A tutorial on call-by-push-value

Paul Blain Levy

University of Birmingham

December 19, 2007
1 Typed λ-calculus

2 Typed λ-calculus: denotational semantics

3 Call-by-push-value

4 Stacks

5 State

6 Control
Typed $\lambda$-calculus

We consider typed $\lambda$-calculus with boolean, function and sum types.

**Types**

\[ A ::= \text{bool} \mid A + A \mid A \rightarrow A \]

**Typing judgement** $\Gamma \vdash M : B$

**Terms**

\[ M ::= x \mid \text{let } M \text{ be } x. \ M \\
\quad \mid \text{true} \mid \text{false} \mid \text{pm } M \text{ as } \{\text{true. } M, \text{false. } M\} \\
\quad \mid \text{inl } M \mid \text{inr } M \mid \text{pm } M \text{ as } \{\text{inl } x.M, \text{inr } x.M\} \\
\quad \mid \lambda x.M \mid MM \]
Equational Laws

We consider the equational theory generated by the $\beta\eta$-laws.

$\eta$-law for $A \rightarrow B$

Any term $\Gamma \vdash M : A \rightarrow B$ can be expanded as

$$\lambda x. M x$$

Anything of function type is a $\lambda$-abstraction.

$\eta$-law for bool

Any term $\Gamma, z : \text{bool} \vdash M : B$ can be expanded as

$$\text{pm } z \text{ as } \{ \text{true. } M[\text{true}/z], \text{false. } M[\text{false}, z] \}$$

Anything of boolean type is a boolean.

The $\eta$-law for sum types is similar.
A type denotes a set.

\[
\begin{align*}
\llbracket \text{bool} \rrbracket &= \mathbb{B} \overset{\text{def}}{=} \{\text{true, false}\} \\
\llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\
\llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket
\end{align*}
\]

A term $\Gamma \vdash M : B$ denotes a function $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket B \rrbracket$.

**Substitution Lemma**

Given terms $\Gamma, x : A \vdash M : B$ and $\Gamma \vdash N : B$

we can obtain $\llbracket M[N/x] \rrbracket$ from $\llbracket M \rrbracket$ and $\llbracket N \rrbracket$. It is

\[
\rho \mapsto \llbracket M \rrbracket(\rho, x \mapsto \llbracket N \rrbracket(\rho))
\]

**Corollary**

The denotational semantics validates the $\beta$ and $\eta$ laws.
In CBN the terminals are $\text{true}$, $\text{false}$, $\text{inl} \ M$, $\text{inr} \ M$, $\lambda x. M$

To evaluate

- $\text{true}$, return $\text{true}$.
- $\lambda x. M$, return $\lambda x. M$.
- $\text{inl} \ M$, return $\text{inl} \ M$.
- let $M$ be $x. \ N$, evaluate $N[M/x]$.
- pm $M$ as $\{\text{true.} \ N, \text{false.} \ N'\}$, evaluate $M$. If it returns $\text{true}$, evaluate $N$, but if it returns $\text{false}$, evaluate $N'$.
- pm $M$ as $\{\text{inl} \ x. \ N, \text{inr} \ x. \ N'\}$, evaluate $M$. If it returns $\text{inl} \ P$, evaluate $N[P/x]$, but if it returns $\text{inr} \ P$, evaluate $N'[P/x]$.
- $MN$, evaluate $M$. If it returns $\lambda x. P$, evaluate $P[N/x]$. 
CBV terminals $T ::= \text{true} \mid \text{false} \mid \text{inl } T \mid \text{inr } T \mid \lambda x. M$

To evaluate

- true, return true.
- $\lambda x. M$, return $\lambda x. M$.
- inl $M$, evaluate $M$. If it returns $T$, return inl $T$.
- pm $M$ as \{true.$N$, false.$N'$\}, evaluate $M$. If it returns true, evaluate $N$, but if it returns false, evaluate $N'$.
- pm $M$ as \{inl $x. N$, inr $x. N'$\}, evaluate $M$. If it returns inl $T$, evaluate $N[T/x]$, but if it returns inr $T$, evaluate $N'[T/x]$.
- $MN$, evaluate $M$. If it returns $\lambda x. P$, evaluate $N$. If that returns $T$, evaluate $P[T/x]$.
Adding computational effects

Errors

Let $E = \{\text{CRASH}, \text{BANG}, \text{WALLOP}\}$ be a set of errors. We add

\[
\Gamma \vdash \text{error } e : B
\]

To evaluate error $e$, halt with error message $e$.

Printing

Let $\mathcal{A}$ be a set of characters. We add

\[
\Gamma \vdash M : B \\
\Gamma \vdash \text{print } c. \ M : B
\]

To evaluate print $c. \ M$, print $c$ and then evaluate $M$. 
Exercises

1. Evaluate

   \[
   \text{let (error CRASH) be } x. 5
   \]

   in CBV and CBN

2. Evaluate

   \[
   (\lambda x.(x + x))(\text{print "hello"}. 4)
   \]

   in CBV and CBN.

3. Evaluate

   \[
   \text{pm (print "hello". inr error CRASH) as }
   \{
   \text{inl x. x + 1, inr y. 5}\}
   \]

   in CBV and CBN.
Big-Step Operational Semantics

We convert our CBV and CBN interpreters into big-step semantics, defined inductively.

**no effects**  We define a relation \( M \Downarrow T \) meaning \( M \) evaluates to \( T \).

**errors**  We define a relation \( M \Downarrow T \) meaning \( M \) evaluates to \( T \), and a relation \( M \Downarrow e \) meaning \( M \) raises error \( e \).

**printing**  We define a relation \( M \Downarrow m, T \) meaning \( M \) prints \( m \in \mathcal{A}^* \) and finally evaluates to \( T \).

For example, in the case of printing we have rules such as

\[
\begin{align*}
\text{true} & \Downarrow \varepsilon, \text{true} \\
\text{true} & \Downarrow \varepsilon, \text{true} & \text{pm} \ M \text{ as } \{
\text{true}.N, \text{false}.N'\} & \Downarrow m + m', T \\

M & \Downarrow m, \text{true} & N & \Downarrow m', T
\end{align*}
\]

These are proved deterministic and total using Tait’s method.
Two terms $\Gamma \vdash M, M' : B$ are observationally equivalent when $C[M]$ and $C[M']$ have the same behaviour for every ground (i.e. boolean) context $C[\cdot]$. 

Same behaviour means: print the same string, raise the same error, return the same boolean.

We write $M \simeq_{\text{CBV}} M'$ and $M \simeq_{\text{CBN}} M'$. 

Observational equivalence
The $\eta$-law for boolean type: has it survived?

$\eta$-law for bool

Any term $\Gamma, z : \text{bool} \vdash M : B$ can be expanded as

$$\text{pm } z \text{ as } \{ \text{true. } M[\text{true}/z], \text{false. } M[\text{false}, z] \}$$

Anything of boolean type is a boolean.

This holds in CBV, because $z$ can only be replaced by true or false.

But it’s broken in CBN, because $z$ might raise an error. For example,

$$\text{true } \not\equiv_{\text{CBN}} \text{pm } z \text{ as } \{ \text{true. true, false. true} \}$$

because we can apply the context

$$\text{let error CRASH be } z. [\cdot]$$

Similarly the $\eta$-law for sum types is valid in CBV but not in CBN.
The $\eta$-law for functions: has it survived?

$\eta$-law for $A \to B$

Any term $\Gamma \vdash M : A \to B$ can be expanded as

$$\lambda x. M x$$

Anything of function type is a function.

This fails in CBV, but it holds in CBN.

Similarly

$$\lambda x. \text{error } e \simeq_{\text{CBN}} \text{error } e$$
$$\lambda x. \text{print } c. M \simeq_{\text{CBN}} \text{print } c. \lambda x. M$$

Yet the two sides have different operational behaviour! What’s going on?

In CBN, a function gets evaluated only by being applied.
The pure calculus satisfies all the $\beta$- and $\eta$-laws.

With computational effects,
- CBV satisfies $\eta$ for boolean and sum types, but not function types
- CBN satisfies $\eta$ for function types, but not boolean and sum types.

We want denotational semantics that validate the appropriate $\eta$-laws.

We’ll do CBV first, as it’s easier.
Denotational Semantics of CBV (Moggi)

Take a (strong) monad $T$ on $\textbf{Set}$.

- For errors: $\bot + E$
- For printing: $A^* \times \bot$

Each type denotes a set (think: the set of terminals)

\[
\begin{align*}
\llbracket \text{bool} \rrbracket &= \mathbb{B} \\
\llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\
\llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow T[\llbracket B \rrbracket]
\end{align*}
\]

Each term $\Gamma \vdash M : B$ denotes a Kleisli morphism, i.e. a function $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} T[\llbracket B \rrbracket]$.

To prove the soundness of the denotational semantics, we need a substitution lemma.
Can we obtain $[M[N/x]]$ from $[M]$ and $[N]$?
Can we obtain $[M[N/x]]$ from $[M]$ and $[N]$? Not in CBV.
Can we obtain \([M[N/x]]\) from \([M]\) and \([N]\)?: Not in CBV.

For example, define \(z : \text{bool} \vdash M, M' : \text{bool}\) and \(\vdash N : \text{bool}\)

\[
\begin{align*}
M & \overset{\text{def}}{=} \text{true} \\
M' & \overset{\text{def}}{=} \text{pm } z \text{ as } \{\text{true. true, false. false}\} \\
N & \overset{\text{def}}{=} \text{error CRASH}
\end{align*}
\]

Then we want

\[
\begin{align*}
[M] &= [M'] \\
[M[N/x]] &\neq [M'[N/x]]
\end{align*}
\]

But we can give a lemma for the substitution of values:

\[
V ::= \text{true} \mid \text{false} \mid \text{inl } V \mid \text{inr } V \mid \lambda x. M \mid x
\]

The terminals are the closed values.
Substitution Lemma For Values

Each value $\Gamma \vdash V : B$ denotes a function $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket V \rrbracket_{\text{val}}} \llbracket B \rrbracket$ such that

\[
\begin{array}{c}
\llbracket \Gamma \rrbracket \\
\downarrow
\end{array}
\xrightarrow{\llbracket V \rrbracket_{\text{val}}}
\begin{array}{c}
\llbracket B \rrbracket \\
\downarrow \eta[B]
\end{array}
\xrightarrow{T[B]}
\]

commutes.

Substitution Lemma

Given a term $\Gamma, x : A \vdash M : B$ and a value $\Gamma \vdash V : A$

we can obtain $\llbracket M[V/x] \rrbracket$ from $\llbracket M \rrbracket$ and $\llbracket V \rrbracket_{\text{val}}$. It is

$$\rho \mapsto \llbracket M \rrbracket(\rho, x \mapsto \llbracket V \rrbracket_{\text{val}} \rho)$$
Errors

- If \( M \Downarrow V \) then \( \llbracket M \rrbracket_{\varepsilon} = \text{inl} (\llbracket V \rrbracket_{\text{val} \varepsilon}) \).
- If \( M \Downarrow e \) then \( \llbracket M \rrbracket_{\varepsilon} = \text{inr} \ e \).

Printing

- If \( M \Downarrow m, V \) then \( \llbracket M \rrbracket_{\varepsilon} = \langle m, \llbracket V \rrbracket_{\text{val} \varepsilon} \rangle \).

These are straightforward inductions, using the substitution lemma.
Each type denotes a set (think: the set of closed terms). For example \( \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \) should denote \( T\text{B} \rightarrow (T\text{B} \rightarrow T\text{B}) \).

We define

\[
\begin{align*}
\llbracket \text{bool} \rrbracket &= T\text{B} \\
\llbracket A + B \rrbracket &= T(\llbracket A \rrbracket + \llbracket B \rrbracket) \\
\llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket
\end{align*}
\]

Each term \( \Gamma \vdash M : B \) should denote a function \( \llbracket \Gamma \rrbracket \xrightarrow{[M]} \llbracket B \rrbracket \).
\[ \Gamma \vdash \text{error } e : B \] denotes \( \rho \mapsto ? \)
Carrier Semantics: What Goes Wrong

\[ \Gamma \vdash \text{error } e : B \]

denotes $\rho \mapsto ?$

Example:

- suppose $B = \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})$
- then $B$ denotes $(\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E))$
- and $\text{error } e \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error } e$
- so the answer should be $\lambda x. \lambda y. \text{inr } e$.

Intuition: go down through the function types until we hit a boolean or sum type.
Example:

- Suppose $B = \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})$
- Then $B$ denotes $(\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E))$
- And $\text{error} \ e \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error} \ e$
- So the answer should be $\lambda x. \lambda y. \text{inr} \ e$.

Intuition: go down through the function types until we hit a boolean or sum type.
A similar problem arises with $\text{pm}$.
A CBN type should denote a set $X$ (the carrier) with some designated elements $E \xrightarrow{\text{error}} X$.

This is called an $E$-set.

Thus bool denotes $\mathbb{B} + E$ with $e \mapsto \text{inr } e$.

If $\llbracket A \rrbracket = (X, \text{error})$ and $\llbracket B \rrbracket = (Y, \text{error}')$, then $A + B$ denotes $(X + Y) + E$ with $e \mapsto \text{inr } e$ and $A \to B$ denotes $X \to Y$ with $e \mapsto \lambda x. \text{error}'(e)$.

Can we generalize the notion of $E$-set to other monads on $\textbf{Set}$?
An *Eilenberg-Moore algebra* for a monad $T$ on $\textbf{Set}$ is
- a set $X$ (the carrier)
- a function $TX \xrightarrow{\theta} X$ (the structure)
satisfying

\[
\begin{array}{ccc}
X & \xrightarrow{\eta_X} & TX \\
\downarrow{id} & & \downarrow{\theta} \\
X & \xrightarrow{\theta} & TX
\end{array}
\quad
\begin{array}{ccc}
X & \xleftarrow{\mu_X} & T^2X \\
\downarrow{\theta} & & \downarrow{T\theta} \\
X & \xleftarrow{\theta} & TX
\end{array}
\]
Examples of Algebras

An algebra for the $\mathcal{-+E}$ monad is an $E$-set.
Examples of Algebras

An algebra for the \( - + E \) monad is an \( E \)-set.

An algebra for \( A^* \times - \) is an \( A \)-set

i.e. a set \( X \) together with a function \( A \times X \xrightarrow{*} X \).

This is what we need to interpret

\[
\frac{\Gamma \vdash M : B}{\Gamma \vdash \text{print } c.\ M : B} \quad c \in A
\]

If \( B \) denotes \((X,*)\) then \( \text{print } c.\ M \) denotes \( \rho \mapsto c \ast ([M]_{\rho}) \)
3 Ways Of Building Algebras

Free Algebras
Given a set $X$, the free $T$-algebra on $X$ has carrier $TX$ and structure $\mu X$.

Product Algebras
Given a family of $T$-algebras $(X_i, \theta_i)$, the product algebra $\prod_{i \in I} (X_i, \theta_i)$ has carrier $\prod_{i \in I} X_i$ and structure given pointwise.

Exponential Algebras
Given a set $A$ and a $T$-algebra $(X, \theta)$, the exponential algebra $A \rightarrow (X, \theta)$ has carrier $A \rightarrow X$ and structure given pointwise.
Let $T$ be a monad on $\textbf{Set}$. 

A type denotes a $T$-algebra.

- $\text{bool}$ denotes the free algebra on $\mathbb{B}$
- If $[A] = (X, \theta)$ and $[B] = (Y, \phi)$
  - then $A + B$ denotes the free algebra on $X + Y$
  - and $A \rightarrow B$ denotes the exponential algebra $X \rightarrow (Y, \phi)$. 
A term \( x : A, x' : A' \vdash M : B \) denotes a function between the carrier sets
\[ X \times X' \xrightarrow{[M]} Y. \]

\[ \Gamma \vdash M : B \quad \Gamma \vdash N : B \quad \Gamma \vdash N' : B \]
\[ \Gamma \vdash \text{pm } M \text{ as } \{ \text{true}.N, \text{false}.N' \} : B \]

If \( B \) denotes \((Y, \theta)\) then this term denotes
\[ \lambda \Gamma . \begin{array}{c}
\begin{array}{c}
[\Gamma] \\
\downarrow \\
[\Gamma] \times T B
\end{array} \\
\downarrow t[\Gamma],_B \\
T ([\Gamma] \times B) \\
\downarrow T [[N], [N']] \\
TY
\end{array} \]
\[ \begin{array}{c}
\begin{array}{c}
Y \\
\uparrow \theta
\end{array}
\end{array} \]
Soundness of algebra semantics for CBN

Errors

- If $M \downarrow T : B$ then $[M]_{\varepsilon} = [T]_{\varepsilon}$
- If $M \downarrow e : B$ then $[M]_{\varepsilon} = \text{error } e$ where $[B] = (X, \text{error})$

Printing

- If $M \downarrow m, T : B$ then $[M]_{\varepsilon} = m \star \star ([T]_{\varepsilon})$ where $[B] = (X, \star)$

Straightforward inductive proofs using the substitution lemma.
We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.
We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

These are instances of a general recipe using a monad $T$ on $\text{Set}$ and its algebras.
We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

These are instances of a general recipe using a monad $T$ on $\textbf{Set}$ and its algebras.

A CBV type denotes a set; a CBN type denotes a $T$-algebra.
We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

These are instances of a general recipe using a monad $T$ on $\textbf{Set}$ and its algebras.

A CBV type denotes a set; a CBN type denotes a $T$-algebra.

They are fundamentally different things.
We write $F^T X$ for the free $T$-algebra $(TX, \mu X)$ on $X$
We write $F^T X$ for the free $T$-algebra $(TX, \mu X)$ on $X$ and $U^T (X, \theta)$ for the carrier $X$ of a $T$-algebra $(X, \theta)$. 
We write $F^T X$ for the free $T$-algebra $(TX, \mu X)$ on $X$
and $U^T(X, \theta)$ for the carrier $X$ of a $T$-algebra $(X, \theta)$.

Our CBN semantics of types can be written

\[
\begin{align*}
[\text{bool}] &= F^T(1 + 1) \\
[A + B] &= F^T(U^T[A] + U^T[B]) \\
[A \rightarrow B] &= U^T[A] \rightarrow [B]
\end{align*}
\]
We write $F^T X$ for the free $T$-algebra $(TX, \mu X)$ on $X$ and $U^T(X, \theta)$ for the carrier $X$ of a $T$-algebra $(X, \theta)$.

Our CBN semantics of types can be written

\[
\begin{align*}
\llbracket \text{bool} \rrbracket &= F^T(1 + 1) \\
\llbracket A + B \rrbracket &= F^T(U^T[A] + U^T[B]) \\
\llbracket A \rightarrow B \rrbracket &= U^T[A] \rightarrow \llbracket B \rrbracket
\end{align*}
\]

And our CBV semantics of types can be written

\[
\begin{align*}
\llbracket \text{bool} \rrbracket &= 1 + 1 \\
\llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\
\llbracket A \rightarrow B \rrbracket &= U^T(A \rightarrow F^T[B])
\end{align*}
\]
Call-by-push-value has

- value types which (like CBV types) denote sets
- computation types which (like CBN types) denote $T$-algebras.

We underline computation types.
Call-by-push-value has

- value types which (like CBV types) denote sets
- computation types which (like CBN types) denote $T$-algebras.

We underline computation types.

Value types: $A ::= UB | \sum_{i \in I} A_i | 1 | A \times A$

Computation types: $B ::= FA | \prod_{i \in I} B_i | A \rightarrow B$
Call-by-push-value has

- **value types** which (like CBV types) denote sets
- **computation types** which (like CBN types) denote $T$-algebras.

We underline computation types.

\[
A ::= UB | \sum_{i \in I} A_i | 1 | A \times A
\]

\[
B ::= FA | \prod_{i \in I} B_i | A \rightarrow B
\]

Strangely function types are computation types, and $\lambda x. M$ is a computation.
An identifier gets bound to a value, so it has value type.
An identifier gets bound to a value, so it has value type.

A context $\Gamma$ is a finite set of identifiers with associated value type

\[ x_0 : A_0, \ldots, x_{m-1} : A_{m-1} \]
An identifier gets bound to a value, so it has value type.

A context $\Gamma$ is a finite set of identifiers with associated value type

$$x_0 : A_0 , \ldots , x_{m-1} : A_{m-1}$$

Judgement for a value: $\Gamma \vdash^v V : A$

Judgement for a computation: $\Gamma \vdash^c M : B$
An identifier gets bound to a value, so it has value type.

A context $\Gamma$ is a finite set of identifiers with associated value type

$$x_0 : A_0, \ldots, x_{m-1} : A_{m-1}$$

Judgement for a value: $\Gamma \vdash^v V : A$

Judgement for a computation: $\Gamma \vdash^c M : B$

- A value $\Gamma \vdash^v V : A$ denotes a function $[[\Gamma]] \xrightarrow{[V]} [[A]]$
- If $B$ denotes $(X, \theta)$, then a computation $\Gamma \vdash^c M : B$ denotes a function $[[\Gamma]] \xrightarrow{[M]} X$.

Note From the viewpoint of monad/algebra semantics, there is no difference between a computation $\Gamma \vdash^c M : B$ and a value $\Gamma \vdash^v V : UB$. 
**The type $FA$**

A computation in $FA$ returns a value in $A$.

\[
\Gamma \vdash^v V : A \quad \Gamma \vdash^c M : FA \quad \Gamma, x : A \vdash^c N : B
\]

\[
\Gamma \vdash^c \text{return } V : FA \quad \Gamma \vdash^c M \text{ to } x. N : B
\]

This follows Moggi and Filinski. to uses the structure of $[[B]]$.

**The type $UB$**

A value in $UB$ is a thunk of a computation in $B$.

\[
\Gamma \vdash^c M : B \quad \Gamma \vdash^v V : UB
\]

\[
\Gamma \vdash^v \text{thunk } M : UB \quad \Gamma \vdash^c \text{force } V : B
\]

The constructs thunk and force are inverse. They are invisible in monad/algebra semantics.
Identifiers

An identifier is a value.

\[
\frac{\Gamma, x : A, \Gamma' \vdash v : A}{\Gamma, x : A, \Gamma' \vdash v \text{ let } V \text{ be } x. M : B}
\]

We write `let` to bind an identifier.
The rules for 1 are similar.
Functions

\[ \Gamma, x : A \vdash^c M : B \]
\[ \Gamma \vdash^c \lambda x. M : A \rightarrow B \]
\[ \Gamma \vdash^c \lambda \{i : M_i\}_{i \in I} : \prod_{i \in I} B_i \]

\[ \Gamma \vdash^c M : A \rightarrow B \quad \Gamma \vdash^v V : A \]
\[ \Gamma \vdash^c MV : B \]
\[ \Gamma \vdash^c M : \prod_{i \in I} B_i \quad \hat{i} \in I \]
\[ \Gamma \vdash^c M \hat{i} : B \hat{i} \]
Functions

\[\begin{align*}
\Gamma, x : A &\vdash^c M : B \\
\Gamma &\vdash^c \lambda x. M : A \to B \\
\Gamma &\vdash^c M_i : B_i \quad (\forall i \in I) \\
\Gamma &\vdash^c \lambda\{i. M_i\}_{i \in I} : \prod_{i \in I} B_i \\
\Gamma, v : A &\vdash^v V : \mathcal{A} \\
\Gamma &\vdash^c M : A \to B \\
\Gamma &\vdash^c M V : B \\
\Gamma, M \hat{i} : B_{\hat{i}} &\vdash^c \lambda\{i. M\}_{i \in I} \hat{i} \in I \\
\Gamma &\vdash^c M^{\hat{i}} : B_{\hat{i}}
\end{align*}\]

It is often convenient to write applications operand-first, as \(V^\prime M\) and \(\hat{i}^\prime M\).
The terminals are computations:

\[
\text{return } V \quad \lambda x. M \quad \lambda \{i. M_i\}_{i \in I}
\]

To evaluate

- return \( V \), return return \( V \).
- \( M \) to \( x \). \( N \), evaluate \( M \). If it returns return \( V \), then evaluate \( N[V/x] \).
- \( \lambda x. N \), return \( \lambda x. N \)
- \( MV \), evaluate \( M \). If it returns \( \lambda x. N \), evaluate \( N[V/x] \).
- \( \lambda \{i. N_i\}_{i \in I} \), return \( \lambda \{i. N_i\}_{i \in I} \).
- \( M\hat{i} \), evaluate \( M \). If it returns \( \lambda \{i. N_i\}_{i \in I} \), evaluate \( N\hat{i} \).
- let \( V \) be \( x \). \( M \), evaluate \( M[V/x] \).
- force thunk \( M \), evaluate \( M \).
- \( \text{pm } \langle \hat{i}, V \rangle \) as \( \{\langle i, x \rangle. M_i\}_{i \in I} \), evaluate \( M\hat{i}[V/x] \).
- \( \text{pm } \langle V, V' \rangle \) as \( \langle x, y \rangle. M \), evaluate \( M[V/x, V'/y] \).
Decomposing CBV into CBPV

A CBV type translates into a value type.

\[ A \rightarrow B \iff U(A \rightarrow FB) \]

A CBV term \( x : A, y : B \vdash M : C \) translates as \( x : A, y : B \vdash^c M : FC \).

\[
\begin{align*}
x & \mapsto \text{return } x \\
\lambda x. M & \mapsto \text{return thunk } \lambda x. M \\
M N & \mapsto M \text{ to } f. N \text{ to } y. ((\text{force } f) y) \\
\text{let } M \text{ be } x. N & \mapsto M \text{ to } y. \text{let } y \text{ be } x. N
\end{align*}
\]
Decomposing CBV into CBPV

A CBV type translates into a value type.

\[ A \rightarrow B \iff U(A \rightarrow FB) \]

A CBV term \( x : A, y : B \vdash M : C \) translates as \( x : A, y : B \vdash^c M : FC \).

\[
\begin{align*}
x & \mapsto \text{return } x \\
\lambda x. M & \mapsto \text{return thunk } \lambda x. M \\
M \ N & \mapsto M \to f. N \to y. ((\text{force } f) \ y) \\
\text{let } M \ \text{be } x. \ N & \mapsto M \to y. \text{let } y \ \text{be } x. \ N \\
\text{or} & \mapsto M \to x. \ N
\end{align*}
\]
A CBN type translates into a computation type.

\[
\begin{align*}
\text{bool} & \mapsto F(1 + 1) \\
A + B & \mapsto F(UA + UB) \\
A \rightarrow B & \mapsto UA \rightarrow B
\end{align*}
\]

A CBN term \( x : A, y : B \vdash M : C \) translates as \( x : UA, y : UB \vdash^c M : B \).

\[
\begin{align*}
x & \mapsto \text{force } x \\
\text{let } M \text{ be } x. \ N & \mapsto \text{let (thunk } M \text{) be } x. \ N \\
\lambda x. \ M & \mapsto \lambda x. \ M \\
M \ N & \mapsto M \ (\text{thunk } N) \\
\text{inl } M & \mapsto \text{return inl thunk } M
\end{align*}
\]
We’ve seen the CBPV calculus, its operational and monad/algebra semantics.
We’ve seen the CBPV calculus, its operational and monad/algebra semantics.

The translations from CBV and CBN into CBPV preserve these semantics.
Summary

We’ve seen the CBPV calculus, its operational and monad/algebra semantics.
The translations from CBV and CBN into CBPV preserve these semantics.
Moggi’s $TA$ is $UFA$. 

Summary

We’ve seen the CBPV calculus, its operational and monad/algebra semantics.

The translations from CBV and CBN into CBPV preserve these semantics.

Moggi’s $TA$ is $UFA$.

We still don’t understand why a function is a “computation”.

Paul Blain Levy (University of Birmingham) Call-by-push-value December 19, 2007 39 / 61
An operational semantics due to Felleisen-Friedman (1986).

It can be used for CBV, CBN and CBPV.

At any time, there’s a computation (C) and a stack of contexts (K).

Initially and finally, K is the empty stack \texttt{nil}.

Some authors make K into a single context, called an evaluation context.
Transitions for sequencing

To evaluate $\text{M to x. N}$, first evaluate $\text{M}$. If this returns the terminal return $\text{V}$, then evaluate $\text{N}[\text{V/x}]$.

$\text{M to x. N}$

$\text{M}$

$\text{K} \rightsquigarrow$

$\text{to x. N :: K}$

$\text{return V}$

$\text{N[V/x]}$

$\text{to x. N :: K} \rightsquigarrow$

$\text{K}$
To evaluate $V'M$, first evaluate $M$. If this returns the terminal $\lambda x.N$, then evaluate $N[V/x]$.

\[
\begin{align*}
V'M & \quad K \\
M & \quad V :: K \\
\lambda x.N & \quad V :: K \\
N[V/x] & \quad K
\end{align*}
\]
Those function rules again

\[
\begin{align*}
V'M & \quad K \quad \leadsto \\
M & \quad V :: K \\
\lambda x.N & \quad V :: K \quad \leadsto \\
N[V/x] & \quad K
\end{align*}
\]
Those function rules again

\[
V^\prime M \quad \quad K \\
M \quad \quad V :: K
\]

\[
\lambda x. N \quad \quad V :: K \\
N[V/x] \quad \quad K
\]

We can read \(V^\prime\) as an instruction “push \(V\)”.

We can read \(\lambda x\) as an instruction “pop \(x\)”.
Those function rules again

\[ \begin{align*}
V' & \Rightarrow M & \quad K \\
M & \Rightarrow V :: K
\end{align*} \]

\[ \begin{align*}
\lambda x. N & \Rightarrow V :: K & \quad \sim \Rightarrow \\
N[V/x] & \Rightarrow K
\end{align*} \]

We can read \( V' \) as an instruction “push \( V \)”.

We can read \( \lambda x \) as an instruction “pop \( x \)”.

Revisiting some equations:

\[ \begin{align*}
V' \lambda x. M & = M[V/x] \\
M & = \lambda x. x' M \quad (x \text{ fresh}) \\
\lambda x. \text{error } e & = \text{error } e \\
\lambda x. \text{print } c. M & = \text{print } c. \lambda x. M
\end{align*} \]
A value is, a computation does.

- A value of type $UB$ is a thunk of a computation of type $B$.
- A value of type $\sum_{i \in I} A_i$ is a pair $\langle i, V \rangle$.
- A value of type $A \times A'$ is a pair $\langle V, V' \rangle$.

- A computation of type $FA$ returns a value of type $A$.
- A computation of type $A \rightarrow B$ pops a value in $A$, then behaves in $B$.
- A computation of type $\prod_{i \in I} B_i$ pops a tag $i \in I$, then behaves in $B_i$. 
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
Typing the CK-machine

Initial Configuration

\[ M \quad C \quad \text{nil} \quad C \]

Transitions

- \[ M \text{ to } x.\ N \quad B \quad K \quad C \quad \leadsto \]
- \[ M \quad FA \quad \text{to } x.\ N :: K \quad C \]
- \[ \text{return } V \quad FA \quad \text{to } x.\ N :: K \quad C \quad \leadsto \]
- \[ N[V/x] \quad B \quad K \quad C \]

We write \( B \vdash^k K : C \) to mean that \( K \) can accompany a computation of type \( B \) during the evaluation of a computation of type \( C \).
Typing the CK-machine

Initial Configuration

\[ M \quad C \quad \text{nil} \quad C \]

Transitions

\[ M \text{ to } x. \ N \quad B \quad K \quad C \quad \rightsquigarrow \]
\[ M \quad FA \quad \text{to } x. \ N :: K \quad C \]

\[ \text{return } V \quad FA \quad \text{to } x. \ N :: K \quad C \quad \rightsquigarrow \]
\[ N[V/x] \quad B \quad K \quad C \]

We write \( B \vdash^k K : C \) to mean that \( K \) can accompany a computation of type \( B \) during the evaluation of a computation of type \( C \).

More generally \( \Gamma \mid B \vdash^k K : C \) when there are free identifiers.
The Stack Judgement

The typing rules can be read off from the CK-machine transitions.

### Typing Rules For Stacks

\[
\begin{align*}
\Gamma | C \vdash^k \text{nil} : C \\
\Gamma \vdash^v V : A & \quad \Gamma | B \vdash^k K : C \\
\Gamma | A \rightarrow B \vdash^k V :: K : C \\
\Gamma, x : A \vdash^c M : B & \quad \Gamma | B \vdash^k K : C \\
\Gamma | FA \vdash^k \text{to } x. M :: K : C \\
\Gamma | B_{\hat{i}} \vdash^k K : C \\
\Gamma | \prod_{i \in I} B_i \vdash^k \hat{i} :: K : C
\end{align*}
\]
The Stack Judgement

The typing rules can be read off from the CK-machine transitions.

### Typing Rules For Stacks

**Typing Rules For Stacks**

\[
\begin{align*}
\frac{}{\Gamma \mid \text{nil} : C} & \quad \frac{\Gamma, x : A \vdash M : B \quad \Gamma \mid B \vdash K : C}{\Gamma \mid FA \vdash \text{to } x. \ M :: K : C} \\
\frac{\Gamma \vdash V : A \quad \Gamma \mid B \vdash K : C}{\Gamma \mid A \rightarrow B \vdash V :: K : C} & \quad \frac{\Gamma \mid B \hat{i} \vdash K : C}{\Gamma \mid \prod_{i \in I} B_i \vdash \hat{i} :: K : C} \\
\end{align*}
\]

A stack from an $F$ type is often called a **continuation**.
Denotational semantics of stacks

If $\llbracket B \rrbracket = (X, \theta)$ and $\llbracket C \rrbracket = (Y, \phi)$

then a stack $\Gamma \mid B \vdash^k K : C$ denotes a function $\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y$

homomorphic in its second argument.

Concatenation of stacks corresponds to composition of homomorphisms.
Denotational semantics of stacks

If $[[B]] = (X, \theta)$ and $[[C]] = (Y, \phi)$

then a stack $\Gamma \mid_B \vdash^k K : C$ denotes a function $[[\Gamma]] \times X \xrightarrow{[K]} Y$

homomorphic in its second argument.

Concatenation of stacks corresponds to composition of homomorphisms.

We have an adjunction between the category of values (semantically: sets and functions) and the category of stacks (semantically: $T$-algebras and homomorphisms).

$$
\begin{array}{ccc}
\text{Set} & \xrightarrow{F^T} & \text{Set}^T \\
\downarrow & & \downarrow U^T \\
\text{Set} & \xleftarrow{\perp} & \text{Set}^T
\end{array}
$$

This resolves the monad $T$ on $\text{Set}$. 

Consider CBPV extended with 2 storage cells: \texttt{fred} stores a natural number and \texttt{mary} stores a boolean.

\[
\begin{align*}
\Gamma \vdash^c M : B \\
\Gamma \vdash^c \texttt{fred} := n. M : B & \quad n \in \mathbb{N} \\
\Gamma \vdash^c M_n : B & \quad (\forall n \in \mathbb{N}) \\
\Gamma \vdash^c \text{read \texttt{fred} as \{n. M_n\}}_{n \in \mathbb{N}} : B
\end{align*}
\]

A state is \texttt{fred} $\mapsto n$, \texttt{mary} $\mapsto b$.

The set of states is $S \cong \mathbb{N} \times \mathbb{B}$.
The big-step semantics takes the form $s, M \downarrow s', T$.

A pair $s, M$ is called an SC-configuration.

Formally, we define a judgement $\Gamma \vdash^{sc} P : B$ with formation rule

$$
\frac{
\Gamma \vdash^c M : B
}{
\Gamma \vdash^{sc} s, M : B
}
$$
Moggi’s monad for global state is $S \rightarrow (S \times \_)$.

We can take algebras for this and obtain a denotational semantics of CBPV with state.
Moggi’s monad for global state is $S \to (S \times -)$.

We can take algebras for this and obtain a denotational semantics of CBPV with state.

But it doesn’t fit well with SC-configurations.

We’d like a soundness result of the following form:

$$\text{If } s, M \Downarrow s', T \text{ then } \llbracket s, M \rrbracket_\varepsilon = \llbracket s', T \rrbracket_\varepsilon$$

This requires an SC-configuration to have a denotation.
Value type $A$ denotes the set of \textit{denotations of} values of type $A$. Like in monad semantics.
Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.

Computation type $[[B]]$ denotes the set of behaviours of configurations of type $B$. 
Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.

Computation type $[[B]]$ denotes the set of behaviours of configurations of type $B$.

Thus an SC-configuration $\Gamma \vdash^{sc} P : B$ denotes a function $[[\Gamma]] \xrightarrow{[P]} [[B]]$. 
Semantics of SC-configurations

Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.

Computation type $[B]$ denotes the set of behaviours of configurations of type $B$.

Thus an SC-configuration $\Gamma \vdash^{\text{sc}} P : B$ denotes a function $[\Gamma] \overset{[P]}{\rightarrow} [B]$.

The behaviour of a computation $\Gamma \vdash^{\text{c}} M : B$ depends on state and environment. So $\Gamma \vdash^{\text{c}} M : B$ denotes a function $S \times [\Gamma] \overset{[M]}{\rightarrow} [B]$.

In particular, the configuration $s, M$ denotes $\rho \mapsto [M](s, \rho)$. 
State: semantics of types

An SC-configuration of type $FA$ will terminate as $s$, return $V$.

$$\llbracket FA \rrbracket = S \times \llbracket A \rrbracket$$

An SC-configuration of type $A \to B$ will pop $x : A$, then behave in $B$.

$$\llbracket A \to B \rrbracket = \llbracket A \rrbracket \to \llbracket B \rrbracket$$

An SC-configuration of type $\prod_{i \in I} B_i$ will pop $i \in I$, then behave in $B_i$.

$$\llbracket \prod_{i \in I} B_i \rrbracket = \prod_{i \in I} \llbracket B_i \rrbracket$$

A value $\Gamma \vdash^v V : UB$ can be forced in any state $s$, giving an SC-configuration $s$, force $V$.

$$\llbracket UB \rrbracket = S \to \llbracket B \rrbracket$$
An SC-configuration of type $FA$ will terminate as $s$, return $V$.

$$[FA] = S \times [A]$$

An SC-configuration of type $A \rightarrow B$ will pop $x : A$, then behave in $B$.

$$[A \rightarrow B] = [A] \rightarrow [B]$$

An SC-configuration of type $\prod_{i \in I} B_i$ will pop $i \in I$, then behave in $B_i$.

$$[\prod_{i \in I} B_i] = \prod_{i \in I} [B_i]$$

A value $\Gamma \vdash^V V : UB$ can be forced in any state $s$, giving an SC-configuration $s$, force $V$.

$$[UB] = S \rightarrow [B]$$

We recover standard semantics for CBV, and O’Hearn’s semantics for CBN.
A stack $\Gamma \vdash B \hookrightarrow^k K : C$ can be applied to an SC-configuration giving another SC-configuration.
A stack $\Gamma \vdash^k K : C$ can be applied to an SC-configuration giving another SC-configuration.

Accordingly it denotes a function $[[\Gamma]] \times [[B]] \xrightarrow{[K]} [[C]]$. 
A stack $\Gamma \upharpoonright B \vdash^k K : C$ can be applied to an SC-configuration giving another SC-configuration.

Accordingly it denotes a function $[[\Gamma]] \times [[[B]]] \xrightarrow{[[K]]} [[[C]]]$.

Concatenation of stacks corresponds to composition of functions.
A stack $\Gamma \mid B \vdash^k K : C$ can be applied to an SC-configuration giving another SC-configuration.

Accordingly it denotes a function $[[\Gamma]] \times [[[B]]] \rightarrow [[[C]]]$.

Concatenation of stacks corresponds to composition of functions.

So we have an adjunction

$$
\begin{array}{ccc}
\text{Set} & \overset{\uparrow}{\underset{\rightarrow}{\longrightarrow}} & \text{Set} \\
\downarrow & & \\
S & & S
\end{array}
$$
Extend CBPV with two instructions for changing the stack:

- `letstk x` means “let x be the current stack”
- `changestk V` means “change the current stack to V”.

A stack $K$ can now be turned into a value $sv\ K$.

\[
\begin{align*}
&\text{letstk } x.\ M \quad B \quad K \quad C \\
&\quad \xrightarrow{\sim} \\
&\quad M[sv\ K/x] \quad B \quad K \quad C
\end{align*}
\]

\[
\begin{align*}
&\text{changestk } sv\ K.\ M \quad B' \quad K \quad C \\
&\quad \xrightarrow{\sim} \\
&\quad M \quad B \quad K \quad C
\end{align*}
\]
Typing rules

We need a new kind of value type:

\[ A ::= UB \mid \sum_{i \in I} A_i \mid 1 \mid A \times A \mid \text{stk } B \]

\[ B ::= FA \mid \prod_{i \in I} B_i \mid A \rightarrow B \]

A value of type \text{stk } B is a stack from B. (The target type is fixed within a given term.)

Typing rules for control operators

\[ \Gamma, x : \text{stk } B \vdash^c M : B \quad \Gamma \vdash^v V : \text{stk } B \quad \Gamma \vdash^c M : B \]

\[ \Gamma \vdash^c \text{letstk } x. M : B \quad \Gamma \vdash^c \text{changestk } V. M : B' \]

We have to treat \text{nil} as a free identifier:

\[ \Gamma \mid B \vdash^k K : C \]

\[ \Gamma, \text{nil} : \text{stk } C \vdash^k \text{sv } K : \text{stk } B \]
Fix a set $R$, the set of behaviours of CK-configurations.
Monad/algebra semantics of control

Fix a set $R$, the set of behaviours of CK-configurations.

Moggi's monad for control operators ("continuations") is $(- \to R) \to R$. 

Fix a set $R$, the set of behaviours of CK-configurations.

Moggi’s monad for control operators ("continuations") is $(- \to R) \to R$.

Maybe we can use algebras for this to build a denotational semantics of control.
Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.
Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.

Computation type $\llbracket B \rrbracket$ denotes the set of stacks from $B$. 
Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.

Computation type $[[B]]$ denotes the set of stacks from $B$.
Thus we will have $[[\text{stk } B]] = [[B]]$. 
Value type $A$ denotes the set of denotations of values of type $A$. Like in monad semantics.

Computation type $[[B]]$ denotes the set of stacks from $B$.

Thus we will have $[[\text{stk } B]] = [[B]]$.

The behaviour of a computation $\Gamma \vdash^c M : B$ depends on environment and stack, so it denotes $[[\Gamma]] \times [[B]] \xrightarrow{[[M]]} R$.
Control: semantics of types

A stack from $FA$ receives a value $x : A$ and then behaves as a configuration.

$$[[FA]] = [[A]] \to R$$

A stack from $A \to B$ is a pair $V :: K$.

$$[[A \to B]] = [[A]] \times [[B]]$$

A stack from $\prod_{i \in I} B_i$ is a pair $i :: K$.

$$[[\prod_{i \in I} B_i]] = \sum_{i \in I} [[B_i]]$$

A value of type $UB$ can be forced alongside any stack $K$, giving a configuration.

$$[[UB]] = [[B]] \to R$$
A stack from $FA$ receives a value $x : A$ and then behaves as a configuration.

$$\llbracket FA \rrbracket = \llbracket A \rrbracket \rightarrow R$$

A stack from $A \rightarrow B$ is a pair $V :: K$.

$$\llbracket A \rightarrow B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket$$

A stack from $\prod_{i \in I} B_i$ is a pair $i :: K$.

$$\llbracket \prod_{i \in I} B_i \rrbracket = \sum_{i \in I} \llbracket B_i \rrbracket$$

A value of type $UB$ can be forced alongside any stack $K$, giving a configuration.

$$\llbracket UB \rrbracket = \llbracket B \rrbracket \rightarrow R$$

We recover standard continuation semantics for CBV, and Streicher-Reus’ semantics for CBN.
A stack $\Gamma \mid B \vdash^k K : C$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk} C \vdash^v V : \text{stk} B$$
A stack $\Gamma \mid B \vdash^k K : C$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk} \ C \vdash^v V : \text{stk} \ B$$

Accordingly it denotes a function $[[\Gamma]] \times [[C]] \xrightarrow{[K]} [[B]]$. 
A stack $\Gamma \mid B \vdash^k K : C$ corresponds to a value

$$
\Gamma, \text{nil} : \text{stk} \ C \vdash^v V : \text{stk} \ B
$$

Accordingly it denotes a function $[[\Gamma]] \times [[C]] \xrightarrow{[K]} [[B]]$.

Concatenation of stacks corresponds to composition of “op-functions”.
Control: the value/stack adjunction

A stack $\Gamma \mid B \vdash^k K : C$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk} \ C \vdash^v V : \text{stk} \ B$$

Accordingly it denotes a function $[[\Gamma]] \times [[C]] \xrightarrow{[K]} [[B]]$.

Concatenation of stacks corresponds to composition of “op-functions”.

So we have an adjunction

$$\text{Set} \xrightarrow{-\rightarrow R} \text{Set}^{\text{op}} \xleftarrow{-\rightarrow R}$$
Summary of models

For every monad $T$ on $\text{Set}$ we have an adjunction

$$\text{Set} \xrightarrow{\perp} \text{Set}^T \xleftarrow{\perp} \text{Set}$$

This is useful for modelling CBPV with errors and printing.
Summary of models

For every monad $T$ on $\textbf{Set}$ we have an adjunction

$$\textbf{Set} \xrightarrow{F^T} \textbf{Set}^T \xleftarrow{U^T} \textbf{Set}$$

This is useful for modelling CBPV with errors and printing.

For a set $S$ we have an adjunction

$$\textbf{Set} \xrightarrow{S \times -} \textbf{Set} \xleftarrow{S \rightarrow -}$$

This is useful for modelling CBPV with state.
Summary of models

For every monad $T$ on $\text{Set}$ we have an adjunction

$$
\text{Set} \xleftarrow{U^T} \text{Set}^T \xrightarrow{F^T} \text{Set}
$$

This is useful for modelling CBPV with errors and printing.

For a set $S$ we have an adjunction

$$
\text{Set} \xleftarrow{S \rightarrow \_} \text{Set} \xrightarrow{\_ \times S} \text{Set}
$$

This is useful for modelling CBPV with state.

For a set $R$ we have an adjunction

$$
\text{Set} \xleftarrow{\_ \rightarrow \_} \text{Set}^{\text{op}} \xrightarrow{R \rightarrow \_} \text{Set}
$$

This is useful for modelling CBPV with control.
Summary of models

For every monad $T$ on $\text{Set}$ we have an adjunction

$$
\begin{array}{c}
\text{Set} \\
\downarrow F^T \\
\downarrow U^T \\
\text{Set}^T
\end{array}
$$

This is useful for modelling CBPV with errors and printing.

For a set $S$ we have an adjunction

$$
\begin{array}{c}
\text{Set} \\
\downarrow S \times - \\
\downarrow S \rightarrow - \\
\text{Set}
\end{array}
$$

This is useful for modelling CBPV with state.

For a set $R$ we have an adjunction

$$
\begin{array}{c}
\text{Set} \\
\downarrow - \rightarrow R \\
\downarrow - \rightarrow R \\
\text{Set}^{\text{op}}
\end{array}
$$

This is useful for modelling CBPV with control.