University of Birmingham, UK

Call-by-push-value (CBPV) is a form of typed λ-calculus that plays a fundamental role in the study of computational effects. This article is intended to be an accessible introduction. I thank Mike Mislove and SIGLOG for inviting me to expound one of my favourite subjects.

1. INTRODUCTION

1.1. Calculi for functional programming: pure and effectful

Programming language semantics is a field where we study simple programming languages or *calculi*, each of which can have various forms of mathematical description. Firstly, the *syntax* provides ways of constructing expressions—types and terms—and a *program* is a special kind of term. An *operational semantics* prescribes how to evaluate programs. By contrast, a *denotational semantics* (or model) provides every expression—type or term—with a "denotation" or meaning. It does this compositionally, by interpreting each way of constructing expressions as an operation on denotations. Lastly, an *axiomatic theory*, such as an equational theory, provides a formal way of reasoning about expressions.

Prominent among the many calculi of interest is the *simply typed λ-calculus*. It can be seen as a purely functional programming language: when a program is evaluated, all that happens is that the result is returned.

However, many programs of interest are not purely functional, as they involve *computational effects*. An effect can be defined as an "imperative feature" added to a functional program, or perhaps more accurately as an "instruction to the interpreter". For example: raising an error; printing; interactive input; making nondeterministic or probabilistic choices; running forever (via iteration or recursion); reading and writing to memory; generating new memory cells; saving and restoring the execution stack.

CBPV is a variant of the simply typed λ-calculus that incorporates computational effects, and therefore synthesizes functional and imperative programming. It was formulated and developed in my book and related papers [Levy 2004; 2001; 1999; 2006a; 2005], building on a large amount of earlier research, especially [Moggi 1991; Filinski 1996].

The fundamental CBPV principle is the distinction between values and computations, with functions classified as computations. The distinction is not as severe as it might seem, because to each computation there corresponds a value, called its *thunk*, which can be *forced* (i.e. executed) when desired.

A special feature of CBPV is that it *subsumes* both the call-by-value (CBV) and call-by-name (CBN) versions of the simply typed λ-calculus with effects. By "subsume" I mean not only that CBV and CBN can be translated into CBPV, but that these transforms *preserve every known form of semantics*. That includes operational and abstract machine semantics, and also denotational models using domain theory, possible

worlds, continuations, games, and more.[1] This phenomenon suggests that, from the semantic viewpoint, CBV and CBN simply typed λ-calculi are nothing more than subsystems of CBPV.

Thus, while it was common practice in the 1990s to carry out each piece of semantic research twice—once for CBV and once for CBN—this is no longer necessary. We can just work with CBPV, and doing so gives a precise, fine-grained understanding that was previously missing. For example, game models were developed for CBN and CBV [Hyland and Ong 2000; Nickau 1996; McCusker 2000; Abramsky and McCusker 1998; Honda and Yoshida 1997], and they involve two kinds of move, dubbed "Question" and "Answer". These correspond, respectively, to the CBPV operations of forcing and returning [Levy 2004, Chapter 8], which the CBN and CBV calculi do not make explicit.

Working in CBPV has other advantages. For example, in the presence of type recursion (Section 4), CBV can express *simple lists* and CBN can express *lazy lists*, so CBPV can express both.[2] CBPV also provides optimizations that compilers can exploit—an example is given in Section 6.

The name "call-by-push-value" has sometimes been criticized, as CBPV is not inherently about stack usage, or at least no more than CBV and CBN are. I chose this name because, whilst in the CBN stack machine (known as the "Krivine machine") function application causes a *term* to be pushed, in the CBPV version it causes a *value* to be pushed. (This is explained in Section 3.3.) So, in an alternative history, CBN would be dubbed "call-by-push", and we could then say that call-by-push-value combines call-by-push and call-by-value.

1.2. Overview of the article

To set the scene, we begin (Section 2) with an account of the simply typed λ-calculus. This, together with special equations known as β- and η-laws, constitutes the canonical pure functional programming language. The word "pure" means that effects are excluded.

Then we move to CBPV, which modifies simply typed λ-calcuus to allow the inclusion of effects, while retaining all the β- and η-laws. We give the syntax and operational semantics using an interpreter (Section 3.2), and using the CK-machine, which involves stack terms (Section 3.3). We give syntax and operational semantics for various computational effects (Section 4), and then give a denotational model for each extension (Section 5), taking care to motivate each model via computational ideas. Section 6 presents the CBV and CBN subsystems of CBPV, and we wrap up in Section 7.

2. PURE FUNCTIONAL PROGRAMMING

2.1. Notation for sets and elements

Let's begin with some mathematical notation. For sets A and B, we write $A \to B$ or B^A for the set of all functions from A to B. We write $A \times B$ for the set $\{\langle a, b\rangle \mid a \in A, b \in B\}$, and $A + B$ for the tagged union $\{\mathsf{inl}\ a \mid a \in A\} \cup \{\mathsf{inr}\ b \mid b \in B\}$, where $\mathsf{inl}\ a \stackrel{\text{def}}{=} \langle 0, a\rangle$ and $\mathsf{inr}\ b \stackrel{\text{def}}{=} \langle 1, b\rangle$. We write 1 for the singleton set $\{\langle\rangle\}$ and 0 for the empty set.

We write λ for function abstraction, e.g. $\lambda x_{\mathbb{N}}.\, x + 3$ is the function sending a natural number x to $x + 3$. And we write application as juxtaposition, so $(\lambda x_{\mathbb{N}}.\, x + 3)\, 5$ is 8.

[1] Strictly speaking, the preservation of operational semantics is up to administrative reductions, and that of denotational semantics is up to isomorphism, but these are small matters.

[2] In CBV, although in principle thunks can be used to express lazy lists, it is rather challenging to correctly write the concatenation operator (for example), or even its type. See also [MacQueen et al. 1998].

We write let for a list of local definitions, so let $(x$ be $3, y$ be $4).\ x + y$ is 7. For an ordered pair x, we write $x.$left for its left component and $x.$right for its right component. Finally, we write match for pattern-matching, so match inl 8 as $(\mathsf{inl}\ x.\ x + 2, \mathsf{inr}\ y.\ y + 3)$ is 10, and match $\langle 3, 5\rangle$ as $\langle x, y\rangle.\ x + y$ is 8.

2.2. Formal syntax

The simply typed λ-calculus is a formalized version of the notation given in Section 2.1. Its types are as follows.

$$A ::= \quad 0 \mid A + A \mid 1 \mid A \times A \mid A \to A$$

Additional "base types" can be included, but we shall not do so. Let me stress that all the type constructors are equally important for our purpose, although some authors include only $\to$ and omit or downplay the others.

A *variable declaration* $\mathtt{x} : A$ consists of a variable and a type. A *typing context* Γ is a list of such declarations with no variable declared more than once. The judgement $\Gamma \vdash M : A$ is intended to say that M is a term of type A whose free variables all appear in Γ with an appropriate type. Let us now see the typing rules for this judgement.

The rules for variables and local definitions are as follows:

$$\frac{(\mathtt{x} : A) \in \Gamma}{\Gamma \vdash \mathtt{x} : A} \qquad \frac{\overrightarrow{\Gamma \vdash N : A} \quad \Gamma, \overrightarrow{\mathtt{x} : A} \vdash M : B}{\Gamma \vdash \mathtt{let}\ \overrightarrow{\mathtt{x}\ \mathtt{be}\ N}.\ M : B}$$

where the vector notation $\overrightarrow{b}$ stands for $b_0, \ldots, b_{n-1}$. The type $A \to B$ has an introduction rule

$$\frac{\Gamma, \mathtt{x} : A \vdash M : B}{\Gamma \vdash \lambda \mathtt{x}.\ M : A \to B}$$

and an elimination rule

$$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

The introduction rules for $A + B$ are as follows:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathtt{inl}\ M : A + B} \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathtt{inr}\ M : A + B}$$

and the elimination rules for $A + B$ and 0 are as follows:

$$\frac{\Gamma \vdash N : A + B \quad \Gamma, \mathtt{x} : A \vdash M : C \quad \Gamma, \mathtt{y} : B \vdash M' : C}{\Gamma \vdash \mathtt{match}\ N\ \mathtt{as}\ (\mathtt{inl}\ \mathtt{x}.\ M, \mathtt{inr}\ \mathtt{y}.\ M') : C} \qquad \frac{\Gamma \vdash N : 0}{\Gamma \vdash \mathtt{match}\ N\ \mathtt{as}\ (\,) : C}$$

The introduction rules for $A \times B$ and 1 are as follows:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \times B} \qquad \frac{}{\Gamma \vdash \langle\,\rangle : 1}$$

Elimination rules for these types can be given using projections:

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M.\mathsf{left} : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M.\mathsf{right} : B}$$

Or using pattern-matching:

$$\frac{\Gamma \vdash N : A \times B \quad \Gamma, \mathtt{x} : A, \mathtt{y} : B \vdash M : C}{\Gamma \vdash \mathtt{match}\ N\ \mathtt{as}\ \langle \mathtt{x}, \mathtt{y}\rangle.\ M : C} \qquad \frac{\Gamma \vdash N : 1 \quad \Gamma, \vdash M : C}{\Gamma \vdash \mathtt{match}\ N\ \mathtt{as}\ \langle\,\rangle.\ M : C}$$

2.3. Infinitary extension

The simply typed λ-calculus has an infinitary extension with the following types:

$$A ::= \quad 0 \mid A + A \mid 1 \mid A \times A \mid A \to A \mid \textstyle\sum_{i\in\mathbb{N}} A_i \mid \prod_{i\in\mathbb{N}} A_i$$

The type $\sum_{i\in\mathbb{N}} A_i$ has the following introduction and elimination rules:

$$\frac{\Gamma \vdash M : A_i}{\Gamma \vdash \mathtt{in}_i\, M : \sum_{i\in\mathbb{N}} A_i}\, i \in \mathbb{N} \qquad \frac{\Gamma \vdash N : \sum_{i\in I} A_i \quad (\Gamma, \mathtt{x} : A_i \vdash M_i : B)_{i\in\mathbb{N}}}{\Gamma \vdash \mathtt{match}\ N\ \mathtt{as}\ (\mathtt{in}_i\, x.\, M_i)_{i\in I} : \underline{B}}$$

The type $\prod_{i\in\mathbb{N}} A_i$ has the following introduction and elimination rules:

$$\frac{(\Gamma \vdash M_i : A_i)_{i\in\mathbb{N}}}{\Gamma \vdash \lambda(i.\, M_i)_{i\in\mathbb{N}} : \prod_{i\in\mathbb{N}} A_i} \qquad \frac{\Gamma \vdash M : \prod_{i\in\mathbb{N}} A_i}{\Gamma \vdash M\hat{\imath} : A_{\hat{\imath}}}\, i \in \mathbb{N}$$

Henceforth we shall include these extra types, but readers who prefer to stick to finitary syntax can ignore them. We abbreviate $\mathtt{bool} \stackrel{\text{def}}{=} 1 + 1$ and $\mathtt{nat} \stackrel{\text{def}}{=} \sum_{n\in\mathbb{N}} 1$ with the usual constants.

2.4. Denotational semantics: sets and functions

We furnish the calculus with a denotational semantics. Firstly, each type A denotes a set $[\![A]\!]$. This is given by induction on A, using clauses such as

$$[\![A \to B]\!] \stackrel{\text{def}}{=} [\![A]\!] \to [\![B]\!]$$

Let me stress that $\to$ in the LHS is a formal symbol, whereas $\to$ in the RHS is an operation on sets. Each typing context Γ also denotes a set, given by

$$[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{(\mathtt{x}:A)\in\Gamma} [\![A]\!]$$

An element of this set is called a *semantic* Γ*-environment* , as it provides an element for each variable.

Finally each term $\Gamma \vdash M : B$ denotes a function from $[\![\Gamma]\!]$ to $[\![B]\!]$. It is written $[\![\Gamma \vdash M : B]\!]$, or $[\![M]\!]$ for short, and is defined by induction on the derivation as follows:

$$\begin{array}{lcl} [\![\mathtt{x}]\!] & : & \rho \mapsto \rho(\mathtt{x}) \\ [\![\lambda\mathtt{x}.\, M]\!] & : & \rho \mapsto \lambda a \in [\![A]\!].\, [\![M]\!](\rho, \mathtt{x} \mapsto a) \\ [\![MN]\!] & : & \rho \mapsto ([\![M]\!]\rho)([\![N]\!]\rho) \end{array}$$

and so forth. Strictly speaking, we must prove *semantic coherence*, the property that any two derivations of a judgement $\Gamma \vdash M : B$ lead to the same denotation. Alternatively, sufficient type annotations can be put into the syntax, so that derivations are unique. Henceforth we ignore the issue.

2.5. Weakening and substitution

If $\Gamma \subseteq \Gamma'$, i.e. each declaration in Γ appears also in Γ', then the judgement $\Gamma \vdash M : A$ implies $\Gamma' \vdash M : A$. The *weakening lemma* obtains $[\![\Gamma' \vdash M : A]\!]$ from $[\![\Gamma \vdash M : A]\!]$ by environment restriction.

Terms that are α*-equivalent*, i.e. differ only by renaming of bound variables, can be shown to have the same denotation. Henceforth we identify such terms.

For typing contexts Γ and Γ', a *substitution* $k : \Gamma \to \Gamma'$ sends each variable $(\mathtt{x} : A) \in \Gamma$ to $(k(\mathtt{x}) : A) \in \Gamma'$, and it denotes a function $[\![k]\!] : [\![\Gamma']\!] \to [\![\Gamma]\!]$ sending ρ to $(\rho(k(\mathtt{x}))_{(\mathtt{x}:A)\in\Gamma}$. Any term $\Gamma \vdash M : B$ gives a substituted term $\Gamma' \vdash M[k] : A$, which is M with each

identifier $(\mathtt{x} : A) \in \Gamma$ replaced by $k(\mathtt{x})$, avoiding capture. The *substitution lemma* says that we can obtain $[\![M[k]]\!]$ from $[\![M]\!]$ and $[\![k]\!]$ by composition.

There is a connection to category theory, as contexts and substitutions form a category, and the mapping $\Gamma \mapsto [\![\Gamma]\!]$ gives a contravariant functor from this category to **Set**.

2.6. The $\beta\eta$-theory

Two kinds of equation between terms are important. Firstly, we have the *β-laws*, which can be seen as a reduction mechanism. For example, here is the β-law for functions:

$$\frac{\Gamma \vdash N : A \quad \Gamma, \mathtt{x} : A \vdash M : B}{\Gamma \vdash (\lambda \mathtt{x}. .M)N = M[N/\mathtt{x}] : B}$$

Here is one for pairs using projection syntax:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N).\mathsf{left} = M : A}$$

Here is one for tagged terms:

$$\frac{\Gamma \vdash N : A \quad \Gamma, \mathtt{x} : A \vdash M : C \quad \Gamma, \mathtt{y} : B \quad M' : C}{\Gamma \vdash \mathtt{match}\ \mathtt{inl}\ N\ \mathtt{as}\ (\mathtt{inl}\ \mathtt{x}.\, M, \mathtt{inr}\ \mathtt{y}.\, M') = M[N/\mathtt{x}] : C}$$

And one for local definition:

$$\frac{\overrightarrow{\Gamma \vdash N : A} \quad \Gamma, \overrightarrow{\mathtt{x} : A} \vdash M : B}{\Gamma \vdash \mathtt{let}\ \overrightarrow{\mathtt{x}\ \mathtt{be}\ N}.\ M = M[\overrightarrow{N/\mathtt{x}}] : B}$$

Secondly, we have the *η-laws*, which can be seen as an expansion mechanism. For example, the following η-law for the type $A \to B$ expresses the idea that everything of this type is a λ-abstraction:

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash M = \lambda \mathtt{x}.\, M\mathtt{x} : A \to B}\ \mathtt{x} \# \Gamma$$

where the *freshness condition* $\mathtt{x} \# \Gamma$ means that $\mathtt{x}$ does not appear in Γ. The following η-law for the type $A \times B$ (using projection syntax) expresses the idea that everything of this type is a pair:

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M = (M.\mathsf{left}, M.\mathsf{right}) : A \times B}$$

The following η-law for the type $A + B$ expresses the idea that everything of this type is constructed by `inl` or `inr`:

$$\frac{\Gamma \vdash N : A + B \quad \Gamma, \mathtt{z} : A + B \vdash M : C}{\Gamma \vdash M[N/\mathtt{z}] = \mathtt{match}\ N\ \mathtt{as}\ (\mathtt{inl}\ \mathtt{x}.\, M[\mathtt{inl}\ \mathtt{x}/\mathtt{z}],\ \mathtt{inr}\ \mathtt{y}.\, M[\mathtt{inr}\ \mathtt{y}/\mathtt{z}]) : C}\ \mathtt{x}, \mathtt{y} \# \Gamma$$

The *$\beta\eta$-theory* is defined to be the least congruence that contains all the β- and η-laws. We write $\Gamma \vdash M = N : A$ to say that M and N are equated. The typing context Γ is essential because, for example, the equation $\mathtt{x} : 0 \vdash \mathtt{true} = \mathtt{false} : \mathtt{bool}$ holds but the equation $\vdash \mathtt{true} = \mathtt{false} : \mathtt{bool}$ does not.

We can show that $\Gamma \vdash M = N : A$ implies $\Gamma' \vdash M[k] = N[k] : A$ for all $k :\ \Gamma \to \Gamma'$, and also implies $[\![M]\!] = [\![N]\!]$.

3. INTRODUCING CALL-BY-PUSH-VALUE

3.1. Preface

In this section, I introduce CBPV by "pulling it out of a hat". I have made this decision for the sake of accessibility and brevity, but it has the drawback of requiring the reader to accept the calculus on trust to a certain extent. Only in Section 6 do we see the translations from CBV and CBN, and the semantics they give rise to.

(The alternative—more suitable for sceptical readers who want to know exactly why CBPV is formulated the way it is—would be to start with an arduous study of the CBV and CBN simply typed λ-calculi, including various syntactic options and the associated semantics. A comparison of all these semantics reveals that the same pattern of operations appears in each one, and the components of this pattern constitute CBPV. This story is rather long and intricate, but makes clear that CBPV is an *empirically observed* calculus.)

The name "call-by-push-value" is unintelligible in Section 3.2, as it is only in Section 3.3 that we learn about the stack.

3.2. Values and Computations

To present CBPV, we begin with the distinction between *values*, which have value type, and *computations*, which have computation type. The CBPV types are given as follows.

$$\begin{array}{lll} \text{value type} & A ::= & U\underline{B} \mid 1 \mid A \times A \mid 0 \mid A + A \mid \sum_{i\in\mathbb{N}} A_i \\ \text{computation type} & \underline{B} ::= & FA \mid A \to \underline{B} \mid 1_{\Pi} \mid \underline{B} \,\Pi\, \underline{B} \mid \prod_{i\in\mathbb{N}} \underline{B}_i \end{array}$$

This includes the type $U\underline{B}$ of thunks of computations in $\underline{B}$, and the type FA of computations that aim to return a value in A. We write $\times$ for multiplication of value types and Π for that of computation types, and 1 and 1_{Π} for their respective units. We think of $\underline{A} \,\Pi\, \underline{B}$ as a type of functions on the set $\{\mathsf{left}, \mathsf{right}\}$, and 1_{Π} as a type of functions on the empty set. We define the types $\mathtt{bool} \stackrel{\text{def}}{=} 1 + 1$ and $\mathtt{nat} \stackrel{\text{def}}{=} \sum_{i\in\mathbb{N}} 1$.

As in Section 2.3, the infinitary connectives $\sum_{i\in\mathbb{N}}$ and $\prod_{i\in\mathbb{N}}$ are an optional extra, and readers who prefer to stick to finitary syntax can ignore them. Here is a streamlined presentation of the type syntax:

$$\begin{array}{lll} \text{value type} & A ::= & U\underline{B} \mid 1 \mid A \times A \mid \sum_{i\in I} A_{\mathbb{N}} \\ \text{computation type} & \underline{B} ::= & FA \mid A \to \underline{B} \mid \prod_{i\in I} \underline{B}_i \end{array}$$

where the set I can be either $\emptyset$ or $\{\mathsf{left}, \mathsf{right}\}$ or $\mathbb{N}$. Any chosen element of I is written $\hat{\imath}$.

A *variable declaration* $\mathtt{x} : A$ consists of a variable with value type. A *typing context* Γ is a list of such declarations with no variable declared more than once. The typing judgements are $\Gamma \vdash^{\mathsf{v}} V : A$ for a value (where A is a value type), and $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ for a computation (where $\underline{B}$ is a computation type). They are defined inductively in Figure 1. A *closed* term is one with no free variables. The *terminals* are the closed computations of the form $\mathtt{return}\ V$ or $\lambda(i.\, M_i)_{i\in I}$ or $\lambda x.\, M$. The following interpreter says how to evaluate a closed computation $M : \underline{B}$ to a terminal $T : \underline{B}$.

— To evaluate a terminal T: return T.
— To evaluate $\mathtt{let}\ \mathtt{x}\ \mathtt{be}\ \overrightarrow{V}.\, M$: evaluate $M[\overrightarrow{V/\mathtt{x}}]$.
— To evaluate $M\ \mathtt{to}\ \mathtt{x}.\, N$: first evaluate M to $\mathtt{return}\ V$, then evaluate $N[V/\mathtt{x}]$.
— To evaluate $\mathtt{force}\ \mathtt{thunk}\ M$: evaluate M.
— To evaluate $\mathtt{match}\ \mathtt{in}_i\ V\ \mathtt{as}\ (\mathtt{in}_i\ \mathtt{x}.\, M_i)_{i\in I}$: evaluate $M_i[V/\mathtt{x}]$.
— To evaluate $\mathtt{match}\ \langle\rangle\ \mathtt{as}\ \langle\rangle.\, M$: evaluate M.
— To evaluate $\mathtt{match}\ \langle V, V'\rangle\ \mathtt{as}\ \langle \mathtt{x}, \mathtt{y}\rangle.\, M$: evaluate $M[V/\mathtt{x}, V'/\mathtt{y}]$.
— To evaluate $M\hat{\imath}$: first evaluate M to $\lambda(i.\, N_i)_{i\in I}$, then evaluate N_i.

$$\frac{}{\Gamma \vdash^{\mathsf{v}} x : A}\ (x : A) \in \Gamma \qquad \frac{\overrightarrow{\Gamma \vdash^{\mathsf{v}} V : A} \quad \Gamma, \overrightarrow{\mathtt{x} : A} \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{let}\ \overrightarrow{\mathtt{x}\ \mathtt{be}\ V}.\ M : \underline{B}}\ \mathrm{let}$$

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{v}} \mathtt{thunk}\ M : U\underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : U\underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{force}\ V : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{c}} \mathtt{return}\ V : FA} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : FA \quad \Gamma, x : A \vdash^{\mathsf{c}} N : \underline{B}}{\Gamma \vdash^{\mathsf{c}} M\ \mathtt{to}\ x.\ N : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A_{\hat{\imath}}}{\Gamma \vdash^{\mathsf{v}} \mathtt{in}_{\hat{\imath}}\ V : \sum_{i \in I} A_i}\ \hat{\imath} \in I \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : \sum_{i \in I} A_i \quad \Gamma, \mathtt{x} : A_i \vdash^{\mathsf{c}} M_i : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{match}\ V\ \mathtt{as}\ (\mathtt{in}_i\, \mathtt{x}.\ M_i)_{i \in I} : \underline{B}}$$

$$\frac{}{\Gamma \vdash^{\mathsf{v}} \langle\rangle : 1} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : 1 \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{v}} \mathtt{match}\ V\ \mathtt{as}\ \langle\rangle.\ M : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A \quad \Gamma \vdash^{\mathsf{v}} V' : A'}{\Gamma \vdash^{\mathsf{v}} \langle V, V' \rangle : A \times A'} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \times A' \quad \Gamma, \mathtt{x} : A, \mathtt{y} : A' \vdash^{\mathsf{c}} M : \underline{C}}{\Gamma \vdash^{\mathsf{c}} \mathtt{match}\ V\ \mathtt{as}\ \langle \mathtt{x}, \mathtt{y} \rangle.\ M : \underline{C}}$$

$$\frac{(\Gamma \vdash^{\mathsf{c}} M_i : \underline{B}_i)_{i \in I}}{\Gamma \vdash^{\mathsf{c}} \lambda(i.\ M_i)_{i \in I} : \prod_{i \in I} \underline{B}_i} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : \prod_{i \in I} \underline{B}_i}{\Gamma \vdash^{\mathsf{c}} M\hat{\imath} : \underline{B}_{\hat{\imath}}}\ \hat{\imath} \in I$$

$$\frac{\Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \lambda \mathtt{x}.\ M : A \to \underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : A \to \underline{B} \quad \Gamma \vdash^{\mathsf{v}} V : A}{\Gamma \vdash^{\mathsf{c}} MV : \underline{B}}$$

Fig. 1. CBPV typing rules for values $\Gamma \vdash^{\mathsf{v}} V : A$ and computations $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$

— To evaluate MV: first evaluate M to $\lambda \mathtt{x}.\ N$, then evaluate $N[V/\mathtt{x}]$.

Figure 2 inductively defines a single-valued relation $M \Downarrow T$, where $M : \underline{B}$ is a closed computation and $T : \underline{B}$ a terminal. The intended reading of $M \Downarrow T$ is that the evaluation of M returns T, so $\Downarrow$ is called the *evaluation* relation. It is clearly single-valued, and can be shown to be total [Levy 2006a, Proposition 1].

A *program* is a closed computation of type FA, where A is a ground type such as `bool` or `nat`. Again, the definition of a program's "behaviour" will depend on the effects we are considering. Two terms $\Gamma \vdash M, N : A$ are said to be *observationally equivalent* when replacing one by the other within a program does not affect the program's behaviour.

The CBPV equational theory is shown in Figure 3, and the equations shown are observational equivalences for all the effects we consider. Note that CBPV satisfies the η-law for $+$ types (unlike CBN) and also for $\to$ types (unlike CBV).

3.3. Stacks

There are various kinds of operational semantics. So far we have seen two of them: interpreter and big-step semantics. Another is the *CK-machine* [Felleisen and Friedman 1986], where C stands for "Computation" and K for "stacK". The idea is that a computation is evaluated using a stack, which is initially empty. When we evaluate N, and are instructed (by the interpreter) to first evaluate one part of N, the rest is placed on the stack until this evaluation phase is complete.

$$\frac{T : \underline{B} \text{ terminal}}{T \Downarrow T} \qquad \frac{M \Downarrow T}{\texttt{force thunk}\ M \Downarrow T} \qquad \frac{M \Downarrow \texttt{return}\ V \quad N[V/\mathtt{x}] \Downarrow T}{M\ \texttt{to}\ \mathtt{x}.\ N \Downarrow T}$$

$$\frac{M[\overrightarrow{V/\mathtt{x}}] \Downarrow T}{\texttt{let}\ \overrightarrow{\mathtt{x}\ \texttt{be}\ V}.\ M \Downarrow T} \qquad \frac{M \Downarrow T}{\texttt{match}\ \langle\rangle\ \texttt{as}\ \langle\rangle.\ M \Downarrow T}$$

$$\frac{M \Downarrow \lambda \mathtt{x}.\ N \quad N[V/\mathtt{x}] \Downarrow T}{MV \Downarrow T} \qquad \frac{M[V/\mathtt{x}, V'/\mathtt{y}] \Downarrow T}{\texttt{match}\ \langle V, V'\rangle\ \texttt{as}\ \langle \mathtt{x}, \mathtt{y}\rangle.\ M \Downarrow T}$$

$$\frac{M \Downarrow \lambda(i.\ N_i)_{i\in I} \quad N_{\hat{\imath}} \Downarrow T}{M\hat{\imath} \Downarrow T} \qquad \frac{M_{\hat{\imath}}[V/\mathtt{x}] \Downarrow T}{\texttt{match}\ \mathtt{in}_{\hat{\imath}}\ V\ \texttt{as}\ (\mathtt{in}_i\ \mathtt{x}.\ M_i)_{i\in I} \Downarrow T}$$

Fig. 2. Big-step semantics of CBPV

β-laws

$$\begin{array}{rcl}
\texttt{let}\ \overrightarrow{\mathtt{x}\ \texttt{be}\ V}.\ M & = & M[\overrightarrow{V/\mathtt{x}}] \\
(\texttt{return}\ V)\ \texttt{to}\ \mathtt{x}.\ M & = & M[V/\mathtt{x}] \\
\texttt{force thunk}\ M & = & M \\
\texttt{match}\ \mathtt{in}_{\hat{\imath}}\ V\ \texttt{as}\ (\mathtt{in}_i\ \mathtt{x}.\ M_i)_{i\in I} & = & M_{\hat{\imath}}[V/\mathtt{x}] \\
\texttt{match}\ \langle\rangle\ \texttt{as}\ \langle\rangle.\ M & = & M \\
\texttt{match}\ \langle V, V'\rangle\ \texttt{as}\ \langle \mathtt{x}, \mathtt{y}\rangle.\ M & = & M[V/\mathtt{x}, V'/\mathtt{y}] \\
\lambda(i.\ M_i)_{i\in I}\hat{\imath} & = & M_{\hat{\imath}} \\
(\lambda \mathtt{x}.M)V & = & M[V/\mathtt{x}]
\end{array}$$

η-laws

$$\begin{array}{rcl}
V & = & \texttt{thunk force}\ V \\
M[V/\mathtt{z}] & = & \texttt{match}\ V\ \texttt{as}\ (\mathtt{in}_i\ \mathtt{x}.\ M[\mathtt{in}_i\ \mathtt{x}/\mathtt{z}])_{i\in I} \\
M[V/\mathtt{z}] & = & \texttt{match}\ V\ \texttt{as}\ \langle\rangle.M[\langle\rangle/\mathtt{z}] \\
M[V/\mathtt{z}] & = & \texttt{match}\ V\ \texttt{as}\ \langle \mathtt{x}, \mathtt{y}\rangle.M[\langle \mathtt{x}, \mathtt{y}\rangle/\mathtt{z}] \\
M & = & \lambda(i.\ Mi)_{i\in I} \\
M & = & \lambda \mathtt{x}.(M\mathtt{x})
\end{array}$$

Sequencing laws

$$\begin{array}{rcl}
M & = & M\ \texttt{to}\ \mathtt{x}.\ \texttt{return}\ \mathtt{x} \\
(P\ \texttt{to}\ \mathtt{x}.\ M)\ \texttt{to}\ \mathtt{y}.\ N & = & P\ \texttt{to}\ \mathtt{x}.\ (M\ \texttt{to}\ \mathtt{y}.\ N) \\
(P\ \texttt{to}\ \mathtt{x}.\ M)\hat{\imath} & = & P\ \texttt{to}\ \mathtt{x}.\ (M\hat{\imath}) \\
(P\ \texttt{to}\ \mathtt{x}.\ M)V & = & P\ \texttt{to}\ \mathtt{x}.\ (MV)
\end{array}$$

Fig. 3. Equational laws of CBPV (typing assumptions omitted)

The judgement $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ says that, once values are provided for the identifiers in Γ, the stack K may accompany a computation of type $\underline{B}$. The *top-level type* $\underline{C}$ is the type of the computation initially loaded onto the machine. (Some readers may prefer to think of K as an "evaluation context" of type $\underline{C}$, with a hole of type $\underline{B}$.) The typing rules are shown in Figure 4.

The CK-machine itself is shown in Figure 5. A configuration $M, K : \underline{C}$ consists of a computation type $\underline{B}$ (which we omit), a computation $\vdash^{\mathsf{c}} M : \underline{B}$ and a stack $\vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow$

$$\frac{}{\Gamma \vdash^{\mathsf{k}} \mathtt{nil} : \underline{C} \Longrightarrow \underline{C}} \qquad \frac{\Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} M : \underline{B} \quad \Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \mathtt{to}\ \mathtt{x}.\ M :: K : FA \Longrightarrow \underline{C}}$$

$$\frac{\Gamma \vdash^{\mathsf{k}} K : \underline{B}_{\hat{\imath}} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \hat{\imath} :: K : \prod_{i \in I} \underline{B}_i \Longrightarrow \underline{C}}\ \hat{\imath} \in I \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \quad \Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} V :: K : A \to \underline{B} \Longrightarrow \underline{C}}$$

Fig. 4. CBPV typing rules for stacks $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$

Initial configuration for evaluation of M

M			$\mathtt{nil}$	

Transitions

$\mathtt{let}\ \overrightarrow{\mathtt{x}\ \mathtt{be}\ V}.\ M$	K	$\rightsquigarrow$	$M[\overrightarrow{V/\mathtt{x}}]$	K
$M\ \mathtt{to}\ \mathtt{x}.\ N$	K	$\rightsquigarrow$	M	$\mathtt{to}\ \mathtt{x}.\ N :: K$
$\mathtt{return}\ V$	$\mathtt{to}\ \mathtt{x}.\ N :: K$	$\rightsquigarrow$	$N[V/\mathtt{x}]$	K
$\mathtt{force}\ \mathtt{thunk}\ M$	K	$\rightsquigarrow$	M	K
$\mathtt{match}\ \mathtt{in}_i\ V\ \mathtt{as}\ (\mathtt{in}_i\ \mathtt{x}.\ M_i)_{i \in I}$	K	$\rightsquigarrow$	$M_{\hat{\imath}}[V/\mathtt{x}]$	K
$\mathtt{match}\ \langle V, V' \rangle\ \mathtt{as}\ \langle \mathtt{x}, \mathtt{y} \rangle.\ M$	K	$\rightsquigarrow$	$M[V/\mathtt{x}, V'/\mathtt{y}]$	K
$M\hat{\imath}$	K	$\rightsquigarrow$	M	$\hat{\imath} :: K$
$\lambda(i.\ M_i)_{i \in I}$	$\hat{\imath} :: K$	$\rightsquigarrow$	$M_{\hat{\imath}}$	K
MV	K	$\rightsquigarrow$	M	$V :: K$
$\lambda \mathtt{x}.\ M$	$V :: K$	$\rightsquigarrow$	$M[V/\mathtt{x}]$	K

Terminal configuration yielding a terminal T

T			$\mathtt{nil}$	

Fig. 5. CK-Machine For CBPV

$\underline{C}$. We write $\rightsquigarrow$ for the transition relation between configurations. It is easily seen that, if M, K is not terminal, then $M, K \rightsquigarrow N, L$ for a unique configuration $N, L : \underline{C}$.

So far, we have used operator-first notation for application. But we can also use operand-first, writing $V`M$ instead of $MV$, and $\hat{\imath}`M$ instead of $M\hat{\imath}$. Using this notation, $\lambda \mathtt{x}$ can be read as "pop x", and $V`$ can be read as "push $V$". Likewise, $\lambda i$ can be read as "pop $i$" and $\hat{\imath}`$ can be read as "push $\hat{\imath}$". This is illustrated by following program, which uses printing and arithmetic.

```
print "hello0".
let 3 be x.
let  thunk (
        print "hello1".
        λz.
        print "we just popped "z.
        return x + z
    ) be y.
print "hello2".
```

```
( print "hello3".
  7‘
  print "we just pushed 7".
  force y
) to w.
print "w is bound to "w.
return w + 5
```

Following the CK-machine transitions (and evaluating arithmetical expressions where necessary), we print the following:

```
hello0
hello2
hello3
we just pushed 7
hello1
we just popped 7
w is bound to 10
```

and finally return the value 15. We now can understand the CBPV slogan: "A value is, a computation does." Here is a summary of the types, from this perspective.

— A value of type $U\underline{B}$ is a thunk of a computation of type $\underline{B}$.
— A value of type 1 is the empty tuple $\langle\rangle$.
— A value of type $A \times A'$ is a pair consisting of a value of type A and a value of type A'.
— A value of type $\sum_{i\in I} A_i$ is a pair consisting of a tag $\hat{\imath} \in I$ and a value of type $A_{\hat{\imath}}$.
— A computation of type FA aims to return a value of type A.
— A computation of type $A \to \underline{B}$ aims to pop a value of type A and then behave in $\underline{B}$.
— A computation of type $\prod_{i\in I} \underline{B}_i$ aims to pop a tag $\hat{\imath} \in \underline{B}$ and then behave in $\underline{B}_{\hat{\imath}}$.

There are three important operations on stacks [Levy 2005]. Firstly, the *dismantling* of a $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ onto a computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ is written $\Gamma \vdash^{\mathsf{c}} M \bullet K : \underline{C}$. It is defined by induction on K as follows.

$$\begin{aligned} M \bullet \mathtt{nil} &\stackrel{\text{def}}{=} M \\ M \bullet (\mathtt{to}\ \mathtt{x}.\ N :: K) &\stackrel{\text{def}}{=} (M\ \mathtt{to}\ \mathtt{x}.\ N) \bullet K \\ M \bullet (\hat{\imath} :: K) &\stackrel{\text{def}}{=} (M\hat{\imath}) \bullet K \\ M \bullet (V :: K) &\stackrel{\text{def}}{=} (MV) \bullet K \end{aligned}$$

Secondly, the *concatenation* of stacks $\Gamma \vdash^{\mathsf{k}} K : \underline{A} \Longrightarrow \underline{B}$ and $\Gamma \vdash^{\mathsf{k}} L : \underline{B} \Longrightarrow \underline{C}$ is written $\Gamma \vdash^{\mathsf{k}} K +\!\!+ L : \underline{A} \Longrightarrow \underline{C}$. It is defined by induction on K as follows.

$$\begin{aligned} \mathtt{nil} +\!\!+ L &\stackrel{\text{def}}{=} L \\ (\mathtt{to}\ \mathtt{x}.\ N :: K) +\!\!+ L &\stackrel{\text{def}}{=} \mathtt{to}\ \mathtt{x}.\ N :: (K +\!\!+ L) \\ (\hat{\imath} :: K) +\!\!+ L &\stackrel{\text{def}}{=} \hat{\imath} :: (K +\!\!+ L) \\ (V :: K) +\!\!+ L &\stackrel{\text{def}}{=} V :: (K +\!\!+ L) \end{aligned}$$

Thirdly, given a stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$, we can *substitute* values for the variables in Γ.

A stack from an F type is called a *continuation*, and the following is called the *pure continuation law*:

$$\frac{\Gamma \vdash^{\mathsf{k}} K : FA \Longrightarrow \underline{B} \quad \Gamma \vdash^{\mathsf{k}} L : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} K +\!\!+ L = \mathtt{to}\ \mathtt{x}.\ ((\mathtt{return}\ \mathtt{x}) \bullet K) :: L : FA \Longrightarrow \underline{C}}$$

It can be seen as an η-law for the F connective.

4. EXTENDING CBPV WITH VARIOUS EFFECTS

So far we have seen only the pure CBPV calculus. Let's now adapt the syntax and operational semantics to incorporate various effects.

4.1. Recursion with isorecursive types

Besides the term-level variables ($\mathtt{x}$), we also have value type variables ($\mathtt{X}$) and computation type variables ($\underline{\mathtt{X}}$). The type syntax is extended as follows:

$$A ::= \cdots \mid \mathtt{X} \mid \mathtt{rec}\ \mathtt{X}.\ A$$
$$\underline{B} ::= \cdots \mid \underline{\mathtt{X}} \mid \mathtt{rec}\ \underline{\mathtt{X}}.\ \underline{B}$$

Only closed types may be used in a variable declaration or typing judgement. The term syntax is extended as follows:

$$\frac{\Gamma, \mathtt{x} : U\underline{B} \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{rec}\ \mathtt{x}.\ M : \underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A[\mathtt{rec}\ X.\ A/X]}{\Gamma \vdash^{\mathsf{v}} \mathtt{fold}\ V : \mathtt{rec}\ X.\ A}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{rec}\ X.\ A \quad \Gamma, \mathtt{x} : A[\mathtt{rec}\ X.\ A/X] \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{match}\ V\ \mathtt{as}\ \mathtt{fold}\ \mathtt{x}.\ M : \underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B}[\mathtt{rec}\ \underline{X}.\ \underline{B}/\underline{X}]}{\Gamma \vdash^{\mathsf{c}} \mathtt{fold}\ M : \mathtt{rec}\ \underline{X}.\ \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \mathtt{rec}\ \underline{X}.\ \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{unfold}\ M : \underline{B}[\mathtt{rec}\ \underline{X}.\ \underline{B}/\underline{X}]} \qquad \frac{\Gamma \vdash^{\mathsf{k}} K : \underline{B}[\mathtt{rec}\ \underline{X}.\ \underline{B}/\underline{X}] \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \mathtt{unfold} :: K : \mathtt{rec}\ \underline{X}.\ \underline{B} \Longrightarrow \underline{C}}$$

We consider $\mathtt{fold}\ M$ to be terminal, and extend the interpreter as follows:

— To evalute $\mathtt{rec}\ \mathtt{x}.\ M$: evaluate $M[\mathtt{thunk}\ \mathtt{rec}\ \mathtt{x}.\ M/\mathtt{x}]$.
— To evaluate $\mathtt{match}\ \mathtt{fold}\ V\ \mathtt{as}\ \mathtt{fold}\ \mathtt{x}.\ M$: evaluate $M[V/\mathtt{x}]$.
— To evaluate $\mathtt{unfold}\ M$: evaluate M to $\mathtt{fold}\ N$, then evaluate N.

The $\Downarrow$ relation and CK-machine transltion are extended accordingly. A closed computation M is *divergent* when execution of it runs forever, which is equivalent to the nonexistence of T such that $M \Downarrow T$. We write $M \Uparrow$ to say that M diverges.

4.2. Nondeterminism

We add to the language the following typing rule

$$\frac{(\Gamma \vdash^{\mathsf{c}} M_i : \underline{B})_{i \in I}}{\Gamma \vdash^{\mathsf{c}} \mathtt{choose}\ (M_i)_{i \in I} : \underline{B}}$$

According to preference, we can require I to be $\{\mathsf{left}, \mathsf{right}\}$, or to be $\mathbb{N}$, or allow it to be any set (in which case the terms form a proper class).

To evaluate $\mathtt{choose}\ (M_i)_{i \in I}$: nondeterministically choose some $i \in I$, then evaluate M_i. The $\Downarrow$ relation and CK-machine transltion are extended accordingly. Allowing I to be empty requires us to say that evaluation may die at any time.

4.3. Errors

Let E be a set of errors, for example $\{\mathrm{CRASH}, \mathrm{BANG}\}$. We extend CBPV with the typing rule

$$\frac{}{\Gamma \vdash^{\mathsf{c}} \mathtt{error}\ e : \underline{B}}$$

for each $e \in E$. To evaluate $\mathtt{error}\ e$: halt and print the error message e. We extend the big-step semantics with a relation $M \lightning e$, saying that evaluation of M results in the error e, in the evident way.

4.4. Printing

Let $\mathcal{A}$ be a set of *letters*, for example $\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$. We extend CBPV with the typing rule

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{print}\ c.\, M : \underline{B}}$$

for each $c \in \mathcal{A}$. To evaluate $\mathtt{print}\ c.\, M$: firstly print c and then evaluate M. We can present the operational semantics for this language in various ways.

— The inductively defined "big-step" relation has the form $M \Downarrow m, T$, where $M : \underline{B}$ is a closed computation and $m \in \mathcal{A}^*$ is a string and $T : \underline{B}$ is terminal; this means that the evaluation of M prints m and returns T.
— The inductively defined "medium-step" relation [Plotkin and Power 2001] has the form $M \downarrow S$, where $M : \underline{B}$ is a closed computation and $S : \underline{B}$ is either terminal or of the form $\mathtt{print}\ c.\, M$.
— For CK-machine semantics, note that every CK-configuration either has the form $T, \mathtt{nil}$, where T is terminal, or has the form $\mathtt{print}\ c.\, M, K$, or has a successor.

4.5. Interactive input

Let $s = (K_r)_{r \in R}$ be a *signature*, i.e. an indexed family of sets. For a set A, an *s-tree* on A is a tree in which each leaf is labelled with some $a \in A$, and each internal node is labelled with some $r \in R$ and has a K_r-indexed set of children.

We extend CBPV with the typing rule

$$\frac{(\Gamma \vdash^{\mathsf{c}} M_k : \underline{B})_{k \in K_r}}{\Gamma \vdash \mathtt{in}_r\ (M_k)_{k \in K_r} : \underline{B}}$$

for each $r \in R$. To evaluate $\mathtt{in}_r\ (M_k)_{k \in K_r}$: firstly print r and pause, and if the user then inputs $k \in K_r$, evaluate M_k. The medium-step and CK-machine semantics are defined as in Section 4.4. The big-step semantics takes the form $M \Downarrow t$, where $M : \underline{B}$ is a closed computation and t is a well-founded s-tree on the set of terminals of type $\underline{B}$.

Note that interactive input generalizes both errors and printing, as the set E of errors gives the signature $(\emptyset)_{e \in E}$, and the set $\mathcal{A}$ of letters gives the signature $(1)_{c \in \mathcal{A}}$.

4.6. Dynamically generated store

To keep things simple, let's suppose that each memory cell stores a boolean. We extend CBPV with the value type $\mathtt{loc}$ for the cell locations, and the following typing rules.

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{loc} \quad \Gamma, \mathtt{x} : \mathtt{bool} \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{read}\ V\ \mathtt{as}\ \mathtt{x}.\ M : \underline{B}} \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{loc} \quad \Gamma \vdash^{\mathsf{v}} W : \mathtt{bool} \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} V := W.\ M : \underline{B}}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{bool} \quad \Gamma, \mathtt{x} : \mathtt{loc} \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{new}\ \mathtt{x} := V;\ M}$$

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{loc} \quad \Gamma \vdash^{\mathsf{v}} V' : \mathtt{loc} \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \quad \Gamma \vdash^{\mathsf{c}} M' : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{if}\ V = V'\ \mathtt{then}\ M\ \mathtt{else}\ M' : \underline{B}}$$

A *world* is a natural number, indicating how many cells exist at a given time. The poset of worlds is written $\mathcal{W}$. (Here $\mathcal{W}$ is $\mathbb{N}$, since we allow just one sort of cell.) The *empty world* is 0. We have judgements $w|\Gamma \vdash^{\mathsf{v}} V : A$ and $w|\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ and $w|\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ for terms at world w—the case $w = 0$ giving the ordinary judgements—and we have the additional rule

$$\frac{}{w|\Gamma \vdash^{\mathsf{v}} \mathtt{l}_i : \mathtt{loc}}\ 0 \leqslant i < w$$

For a world w, a *state at* w is a list of w booleans. The big-step semantics takes the form $w, s, M \Downarrow w', s', T$, where

— w is a world
— s is a state at w
— $M : \underline{B}$ is a closed computation at w
— w' is a world such that $w' \geqslant w$
— s' is a state at w'
— $T : \underline{B}$ is a terminal at w'.

A *WSC-configuration* $w, s, M : \underline{B}$ consists of a world w, and a state s and closed computation $M : \underline{B}$ at w. Likewise a *WSCK-configuration* $w, s, M, K : \underline{C}$ consists of a world w, and a state s and closed computation $M : \underline{B}$ and closed stack $K : \underline{B} \Longrightarrow \underline{C}$ at w. For a world w, a WSC-configuration $x, s, M : \underline{B}$ or WSCK-configuration $x, s, M, K : \underline{C}$ is said to be *at* w when $w \leqslant x$.

4.7. Global store

The global store extension of CBPV is formulated the same way, except that we fix the world w and disallow $\mathtt{new}$. The big-step semantics takes the form $s, M \Downarrow s', T$, and we simply speak of an *SC-configuration* $s, M : \underline{B}$ and an *SCK-configuration* $s, M, K : \underline{C}$.

4.8. Control effects

We extend CBPV with an instruction $\mathtt{letstk}\ \alpha$ meaning "let α be the current stack", and an instruction $\mathtt{changestk}\ \alpha$ meaning "change the current stack to α". We use $\alpha, \beta, \ldots$ for stack variables, which are distinct from ordinary variables.

A *stack variable declaration* $\alpha : \underline{B}$ says (informally) that α is a stack from $\underline{B}$ to some unspecified type. A *stack context* Δ is a list of these, with no stack variable declared more than once. The value judgement takes the form $\Gamma \vdash^{\mathsf{v}} V : A \mid \Delta$ and the computation judgement $\Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta$, with the following extra rules.

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta, \alpha : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{letstk}\ \alpha.\ M \mid \Delta} \qquad \frac{(\alpha : \underline{B}) \in \Delta \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta}{\Gamma \vdash^{\mathsf{c}} \mathtt{changestk}\ \alpha.\ M : \underline{B}' \mid \Delta}$$

To describe execution with top-level type $\underline{C}$, we need the judgements $\Gamma \vdash^{\mathsf{v}} V : A\ [\underline{C}]\ \Delta$ for *execution values*, and $\Gamma \vdash^{\mathsf{c}} M : \underline{B}\ [\underline{C}]\ \Delta$ for *execution computations*, and $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta$ for stacks. The extra rules for these are as follows.

$$\frac{(\alpha : \underline{B}) \in \Delta}{\Gamma \vdash^{\mathsf{k}} \alpha : \underline{B} \Longrightarrow \underline{C} \mid \Delta}$$

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta, \alpha : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{letstk}\ \alpha.\ M \mid \Delta} \qquad \frac{\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B}\ [\underline{C}]\ \Delta}{\Gamma \vdash^{\mathsf{c}} \mathtt{changestk}\ K.\ M : \underline{B}'\ [\underline{C}]\ \Delta}$$

Clearly $\Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta$ implies $\Gamma \vdash^{\mathsf{c}} M : \underline{B}\ [\underline{C}]\ \Delta$, and likewise for values.

Say that a ***CK-configuration*** $M, K : \underline{C}$ consists of a closed execution computation $M : \underline{B}\ [\underline{C}]$ and a closed stack $K : \underline{B} \Longrightarrow \underline{C}$. So any closed computation $M : \underline{C}$ gives a CK-configuration $M, \mathtt{nil} : \underline{C}$. The extra transition rules are as follows.

$$\begin{array}{llll} \mathtt{letstk}\ \alpha.\ M & K & \rightsquigarrow & M[K/\alpha] \qquad K \\ \mathtt{changestk}\ K.\ M & L & \rightsquigarrow & M \qquad K \end{array}$$

5. DENOTATIONAL SEMANTICS

Having presented many effectful extensions of CBPV, let us give (in outline) a denotational semantics for each of them. These models satisfy the equational laws, including the pure continuation law.

5.1. Recursion

For CBPV extended with recursion, the *observational preorder* relates two terms $\Gamma \vdash M, N : A$ when any program that returns n continues to do so when M is replaced by N. This motivates the use of posets for denotational semantics.

A *cpo* A is a poset in which every ω-chain $a_0 \leqslant a_1 \leqslant \cdots$ has a supremum $\bigvee_{n \in \mathbb{N}} a_n$. A function between cpos $A \to B$ is *continuous* when it preserves these suprema (and hence is monotone). The set of continuous functions from A to B, ordered pointwise, is written $A \to B$. A *cppo* is a cpo with bottom element, written $\bot$. Any cpo A can be "lifted" to form a cppo

$$A_\bot \stackrel{\text{def}}{=} \{\mathsf{up}(a) \mid a \in A\} \cup \{\bot\}$$

ordered as A with an additional bottom element.

Now for the denotational semantics. A value type A denotes a cpo $[\![A]\!]$, thought of as a semantic domain for values of type A. A computation type $\underline{B}$ denotes a cppo $[\![\underline{B}]\!]$, thought of as a semantic domain for computations of type $\underline{B}$. Specifically:

$$\begin{aligned} [\![U\underline{B}]\!] &\stackrel{\text{def}}{=} [\![\underline{B}]\!] \\ [\![FA]\!] &\stackrel{\text{def}}{=} [\![A]\!]_\bot \\ [\![A \to \underline{B}]\!] &\stackrel{\text{def}}{=} [\![A]\!] \to [\![\underline{B}]\!] \end{aligned}$$

and so forth. Recursive types—of both kinds—are interpreted using a recipe that was given in [Smyth and Plotkin 1982]; this leads, for example, to the type $\mathtt{rec}\ \mathtt{X}.\ \mathtt{bool} \times \mathtt{X}$ denoting the empty cpo.

A typing context Γ denotes the cpo $\prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]$, whose elements are semantic environments. Then:

— A value $\Gamma \vdash^{\mathsf{v}} V : A$ denotes a continuous function $[\![\Gamma]\!] \to [\![A]\!]$.
— A computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a continuous function $[\![\Gamma]\!] \to [\![\underline{B}]\!]$.

— A stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ denotes a function $[\![K]\!] : [\![\Gamma]\!] \times [\![\underline{B}]\!] \to [\![\underline{C}]\!]$ that is strict (i.e. preserves $\bot$) in its second argument.

The strictness property expresses the fact that if $M : \underline{B}$ diverges then so does the configuration M, K.

Here are some example semantic equations:

$$
\begin{aligned}
[\![\mathtt{thunk}\ M]\!](\rho) &\stackrel{\text{def}}{=} [\![M]\!](\rho) \\
[\![\mathtt{force}\ V]\!](\rho) &\stackrel{\text{def}}{=} [\![V]\!](\rho) \\
[\![\mathtt{return}\ V]\!](\rho) &\stackrel{\text{def}}{=} \mathsf{up}([\![V]\!](\rho)) \\
[\![M\ \mathtt{to}\ \mathtt{x}.\ N]\!](\rho) &\stackrel{\text{def}}{=} \begin{cases} [\![N]\!](\rho[\mathtt{x} \mapsto a]) & \text{if } [\![M]\!](\rho) = \mathsf{up}(a) \\ \bot & \text{if } [\![M]\!](\rho) = \bot \end{cases} \\
[\![\mathtt{rec}\ \mathtt{x}.\ M]\!] &\stackrel{\text{def}}{=} \bigvee_{n \in \mathbb{N}} f^n(\bot) \quad \text{where } f : b \mapsto [\![M]\!](\rho[\mathtt{x} \mapsto b])
\end{aligned}
$$

All the CBPV equational laws are satisfied, and the pure continuation law follows from strictness.

A key result is the agreement of the denotational and operational semantics, which is called "computational adequacy". Writing ε for the empty environment, we have that $M \Downarrow T$ implies $[\![M]\!]\varepsilon = [\![T]\!]\varepsilon$, and that $M \Uparrow$ implies $[\![M]\!]\varepsilon = \bot$. Furthermore, if we say that a CK-configuration $M, K : \underline{C}$ denotes the element $[\![K]\!](\varepsilon, [\![M]\!]\varepsilon) \in [\![\underline{C}]\!]$, then $M, K \rightsquigarrow N, L$ implies $[\![M, K]\!] = [\![N, L]\!]$, and $M, K \rightsquigarrow^{\omega}$ implies $[\![M, K]\!] = \bot$.

5.2. Errors, printing, interactive input

To give denotational semantics for these effects, we use the following kinds of structured set.

— For errors: let E be a set. An *E-pointed set* consists of a set X (the *carrier*) and a distinguished element p_e for each $e \in E$. On a set Y, the *free* E-pointed set is $Y + E$ with the evident structure.
— For printing: let $\mathcal{A}$ be a set. An *$\mathcal{A}$-set* consists of a set X (the *carrier*) and a map $p_c : X \to X$ for each $c \in \mathcal{A}$. For $m = (c_0, \ldots, c_{n-1}) \in \mathcal{A}^*$ and $x \in X$, we define $m * * x \in X$ to be $c_0 * \cdots * c_{n-1} * x$ right-associated. On a set Y, the *free* $\mathcal{A}$-pointed set is $\mathcal{A}^* \times Y$ with the evident structure.
— For interactive input: let $s = (K_r)_{r\ in R}$ be a signature. An *s-algebra* consists of a set X (the *carrier*) and a map $p_r : X^{K_r} \to X$ for each $r \in R$. On a set Y, the *free* s-algebra is the set of well-founded s-trees on Y with the evident structure.

Let us describe the model for errors, as the others are similar. Each value type A denotes a set $[\![A]\!]$, thought of as a semantic domain for values $V : A$, and each computation type $\underline{B}$ denotes an E-pointed set $[\![\underline{B}]\!]$, thought of as a semantic domain for computations $M : \underline{B}$. In particular:

— $[\![U\underline{B}]\!]$ is the carrier of $[\![\underline{B}]\!]$
— $[\![FA]\!]$ is the free E-pointed set on $[\![A]\!]$
— $A \to \underline{B}$ is $\prod_{a \in [\![A]\!]} [\![\underline{B}]\!]$.

A typing context Γ denotes the set $\prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]$, whose elements are semantic environments. Then:

— A value $\Gamma \vdash V : B$ denotes a function $[\![\Gamma]\!] \to [\![B]\!]$.
— A computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a function $[\![\Gamma]\!] \to [\![\underline{B}]\!]$.

— A stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ denotes a function from $[\![\Gamma]\!] \times [\![\underline{B}]\!]$ to $[\![\underline{C}]\!]$ that is homomorphic (i.e. p_e-preserving for all $e \in E$) in its second argument.

The requirement for a stack K to denote a homomorphism expresses the fact that if $M : \underline{B}$ raises an error e, then so does the configuration M, K. It is crucial that errors cannot be caught by K.

The semantic equations are omitted, but note the following clauses:

$$[\![\mathtt{thunk}\ M]\!] \ = \ [\![M]\!]$$
$$[\![\mathtt{force}\ V]\!] \ = \ [\![V]\!]$$

Again, we have adequacy: if $M \Downarrow T$ then $[\![M]\!] = [\![T]\!]$, and if $M \not\downarrow e$ then $[\![M]\!] = p_e$. These are straightforwardly proved by induction. For printing, we have that $M \Downarrow m, T$ implies $[\![M]\!] = m * *[\![T]\!]$, and similarly for interactive input.

5.3. Nondeterminism

Let us give in outline a denotational semantics for CBPV extended with nondeterminism. A value type A denotes a set, thought of a semantic domain for values $V : A$. A computation type $\underline{B}$ denotes a set, thought of a semantic domain for *possible behaviours* of computations $M : \underline{B}$. Specifically:

$[\![FA]\!]$	$\stackrel{\text{def}}{=} [\![A]\!]$	since a computation : FA may return a value : A.
$[\![A \to \underline{B}]\!]$	$\stackrel{\text{def}}{=} [\![A]\!] \times [\![\underline{B}]\!]$	since a computation : $A \to \underline{B}$ may pop an value : A and then behave as a computation : $\underline{B}$.
$[\![\prod_{i \in I} \underline{B}_i]\!]$	$\stackrel{\text{def}}{=} \sum_{i \in I} [\![\underline{B}_i]\!]$	since a computation : $\prod_{i \in I}$ may pop $i \in I$ and then behave as a computation : $\underline{B}_i$.
$[\![U\underline{B}]\!]$	$\stackrel{\text{def}}{=} \mathcal{P}[\![\underline{B}]\!]$	since a value : $U\underline{B}$, when forced, can exhibit a range of possible behaviours : $\underline{B}$.

A typing context Γ denotes the set $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]$. Then:

— A value $\Gamma \vdash^{\mathsf{v}} V : A$ denotes a function from $[\![\Gamma]\!]$ to $[\![A]\!]$.
— A computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a relation from $[\![\Gamma]\!]$ to $[\![\underline{B}]\!]$. This is thought of as relating ρ to b when b is a possible behaviour of $M[\rho]$.
— A stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ denotes a relation from $[\![\Gamma]\!] \times [\![\underline{B}]\!]$ to $[\![\underline{C}]\!]$. This is thought of as relating (ρ, b) to c when a CK-configuration consisting of a computation exhibiting behaviour b and the stack $K[\rho]$ may exhibit behaviour c.

The semantic equations are omitted, but note the following clause:

$$[\![\mathtt{return}\ V]\!] \ = \ [\![V]\!]$$

using the fact that a function is a special kind of relation. Adequacy takes the form

$$[\![M]\!] \ = \ \bigcup_{M \Downarrow T} [\![T]\!]$$

5.4. Global store

Let us say that we have one cell `l` of type `bool` and one `l'` of type `nat`. So the set of states is $S \stackrel{\text{def}}{=} \mathbb{B} \times \mathbb{N}$. Our denotational semantics for CBPV extended with these global cells is arranged as follows.

— A value type A denotes a set, thought of as a semantic domain for values $V : A$

— A computation type $\underline{B}$ denotes a set, thought of as a semantic domain for SC-configurations $s, M : \underline{B}$.

Here is the semantics of types:

$$\begin{array}{rcll}
[\![FA]\!] & \stackrel{\text{def}}{=} & S \times [\![A]\!] & \text{since an SC-configuration} : FA \text{ evaluates to } s, \texttt{return}\ V \\
 & & & \quad \text{for a state } s \text{ and closed value } V : A. \\
[\![A \to \underline{B}]\!] & \stackrel{\text{def}}{=} & [\![A]\!] \to [\![\underline{B}]\!] & \text{since a configuration} : A \to \underline{B} \text{, pops a value} : A, \\
 & & & \quad \text{and then accordingly behaves as an SC-configuration} : \underline{B}. \\
[\![\prod_{i \in I} \underline{B}_i]\!] & \stackrel{\text{def}}{=} & \prod_{i \in I} [\![\underline{B}_i]\!] & \text{since an SC-configuration} : \prod_{i \in I} \underline{B}_i \text{ pops } i \in I \\
 & & & \quad \text{and then accordingly behaves as an SC-configuration} : \underline{B}_i. \\
[\![U\underline{B}]\!] & \stackrel{\text{def}}{=} & S \to [\![\underline{B}]\!] & \text{since a value } V : U\underline{B}, \text{ when forced in any state } s, \\
 & & & \quad \text{gives an SC-configuration } s, \texttt{force}\ V.
\end{array}$$

A typing context Γ denotes the set $[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]$. Then:

— A value $\Gamma \vdash^{\mathsf{v}} V : A$ denotes a function $[\![\Gamma]\!] \to [\![A]\!]$.
— A computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a function from $S \times [\![\Gamma]\!]$ to $[\![\underline{B}]\!]$, thought of as sending (s, ρ) to the SC-configuration $s, M[\rho]$.
— A stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ denotes a function from $[\![\Gamma]\!] \times [\![\underline{B}]\!]$ to $[\![\underline{C}]\!]$, thought of as sending an environment ρ and SC-configuration s, M to the SC-configuration $s, M \bullet K[\rho]$.

Again, the semantic equations are easy to write, and validate all the CBPV equational laws, as well as the pure continuation law. The adequacy theorem says that $s, M \Downarrow s', T$ implies $[\![M]\!](s, \varepsilon) = [\![T]\!](s, \varepsilon)$, and is proved by induction.

5.5. Dynamically generated, globally visible cells

Now we develop the semantics of store to allow dynamic generation of cells. It is often considered desirable to validate equations such as

$$\begin{array}{rcll}
M & = & \texttt{new x:=}V.\ M & \text{(where x is fresh)} \\
\texttt{new x:=}V.\ \texttt{new y:=}W.\ M & = & \texttt{new y:=}W.\ \texttt{new x:=}V.\ M &
\end{array}$$

But the model described here is a simple one in which generated cells are "globally visible" and so these equations are not validated.

The semantics of types is arranged as follows.

— A value type A denotes a functor from $\mathcal{W}$ to Set. Spelling this out, for each world w, we provide a set $[\![A]\!]w$, thought of as a semantic domain for values $: A$ at w. Furthermore, for any worlds $w \leqslant x$, we provide a function $[\![A]\!]^w_x$ from $[\![A]\!]w$ to $[\![A]\!]x$, since any value at w is also one at x,. Finally, the diagram $[\![A]\!]w \underset{[\![A]\!]^w_w}{\overset{\mathsf{id}}{\rightrightarrows}} [\![A]\!]w$ commutes for each world w, and the diagram $\begin{array}{ccc} [\![A]\!]w & \xrightarrow{[\![A]\!]^w_x} & [\![A]\!]x \\ & \searrow^{[\![A]\!]^w_y} & \downarrow{\scriptstyle [\![A]\!]^x_y} \\ & & [\![A]\!]y \end{array}$ commutes for all worlds $w \leqslant x \leqslant y$.

— A computation type $\underline{B}$ denotes a a functor from $\mathcal{W}^{\mathsf{op}}$ to Set. Spelling this out, for each world w, we provide a set $[\![\underline{B}]\!]w$, thought of as a semantic domain for WSC-configurations $: \underline{B}$ at w. For worlds $w \leqslant x$, we provide a function $[\![\underline{B}]\!]^w_x$ from $[\![\underline{B}]\!]x$

to $[\![\underline{B}]\!]w$, since any WSC-configuration at x is also one at w. Finally, the diagram $[\![\underline{B}]\!]w \overset{\mathsf{id}}{\underset{[\![\underline{B}]\!]^w_w}{\rightrightarrows}} [\![\underline{B}]\!]w$ commutes for each world w, and the diagram

$$\begin{array}{ccc} [\![\underline{B}]\!]y & \xrightarrow{[\![\underline{B}]\!]^x_y} & [\![\underline{B}]\!]x \\ & \searrow^{[\![\underline{B}]\!]^w_y} & \downarrow{\scriptstyle [\![\underline{B}]\!]^w_x} \\ & & [\![\underline{B}]\!]w \end{array}$$

commutes for all worlds $w \leqslant x \leqslant y$.

For a world w, write $Sw \stackrel{\text{def}}{=} \mathbb{B}^w$ for the set of states at w. The semantics of types is as follows.

$$\begin{array}{rcll} [\![FA]\!]w & \stackrel{\text{def}}{=} & \sum_{x \geqslant w} (Sx \times [\![A]\!]x) & \text{since an WSC-configuration} : FA \text{ at } w \\ & & & \text{evaluates to } x, s, \mathtt{return}\ V \text{ for a world } x \geqslant w \\ & & & \text{and state } s \text{ and closed value } V : A \text{ at } x. \\ [\![A \to \underline{B}]\!]w & \stackrel{\text{def}}{=} & [\![A]\!]w \to [\![\underline{B}]\!]w & \text{since a WSC-configuration} : A \to \underline{B} \\ & & & \text{pops a value} : A \text{ and behaves as an WSC-configuration} : \underline{B}. \\ [\![\textstyle\prod_{i \in I} \underline{B}_i]\!]w & \stackrel{\text{def}}{=} & \prod_{i \in I} [\![\underline{B}_i]\!]w & \text{since a WSC-configuration} : \prod_{i \in I} \underline{B}_i \\ & & & \text{pops } i \in I \text{ and behaves as a WSC-configuration} : \underline{B}_i. \\ [\![U\underline{B}]\!]w & \stackrel{\text{def}}{=} & \prod_{x \geqslant w} (Sx \to [\![\underline{B}]\!]x) & \text{since a value } V : U\underline{B} \text{ at } w, \\ & & & \text{when forced in any world } x \geqslant w \text{ and state } s \text{ at } x, \\ & & & \text{gives a WSC-configuration } x, s, \mathtt{force}\ V \text{ at } x. \end{array}$$

A typing context Γ denotes the functor given at the world w by the set $[\![\Gamma]\!]w \stackrel{\text{def}}{=} \prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]w$, an element of which is a semantic environment. Then:

— A value $w|\Gamma \vdash^{\mathsf{v}} V : A$ denotes a family of functions $([\![V]\!]x : [\![\Gamma]\!]x \to [\![A]\!]x)_{x \geqslant w}$ that is "natural" in the following sense: for any $w \leqslant x \leqslant y$, the square of functions

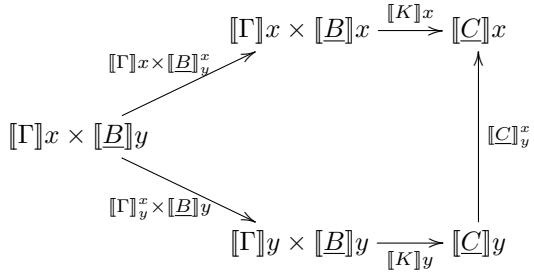
$$\begin{array}{ccc} [\![\Gamma]\!]x & \xrightarrow{[\![V]\!]x} & [\![A]\!]x \\ {\scriptstyle [\![\Gamma]\!]^x_y}\downarrow & & \downarrow{\scriptstyle [\![A]\!]^x_y} \\ [\![\Gamma]\!]y & \xrightarrow[[\![V]\!]y]{} & [\![A]\!]y \end{array}$$

commutes. (Informal explanation: for any Γ-environment ρ at x, each path leads to the same value $V[\rho] : A$ at y.)

— A computation $w|\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a family of functions $([\![M]\!]x : Sx \times [\![\Gamma]\!]x \to [\![\underline{B}]\!]x)_{x \geqslant w}$, where $[\![M]\!]w : (s, \rho) \mapsto b$ means that M, in the world w and state s and environment ρ, has behaviour b.

— A stack $w|\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ denotes a family of functions $([\![K]\!]x : [\![\Gamma]\!]x \times [\![\underline{B}]\!]x \to [\![\underline{C}]\!]x)_{x \geqslant w}$ that is "dinatural" in the following sense: for any $w \leqslant x \leqslant y$, the pentagon

of functions



commutes. (Informal explanation: for any Γ-environment ρ at x and WSC-configuration $y', s, M : \underline{B}$ at y, each path leads to the same WSC-configuration $y', s, M \bullet K[\rho] : \underline{C}$ at x.)

Again, the semantic equations are easy to write, and validate all the CBPV equational laws, as well as the pure continuation law. The adequacy theorem says that $w, s, M \Downarrow x, s', T$ at type $\underline{B}$ implies $[\![M]\!]w(s, \varepsilon) = ([\![\underline{B}]\!]^w_x)([\![T]\!]x(s', \varepsilon))$, and is proved by induction.

5.6. Control

The denotational semantics for CBPV extended with control operators is rather subtle, and I shall explain it using some imaginary notions. For any computation type $\underline{B}$, imagine a notion of *idealized stack* K from $\underline{B}$ that has no top-level type; this excludes `nil`. Likewise, imagine a notion of an *idealized CK-configuration*, which consists of a closed computation $M : \underline{B}$ and an idealized stack K from $\underline{B}$. Although idealized stacks and CK-configurations do not actually exist, these notions are conceptually helpful.

Let us now fix a set R, thought of as a semantic domain for idealized CK-configurations. The semantics of types is arranged as follows.

— A value type A denotes a set, thought of as a semantic domain for values of type A.
— A computation type $\underline{B}$ denotes a set, thought of as a semantic domain for idealized stacks from $\underline{B}$.

The clauses are as follows.

$$[\![FA]\!] \stackrel{\text{def}}{=} [\![A]\!] \to R \quad \text{since an idealized stack } K \text{ from } FA \text{ converts a value } V : A \text{ into an idealized CK-configuration } \texttt{return}\ V,\ K.$$

$$[\![A \to \underline{B}]\!] \stackrel{\text{def}}{=} [\![A]\!] \times [\![\underline{B}]\!] \quad \text{since an idealized stack } V :: K \text{ from } A \to \underline{B} \text{ consists of a value } V : A \text{ and an idealized stack } K \text{ from } \underline{B}.$$

$$[\![\textstyle\prod_{i\in I} \underline{B}_i]\!] \stackrel{\text{def}}{=} \textstyle\sum_{i\in I} [\![\underline{B}_i]\!] \quad \text{since an idealized stack } \hat{\imath} :: K \text{ from } \prod_{i\in I} \underline{B}_i \text{ consists of } \hat{\imath} \in I \text{ and an idealized stack } K \text{ from } \underline{B}_i.$$

$$[\![U\underline{B}]\!] \stackrel{\text{def}}{=} [\![\underline{B}]\!] \to R \quad \text{since a value } V : U\underline{B}, \text{ when forced alongside any idealized stack } K \text{ from } \underline{B}, \text{ gives an idealized CK-configuration } \texttt{force}\ V,\ K.$$

An ordinary context Γ denotes the set $\prod_{(\mathtt{x}:A)\in\Gamma} [\![A]\!]$, an element of which is a semantic environment. A stack context Δ denotes the set $[\![\Delta]\!] \stackrel{\text{def}}{=} \prod_{(\alpha:\underline{B})\in\Gamma} [\![\underline{B}]\!]$, an element of which can be called an "idealized stack environment". Then:

— A value $\Gamma \vdash^{\mathsf{v}} V : A \mid \Delta$ denotes a function from $[\![\Gamma]\!] \times [\![\Delta]\!]$ to $[\![A]\!]$.
— A computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a function from $[\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\underline{B}]\!]$ to R. Intuitively, given environments $\rho \in [\![\Gamma]\!]$ and $\nu \in [\![\Delta]\!]$, and an idealized stack K from $\underline{B}$, we obtain an idealized CK-configuration $M[\rho, \nu],\ K$.

— A stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta$ denotes a function from $[\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\underline{C}]\!]$ to $[\![\underline{B}]\!]$. Intuitively, given environments $\rho \in [\![\Gamma]\!]$ and $\nu \in [\![\Delta]\!]$, and an idealized stack L from $\underline{C}$, we obtain an idealized stack $K[\rho, \nu] +\!\!+ L$ from $\underline{B}$.
— An execution value $\Gamma \vdash^{\mathsf{v}} V : A\ [\underline{C}]\ \Delta$ denotes a function from $[\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\underline{C}]\!]$ to $[\![A]\!]$. Intuitively, V reduces to an ordinary value when we replace `nil` by an idealized stack from $\underline{C}$.
— Likewise, an execution computation $\Gamma \vdash^{\mathsf{c}} M\ [\underline{C}]\ \underline{B}$ denotes a function from $[\![\Gamma]\!] \times [\![\Delta]\!] \times [\![\underline{C}]\!] \times [\![\underline{B}]\!]$ to R.

Here are some key semantic equations:

$$\begin{aligned}
[\![\texttt{return}\ V]\!](\rho, \nu, k) &= k([\![V]\!](\rho, \nu)) \\
[\![\texttt{force}\ V]\!](\rho, \nu, k) &= ([\![V]\!](\rho, \nu))k \\
[\![\texttt{letstk}\ \alpha.\ M]\!](\rho, \nu, k) &= [\![M]\!](\rho, \nu[\alpha \mapsto k], k) \\
[\![\texttt{changestk}\ \alpha.\ M]\!](\rho, \nu, k) &= [\![M]\!](\rho, \nu, \nu(\alpha)) \\
[\![V :: K]\!](\rho, \nu, k) &= \langle [\![V]\!](\rho, \nu), [\![K]\!](\rho, \nu, k) \rangle
\end{aligned}$$

Lastly, a CK-configuration $M, K : \underline{C}$ denotes a function from $[\![\underline{C}]\!]$ to R, given by

$$[\![M, K]\!](c) \stackrel{\text{def}}{=} [\![M]\!](\varepsilon, \varepsilon, c, [\![K]\!](\varepsilon, , \varepsilon, c))$$

and transition preserves denotation. If we take R to be $\mathbb{N}$, then, for any program $M : F\,\texttt{nat}$ that evaluates to n, we can recover n from $[\![M]\!]$ as follows:

$$\begin{aligned}
[\![M]\!](\varepsilon, \varepsilon, \mathsf{id}_{\mathbb{N}}) &= [\![M,\ \texttt{nil}]\!](\mathsf{id}_{\mathbb{N}}) \\
&= [\![\texttt{return}\ n,\ \texttt{nil}]\!](\mathsf{id}_{\mathbb{N}}) \\
&= n
\end{aligned}$$

6. CALL-BY-VALUE AND CALL-BY-NAME

So far we have considered CBPV as a whole; let us now turn to the CBV and CBN subsystems. The translations into CBPV are outlined in Figure 6. A more detailed analysis can be given using a special syntax called *jumbo λ-calculus* [Levy 2006b]. Furthermore, the translation from CBV can be presented in two stages; the intermediate calculus is called *fine-grain call-by-value* [Levy et al. 2003].

Since CBV allows an application MN to be evaluated either operator-first or operand-first, we have to give two translations. The operand-first translation can be "optimized" in the CBPV theory to the following:

$$N\ \texttt{to x.}\ (M\ \texttt{to f.}\ \texttt{force f})\texttt{x}$$

Using ‘ notation this becomes

$$N\ \texttt{to x.}\ \texttt{x`}(M\ \texttt{to f.}\ \texttt{force f})$$

which means that we first evaluate N and push the result. Interestingly, the operand-first version of the (call-by-value) ML language has a compiler called ZINC that does precisely this [Leroy 1991], so we have a semantic explanation of what seemed to be a mere compiler trick.[3]

Via the given translations, each CBPV semantics gives rise to a semantics of CBV and CBN, and in many cases this is standard. For example, in the CBN model of recursion, each type denotes a cppo, and we have

$$\begin{aligned}
[\![\underline{A} + \underline{B}]\!] &= ([\![\underline{A}]\!] + [\![\underline{B}]\!])_\perp \\
[\![\underline{A} \to \underline{B}]\!] &= [\![\underline{A}]\!] \to [\![\underline{B}]\!]
\end{aligned}$$

[3] I thank Hayo Thielecke for bringing this feature of ZINC to my attention.

CBV type	CBPV value type
$A + B$	$A + B$
$A \to B$	$U(A \to FB)$
CBV term $\mathtt{x} : A, \mathtt{y} : B \vdash M : C$	**CBPV computation** $\mathtt{x} : A, \mathtt{y} : B \vdash M : FC$
$\mathtt{x}$	$\mathtt{return}\ \mathtt{x}$
$\mathtt{inl}\ M$	$M\ \mathtt{to}\ \mathtt{x}.\ \mathtt{return}\ \mathtt{inl}\ \mathtt{x}$
$\lambda \mathtt{x}.\, M$	$\mathtt{return}\ \mathtt{thunk}\ \lambda \mathtt{x}.\, M$
MN (operator-first)	$M\ \mathtt{to}\ \mathtt{f}.\ N\ \mathtt{to}\ \mathtt{x}.\ (\mathtt{force}\ \mathtt{f})\mathtt{x}$
MN (operand-first)	$N\ \mathtt{to}\ \mathtt{x}.\ M\ \mathtt{to}\ \mathtt{f}.\ (\mathtt{force}\ \mathtt{f})\mathtt{x}$
CBN type	**CBPV computation type**
$\underline{A} + \underline{B}$	$F(U\underline{A} + U\underline{B})$
$\underline{A} \to \underline{B}$	$U\underline{A} \to \underline{B}$
CBN term $\mathtt{x} : \underline{A}, \mathtt{y} : \underline{B} \vdash M : \underline{C}$	**CBPV computation** $\mathtt{x} : U\underline{A}, \mathtt{y} : U\underline{B} \vdash M : \underline{C}$
$\mathtt{x}$	$\mathtt{force}\ \mathtt{x}$
$\mathtt{inl}\ M$	$\mathtt{return}\ \mathtt{inl}\ \mathtt{thunk}\ M$
$\lambda \mathtt{x}.\, M$	$\lambda \mathtt{x}.\, M$
MN	$M(\mathtt{thunk}\ N)$

Fig. 6. Decomposition of CBV and CBN into CBPV (selected clauses)

Here are some CBV examples:

Errors	$\llbracket A \to B \rrbracket$	$= \llbracket A \rrbracket \to (\llbracket B \rrbracket + E)$
Printing	$\llbracket A \to B \rrbracket$	$= \llbracket A \rrbracket \to (\mathcal{A}^* \times \llbracket B \rrbracket)$
Nondeterminism	$\llbracket A \to B \rrbracket$	$= \mathcal{P}(\llbracket A \rrbracket \times \llbracket B \rrbracket)$
Global store	$\llbracket A \to B \rrbracket$	$= S \to (\llbracket A \rrbracket \to (S \times \llbracket B \rrbracket))$
Dynamically generated store	$\llbracket A \to B \rrbracket w$	$= \prod_{x \geqslant w} (Sx \to (\llbracket A \rrbracket x \to \sum_{y \geqslant x} (Sy \times \llbracket B \rrbracket y)))$
Control	$\llbracket A \to B \rrbracket$	$= (\llbracket A \rrbracket \times (\llbracket B \rrbracket \to R)) \to R$

These are all the same as (or isomorphic to) the model given in [Moggi 1991], except for the model of dynamically generated store, which appeared in [Levy 2002].

7. WRAPPING UP

This article has provided a brief introduction to CBPV. We have seen operational semantics in various forms including the CK-machine. We have also presented denotational models for various effects, explained using computational ideas such as configuration, behaviour and observational preorder. Lastly, we have described (in outline) how the CBV and CBN versions of the simply typed λ-calculus with effects can be seen as subsystems of CBPV. This makes CBPV a natural framework to study many aspects of λ-calculus, especially ones relating to effects.

Since this article is aimed at a general audience, it simply pulled CBPV out of a hat, without giving *a priori* justification. Sceptical readers may object to this, but (as mentoned in Section 3.1) the alternative is arduous: we would begin by carefully investigating the semantics of CBV and CBN—far more complicated than that of CBPV—and then observe empirically the CBPV structure common to all the models.

Another part of the λ-calculus and CBPV story that I have omitted is the connection to category theory. Roughly speaking, whereas a model of λ-calculus is a certain kind of category, a model of CBPV (with stacks and the pure continuation law) consists of an adjunction between a category $\mathcal{V}$ that interprets value types and values and

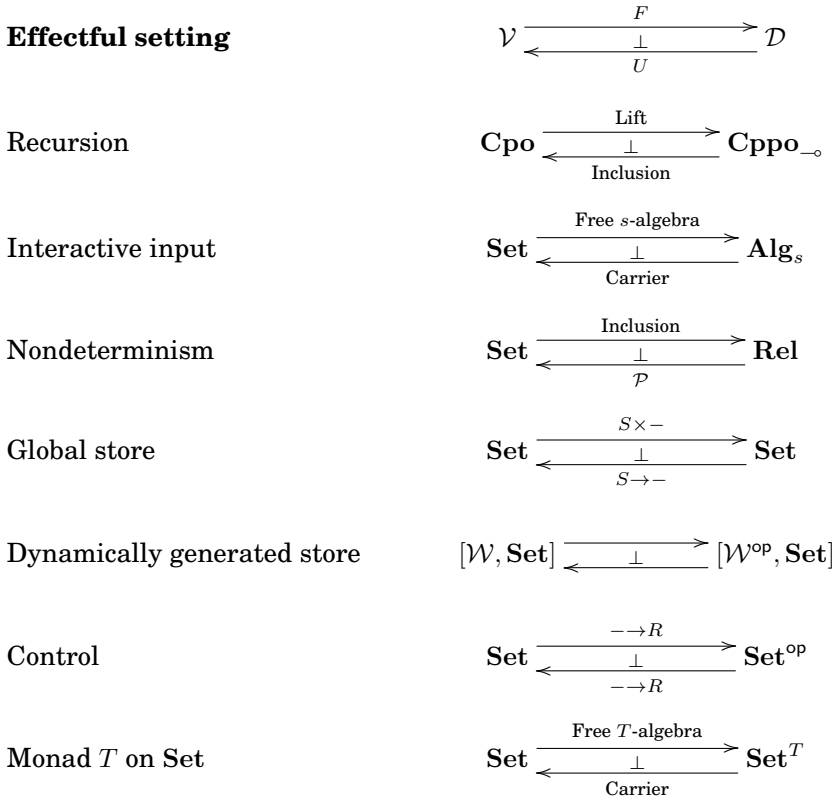


Fig. 7. Examples of adjunctions used in CBPV semantics

another category $\mathcal{D}$ that interprets computation types and stacks. The connectives U/F and the computations are interpreted using the adjunction structure. Thus, whereas Moggi [Moggi 1991] proposed to model effects using a certain kind of category $\mathcal{V}$ with a monad, CBPV provides a more refined picture. For more details, see [Levy 2005; 2004].

Some examples are shown in Figure 7. Here we write $\mathbf{Cpo}$ for the category of cpos and continuous functions; and $\mathbf{Cppo}_{\_\circ}$ for the category of pointed cpos and strict continuous functions; and $\mathbf{Alg}_s$ for the category of s-algebras and homomorphisms; and $\mathbf{Rel}$ for the category of sets and relations; and $[\mathcal{W}, \mathbf{Set}]$ for the category of functors $\mathcal{W} \to \mathbf{Set}$ and natural transformations; and $\mathbf{Set}^T$ for the category of Eilenberg-Moore T-algebras and homomorphisms. In the case of dynamically generated store, the left adjoint sends A to $(\sum_{x \geqslant w}(Sx \times Ax))_{w\in\mathcal{W}}$, and the right adjoint sends $\underline{B}$ to $(\prod_{x\geqslant w}(Sx \to Ax))_{w\in\mathcal{W}}$.

Despite the omission of these advanced topics (and others), I hope that the diverse semantics included in this article have made clear that, just as the simply typed λ-calculus with all of its β- and η-laws is a fundamental calculus of pure functional programming, so CBPV with its laws is a fundamental calculus of functional programming with effects.

REFERENCES

S. Abramsky and G. McCusker. 1998. Call-by-Value Games. In *Computer Science Logic: 11th International Workshop Proceedings (LNCS)*, M. Nielsen and W. Thomas (Eds.). Springer, 1–17.

M. Felleisen and D. Friedman. 1986. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, M. Wirsing (Ed.). North-Holland, 193–217.

A. Filinski. 1996. *Controlling Effects*. Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

K. Honda and N. Yoshida. 1997. Game Theoretic Analysis of Call-by-Value Computation. In *Automata, Languages and Programming, 24th International Colloquium (LNCS)*, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela (Eds.), Vol. 1256. Springer, Bologna, Italy, 225–236.

J. M. E. Hyland and C.-H. L. Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Information and Computation* 163, 2 (2000), 285–408.

X. Leroy. 1991. *The ZINC experiment: an economical implementation of the ML language*. Technical Report 117. INRIA.

P. B. Levy. 1999. Call-By-Push-Value: a Subsuming Paradigm (Extended Abstract). In *Proceedings, Typed λ-Calculi and Applications (LNCS)*, J-Y Girard (Ed.), Vol. 1581. 228–242.

P. B. Levy. 2001. *Call-by-push-value*. Ph.D. Dissertation. Queen Mary, University of London.

P. B. Levy. 2002. Possible World Semantics for General Storage in Call-By-Value. In *Proceedings, 16th Annual Conference of the European Assocation for Computer Science Logic (CSL) (LNCS)*, J. Bradfield (Ed.), Vol. 2471. Springer, 232–246.

P. B. Levy. 2004. *Call-By-Push-Value. A Functional/Imperative Synthesis*. Springer.

P. B. Levy. 2005. Adjunction Models For Call-By-Push-Value With Stacks. *Theory and Applications of Categories* 14 (2005), 75–110.

P. B. Levy. 2006a. Call-By-Push-Value: Decomposing Call-By-Value And Call-By-Name. *Higher-Order and Symbolic Computation* 19, 4 (2006), 377–414.

P. B. Levy. 2006b. Jumbo λ-calculus. In *Proc. , 33rd International Colloquium on Automata, Languages and Programming (LNCS)*, Vol. 4052. Springer, 444–455.

P. B. Levy, A J Power, and H Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185 (2003), 182–210.

D. B. MacQueen, P. Wadler, and W. Taha. 1998. How to add laziness to a strict language without even being odd. In *Proceedings of the 1998 ACM SIGPLAN Workshop on Standard ML*. Association for Computing Machinery, Baltimore, MD, 24–30.

G. McCusker. 2000. Games and Full Abstraction for FPC. *Information and Computation* 160, 1-2 (2000), 1–61.

E Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93 (1991), 55–92.

H. Nickau. 1996. *Hereditarily Sequential Functionals: A Game-Theoretic Approach to Sequentiality*. Shaker-Verlag. Dissertation, Universität Gesamthochschule Siegen.

G. D. Plotkin and J. Power. 2001. Adequacy for Algebraic Effects. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '01)*. Springer-Verlag, Berlin, Heidelberg, 1–24.

M. Smyth and G. D. Plotkin. 1982. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing* 11 (1982), 761–783.