A tutorial on call-by-push-value

Paul Blain Levy

University of Birmingham

October 12, 2013
Outline

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

$$A ::= \text{bool} \mid A + A \mid A \to A$$

### Typing judgement

$$\Gamma \vdash M : A$$

### Terms

$$M ::= x \mid \text{let } M \text{ be } x. \ M$$

$$\mid \text{true} \mid \text{false} \mid \text{case } M \text{ of } \{\text{true. } M, \text{ false. } M\}$$

$$\mid \text{inl } M \mid \text{inr } M \mid \text{case } M \text{ of } \{\text{inl } x. \ M, \text{ inr } x. \ M\}$$

$$\mid \lambda x. M \mid M M$$
Equational Laws

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

<table>
<thead>
<tr>
<th>$\eta$-law for $A \rightarrow B$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Any term $\Gamma \vdash M : A \rightarrow B$ can be expanded as</td>
<td>$\lambda x. M x$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>$\eta$-law for bool</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Any term $\Gamma, z : \text{bool} \vdash M : B$ can be expanded as</td>
<td>case $z$ of {$true. M[true/z]$, $false. M[false/z]$}</td>
</tr>
</tbody>
</table>

Anything of boolean type is a boolean.

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

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

A term \( \Gamma \vdash M : B \) denotes a function \( \Gamma \xrightarrow{[M]} [B] \).

**Substitution Lemma**

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

we can obtain \( [M[N/x]] \) from \( [M] \) and \( [N] \). It is

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

**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 $\text{inl} \ M$.
- $\text{let} \ M \text{ be } x. \ N$: evaluate $N[M/x]$.
- $\text{case} \ M \text{ of } \{ \text{true. } N, \text{ false. } N' \}$: evaluate $M$. If it returns $\text{true}$, evaluate $N$, but if it returns $\text{false}$, evaluate $N'$.
- $\text{case} \ M \text{ of } \{ \text{inl} \ x. \ N, \text{ inr} \ x. \ N' \}$: 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 $P[N/x]$. 
Call-by-value evaluation of a closed term

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**.
- **\text{inl } M**: evaluate **M**. If it returns **T**, return **inl T**.
- **let **M** be x. \text{N}**: evaluate **M**. If it returns **T**, evaluate **N[T/x]**.
- **case **M** of \{ **true. \text{N}, \text{false. \text{N'}} \}**: evaluate **M**. If it returns **true**, evaluate **N**, but if it returns **false**, evaluate **N’**.
- **case **M** of \{ **inl x. \text{N}, \text{inr x. \text{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 \( E = \{\text{CRASH, BANG, WALLOP}\} \) be a set of “errors”. We add

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

To evaluate \text{error } e: halt with error message \( e \).

### Printing

Let \( A = \{a, b, c, d, e\} \) be a set of “characters”. We add

\[
\Gamma \vdash M : B
\]

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

To evaluate \text{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

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

   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 $M \not\Downarrow e$ meaning $M$ raises error $e$.

**printing** We define a relation $M \Downarrow m, T$ meaning $M$ prints $m \in A^*$ and finally evaluates to $T$.

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

\[
\begin{align*}
\text{true} & \Downarrow \varepsilon, \text{true} & \text{case } M \text{ of } \{\text{true. } N, \text{ false. } N'\} & \Downarrow mm', T
\end{align*}
\]

These are proved deterministic and total using Tait’s method.
Observational equivalence

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 $M \simeq_{\text{CBN}} M'$. 
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

$$\text{case } z \text{ of } \{ \text{true. } M[\text{true}/z], \text{false. } M[\text{false}/z]\}$$

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,

$$\text{true } \not\simeq_{\text{CBN}} \text{ case } z \text{ of } \{ \text{true. true, false. true}\}$$

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

- For errors: $- + E$
- For printing: $\mathcal{A}^* \times -$ 

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

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

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

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.

**Example that rules out a general substitution lemma**

Define \(x : \text{bool} \vdash M, M' : \text{bool}\) and \(\vdash N : \text{bool}\)

\[
M \overset{\text{def}}{=} \text{true} \\
M' \overset{\text{def}}{=} \text{case x of } \{\text{true. true, false. true}\} \\
N \overset{\text{def}}{=} \text{error CRASH}
\]

\([M] = [M']\) because \(M =_{\eta \text{bool}} M'\)

\([M[N/x]] \neq [M'[N/x]]\)
Can we obtain \([M[N/x]]\) from \([M]\) and \([N]\)? Not in CBV.

**Example that rules out a general substitution lemma**

Define \(x : \text{bool} \vdash M, M' : \text{bool}\) and \(\vdash N : \text{bool}\)

\[
\begin{align*}
M & \overset{\text{def}}{=} \text{true} \\
M' & \overset{\text{def}}{=} \text{case } x \text{ of } \{ \text{true. true}, \text{false. true} \} \\
N & \overset{\text{def}}{=} \text{error CRASH}
\end{align*}
\]

\([M] = [M']\) because \(M =_{\eta_{\text{bool}}} M'\)

\([M[N/x]] \neq [M'[N/x]]\)

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 $[[\Gamma]] \xrightarrow{[V]^{val}} [[B]]$ such that

$$
\begin{aligned}
&\quad [\Gamma] \xrightarrow{[V]^{val}} [[B]] \\
&\downarrow \quad \downarrow \eta[B] \\
&[[V]] \quad \quad \quad \quad \quad \quad \quad T[[B]]
\end{aligned}
$$

Substitution Lemma

Given a term $\Gamma, x : A \vdash M : B$ and a value $\Gamma \vdash V : A$
we can obtain $[[M[V/x]]]$ from $[[M]]$ and $[[V]^{val}]$. It is

$$
\rho \mapsto [[M]](\rho, x \mapsto [[V]^{val} \rho)
$$
Errors

- If $M \downarrow V$ then $[M]_\varepsilon = \text{inl} \ (\mathrm{Val}(V)_{\varepsilon})$.
- If $M \nmid e$ then $[M]_\varepsilon = \text{inr} \ e$.

Printing

- If $M \downarrow m, V$ then $[M]_\varepsilon = \langle m, \mathrm{Val}(V)_{\varepsilon} \rangle$.

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 $\text{T}\text{B} \rightarrow (\text{T}\text{B} \rightarrow \text{T}\text{B})$.
We define

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

Each term $\Gamma \vdash M : B$ should denote a function $[\Gamma] \xrightarrow{[M]} [B]$. 
\[ \Gamma \vdash \text{error } e : B \]

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

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.

A similar problem arises with case.
$E$-pointed semantics of CBN types

A CBN type should denote a set $X$ (the carrier) with some designated elements $E \xrightarrow{\text{error}} X$.

This is called an $E$-pointed set.

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

If $[A] = (X, \text{error})$ and $[B] = (Y, \text{error}')$,

- then $A + B$ denotes $(X + Y) + E$ with $e \mapsto \text{inr} \ e$
- and $A \to B$ denotes $X \to Y$ with $e \mapsto \lambda x. \text{error}'(e)$. 
$E$-pointed semantics of CBN types

A CBN type should denote a set $X$ (the carrier) with some designated elements $E \xrightarrow{error} X$.

This is called an $E$-pointed set.

Thus $\text{bool}$ 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 $A \rightarrow B$ denotes $X \rightarrow Y$ with $e \mapsto \lambda x. \text{error}'(e)$.

Can we generalize the notion of $E$-pointed set to other monads on $\text{Set}$?
An *Eilenberg-Moore algebra* for a monad $T$ on $\text{Set}$ is

- a set $X$ (the carrier)
- a function $TX \xrightarrow{\theta} X$ (the structure)

satisfying

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

An algebra for the $- + E$ monad is an $E$-pointed set.
Examples of Algebras

An algebra for the $- + E$ monad is an $E$-pointed set.

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

i.e. a set $X$ together with a function $A \times X \rightarrow X$.

This is what we need to interpret

$$\Gamma \vdash M : B$$

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

If $B$ denotes $(X, \ast)$ 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 $A \to (X, \theta)$ has carrier $A \to X$ and structure given pointwise.
Let \( T \) be a monad on \( \text{Set} \).

A type denotes a \( T \)-algebra.

- \( \text{bool} \) denotes the free algebra on \( \mathbb{B} \)
- If \( \llbracket A \rrbracket = (X, \theta) \) and \( \llbracket B \rrbracket = (Y, \phi) \)
  - then \( A + B \) denotes the free algebra on \( X + Y \)
  - and \( A \to B \) denotes the exponential algebra \( X \to (Y, \phi) \).
Algebra semantics for CBN terms

Suppose $B$ denotes the algebra $(Y, \theta)$.

Then a term $\Gamma \vdash M : B$ denotes a function between the carrier sets $\frac{[\Gamma]}{\Gamma \vdash M : \text{bool}} \quad \frac{\Gamma \vdash N : B \quad \Gamma \vdash N' : B}{\Gamma \vdash \text{case } M \text{ of } \{ \text{true}. N, \text{false}. N' \} : B}

This term denotes

\[
\begin{array}{c}
\frac{[\Gamma]}{\langle \text{id}, [M] \rangle} \\
\downarrow \\
\frac{[\Gamma] \times T B}{t_{[\Gamma], B}} \\
\rightarrow \\
\frac{T([\Gamma] \times B)}{T[[N],[N']]坻} \\
\rightarrow \\
\frac{TY}{\theta} \\
\end{array}
\]
Errors

- If $M \Downarrow T : B$ then $[M]_\varepsilon = [T]_\varepsilon$
- If $M \not{\Downarrow} e : B$ then $[M]_\varepsilon = \text{error } e$ where $[B] = (X, \text{error})$

Printing

- If $M \Downarrow m, T : B$ then $[M]_\varepsilon = m ^* ^* ([T]_\varepsilon)$ where $[B] = (X, *)$

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

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 $\text{Set}$ 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 $\text{Set}$ 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

\[
\begin{align*}
[\text{bool}] &= F^T (1 + 1) \\
[A + B] &= F^T (U^T [A] + U^T [B]) \\
[A \to B] &= U^T [A] \to [B]
\end{align*}
\]
Semantics of Types, Again

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

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

And our CBV semantics of types can be written

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

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

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

**value type**

\[ A ::= UB \mid 1 \mid A \times A \mid 0 \mid A + A \mid \sum_{i \in \mathbb{N}} A_i \]

**computation type**

\[ B ::= FA \mid A \to B \mid 1_\Pi \mid B \Pi B \mid \Pi_{i \in \mathbb{N}} B_i \]
Call-by-push-value has

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

value type

\[ A ::= UB | 1 | A \times A | 0 | A + A | \sum_{i \in \mathbb{N}} A_i \]

computation type

\[ B ::= FA | A \rightarrow B | 1\Pi | B \Pi B | \prod_{i \in \mathbb{N}} B_i \]

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

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 $A$.

\[
\begin{align*}
& \quad \quad \Gamma \vdash^v V : A \\
\implies & \quad \quad \Gamma \vdash^c \text{return } V : FA \\
& \quad \quad \Gamma \vdash^c M : FA \\
& \quad \quad \Gamma, x : A \vdash^c N : B \\
\implies & \quad \quad \Gamma \vdash^c M \text{ to } x. N : B
\end{align*}
\]
The type $FA$

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

\[
\begin{align*}
\Gamma \vdash^v V : A & \quad \Gamma \vdash^c M : FA & \quad \Gamma, x : A \vdash^c N : B \\
\Gamma \vdash^c \text{return } V : FA & \quad \Gamma \vdash^c M \text{ to } x. N : B
\end{align*}
\]

The denotation of $M \text{ to } x. N$ uses the structure of $[B]$.

The type $UB$

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

\[
\begin{align*}
\Gamma \vdash^c M : B & \quad \Gamma \vdash^v V : UB \\
\Gamma \vdash^v \text{thunk } M : UB & \quad \Gamma \vdash^c \text{force } V : B
\end{align*}
\]

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

An identifier is a value.

\[ \Gamma \vdash^v x : A \quad (x : A) \in \Gamma \]

\[ \Gamma \vdash^c V : A \\
\Gamma, x : A \vdash^c M : B \\
\Gamma \vdash^c \text{let } V \text{ be } x. M : B \]

We write \texttt{let} to bind an identifier.
The rules for 1 are similar.
Functions

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

Γ, x : A ⊢^c M : B

Γ ⊢^c \( \lambda x. M : A \rightarrow B \)

Γ ⊢^c M : A \rightarrow B  Γ ⊢^v V : A

Γ ⊢^c MV : B

Γ ⊢^c M_i : B_i \ (\forall i \in I)

Γ ⊢^c \( \lambda \{i. M_i\}_{i \in I} : \prod_{i \in I} B_i \)

Γ ⊢^c M : \prod_{i \in I} B_i \ \hat{i} \in I

Γ ⊢^c M\hat{i} : B_{\hat{i}}

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

\[
\text{return } V \quad \lambda x. M \quad \lambda \{ i. M_i \}_{i \in I}
\]
The terminals are computations:

\[
\begin{align*}
\text{return } V & \quad \lambda x. M & \quad \lambda \{ i. M_i \}_{i \in I} \\
\text{To evaluate} & \\
\text{return } V: & \text{ return } \text{return } V. \\
M \text{ to } x. N: & \text{ evaluate } M. \text{ If it returns } \text{return } V, \text{ then evaluate } N[V/x]. \\
\lambda x. N: & \text{ return } \lambda x. N. \\
MV: & \text{ evaluate } M. \text{ If it returns } \lambda x. N, \text{ evaluate } N[V/x]. \\
\lambda \{ i. N_i \}_{i \in I}: & \text{ return } \lambda \{ i. N_i \}_{i \in I}. \\
M \hat{i}: & \text{ evaluate } M. \text{ If it returns } \lambda \{ i. N_i \}_{i \in I}, \text{ evaluate } N \hat{i}. \\
\text{let } V \text{ be } x. M: & \text{ evaluate } M[V/x]. \\
\text{force thunk } M: & \text{ evaluate } M. \\
\text{case } \langle \hat{i}, V \rangle \text{ of } \{ \langle i, x \rangle. M_i \}_{i \in I}: & \text{ evaluate } M \hat{i}[V/x]. \\
\text{case } \langle V, V' \rangle \text{ of } \langle x, y \rangle. M: & \text{ evaluate } M[V/x, V'/y].
\end{align*}
\]
Decomposing CBV into CBPV

A CBV type translates into a value type.

\[ A \rightarrow B \quad \mapsto \quad U(A \rightarrow FB) \]

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

\[ x \quad \mapsto \quad \text{return } x \]
\[ \lambda x. M \quad \mapsto \quad \text{return thunk } \lambda x. M \]
\[ M \; N \quad \mapsto \quad M \; \text{to } f. \; N \; \text{to } y. \; ((\text{force } f) \; y) \]
\[ \text{let } M \; \text{be } x. \; N \quad \mapsto \quad M \; \text{to } y. \; \text{let } y \; \text{be } x. \; N \]
Decomposing CBV into CBPV

A CBV type translates into a value type.

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

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

- \( x \mapsto \text{return } x \)
- \( \lambda x. M \mapsto \text{return thunk } \lambda x. M \)
- \( M \ N \mapsto M \text{ to } f. N \text{ to } y. ((\text{force } f) \ y) \)
- \( \text{let } M \text{ be } x. N \mapsto M \text{ to } y. \text{let } y \text{ be } x. N \)
- \( \text{or } \mapsto M \text{ to } x. N \)
Decomposing CBN into CBPV

A CBN type translates into a computation type.

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

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

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

We’ve seen the call-by-push-value calculus, its operational and monad/algebra semantics.

The translations from CBV and CBN into CBPV preserve these semantics.
We’ve seen the call-by-push-value calculus, its operational and monad/algebra semantics.

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

The translations from CBV and CBN into CBPV preserve these semantics. Moggi’s $TA$ is $UFA$. 

But
We’ve seen the call-by-push-value calculus, its operational and monad/algebra semantics.

The translations from CBV and CBN into CBPV preserve these semantics. Moggi’s $TA$ is $UFA$.

But

- the monad/algebra semantics makes thunk and force invisible
We’ve seen the call-by-push-value calculus, its operational and monad/algebra semantics.

The translations from CBV and CBN into CBPV preserve these semantics. Moggi’s $TA$ is $UFA$.

But

- the monad/algebra semantics makes thunk and force invisible
- we still don’t understand why a function is a “computation”.
Antecedents: CBV

- Landin’s ISWIM: CBV $\lambda$-calculus with effects. Influenced ML and other languages.
- Plotkin: $\lambda \nu$-calculus provided equations for CBV with divergence.
- Numerous researchers: CPS transforms for CBV.
- Felleisen et al: CBV semantics for various effects.
- Moggi: $\lambda c$-calculus, monads for CBV and monadic metalanguage.
- Power and Robinson: Freyd categories for CBV.
- Benton and Kennedy: MIL-lite.
- Thielecke: thunk and force constructs in CBV with callcc.
Antecedents: CBN without $\eta$-law for functions

- Plotkin: CPS transform for CBN without $\eta$-law.
- Abramsky and Ong: untyped lazy $\lambda$-calculus.
- Ong: typed lazy $\lambda$-calculus.
- Moggi: monads for CBN without $\eta$-law.
Antecedents: CBN with $\eta$-law for functions

- Plotkin’s PCF, a CBN calculus for recursion.
- Hennessy and Ashcroft: recursion and nondeterminism.
- O’Hearn: semantics of conditional in Reynolds’ Idealized Algol.
- Streicher and Reus: semantics of control effects in CBN.
- $\neg\neg$ translations of classical logic. Also CBV.
- Game semantics. Also CBV.
Various calculi based on CPS.

Filinski’s Effect-PCF provided the CBPV to construct. $U$ is implicit.

Howard’s $\lambda^{\mu\nu}_\bot$ calculus for recursion. $U$ is implicit.

Egger, Møgelberg and Simpson’s Effect Calculus emphasizes connection to Benton’s Linear/Nonlinear Logic. $U$ is implicit.

Marz’ SFPL for recursion and sequentiality. $F$ is implicit.

Nygaard and Winskel’s HOPLA for recursion and nondeterminism. $F$ is implicit.

Laurent’s LLP has extra type constructors not included in CBPV.

Harper, Licata and Zeilberger’s Polarized Intuitionistic Logic.
An operational semantics due to Felleisen and Friedman (1986). And Landin, Krivine, Streicher and Reus, Bierman, Pitts, …

It is suitable for sequential languages whether CBV, CBN or CBPV. At any time, there’s a computation (C) and a stack of contexts (K). Initially, K is empty.

Some authors make K into a single context, called an “evaluation context”.

Paul Blain Levy (University of Birmingham)
To evaluate \( M \) to \( x. N \): evaluate \( M \). If it returns \textbf{return} \( V \), then evaluate \( N[V/x] \).

\[
\begin{array}{c}
M \text{ to } x. N \quad K \quad \Rightarrow \\
M \quad \text{to } x. N :: K
\end{array}
\]

\[
\begin{array}{c}
\text{return } V \quad \text{to } x. N :: K \quad \Rightarrow \\
N[V/x] \quad K
\end{array}
\]
Transitions for application

To evaluate $V'M$: evaluate $M$. If it returns $\lambda x.N$, evaluate $N[V/x]$.

\[
\begin{array}{|c|c|c|}
\hline
V'M & K & \Rightarrow \\
\hline
M & V :: K & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\lambda x.N & V :: K & \Rightarrow \\
\hline
N[V/x] & K & \\
\hline
\end{array}
\]
Those function rules again

\[
\begin{array}{c|c|c}
V \cdot M & K & \leadsto \\
M & V :: K & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\lambda x. N & V :: K & \leadsto \\
N[V/x] & K & \\
\end{array}
\]
Those function rules again

\[
\begin{array}{c}
V'M & \quad K \\
M & \quad V :: K \\
\end{array}
\]

\[
\begin{array}{c}
\lambda x.N & \quad V :: K \\
N[V/x] & \quad K \\
\end{array}
\]

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

Those function rules again

\[
\begin{array}{c}
V'M & K \\
M & V :: K \\
\end{array} \quad \rightsquigarrow
\]

\[
\begin{array}{c}
\lambda x. N & V :: K \\
N[V/x] & K \\
\end{array} \quad \rightsquigarrow
\]

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

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

Revisiting some equations:

\[
V' \lambda x. M = M[V/x]
\]

\[
M = \lambda x. x' M \quad (x \text{ fresh})
\]

\[
\lambda x. \text{error } e = \text{error } e
\]

\[
\lambda x. \text{print } c. M = \text{print } c. \lambda x.M
\]
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 of type $A$
  then behaves in $B$.
- A computation of type $\prod_{i \in I} B_i$
  pops a tag $i \in I$
  then behaves in $B_i$. 
A stack consists of

- **arguments** that are values
- **arguments** that are tags
- **frames** taking the form \( \text{to } x. \ N \).
Example program of type $F\text{nat}$

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 to evaluate $\Gamma \vdash^c P : C$

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$P$</th>
<th>$C$</th>
<th>nil</th>
<th>$C$</th>
</tr>
</thead>
</table>

Transitions

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$M$ to $x$. $N$</th>
<th>$B$</th>
<th>$K$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$</td>
<td>$M$</td>
<td>$FA$</td>
<td>to $x$. $N :: K$</td>
<td>$C$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>return $V$</th>
<th>$FA$</th>
<th>to $x$. $N :: K$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$</td>
<td>$N[V/x]$</td>
<td>$B$</td>
<td>$K$</td>
<td>$C$</td>
</tr>
</tbody>
</table>

Typically $\Gamma$ would be empty and $C = F \text{bool}$.
Typing the CK-machine

Initial configuration to evaluate $\Gamma \vdash^c P : C$

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$P$</th>
<th>$C$</th>
<th>nil</th>
<th>$C$</th>
</tr>
</thead>
</table>

Transitions

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$M \text{ to } x. \ N$</th>
<th>$B$</th>
<th>$K$</th>
<th>$C$</th>
<th>$\rightsquigarrow$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$M$</th>
<th>$FA$</th>
<th>$\text{to } x. \ N :: K$</th>
<th>$C$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$\text{return } V$</th>
<th>$FA$</th>
<th>$\text{to } x. \ N :: K$</th>
<th>$C$</th>
<th>$\rightsquigarrow$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$N[V/x]$</th>
<th>$B$</th>
<th>$K$</th>
<th>$C$</th>
</tr>
</thead>
</table>

Typically $\Gamma$ would be empty and $C = F\text{bool}$. We write $\Gamma \vdash^k B K : C$ to mean that $K$ can accompany a computation of type $B$ during evaluation.
Typing rules, read off from the CK-machine

Typing a stack

\[
\frac{}{\Gamma \mid C \vdash^k \text{n} \text{i} \text{l} : C} \\
\frac{}{\Gamma \mid B^\hat{i} \vdash^k K : C} \\
\frac{}{\Gamma \mid \prod_{i \in I} B^i \vdash^k \hat{i} : K : C} \\
\frac{}{\Gamma, x : A \vdash^c M : B} \\
\frac{}{\Gamma \mid B \vdash^k K : C} \\
\frac{}{\Gamma \mid FA \vdash^k \text{to} \ x. \ M :: K : C} \\
\frac{}{\Gamma \vdash^v V : A} \\
\frac{}{\Gamma \mid B \vdash^k K : C} \\
\frac{}{\Gamma \mid A \rightarrow B \vdash^k V :: K : C}
\]
Typing rules, read off from the CK-machine

Typing a stack

\[ \Gamma \vdash C \vdash^k \text{nil} : C \]
\[ \Gamma \vdash B_{\hat{i}} \vdash^k K : C \quad \hat{i} \in I \]
\[ \Gamma \vdash \prod_{i \in I} B_i \vdash^k \hat{i} :: K : C \]
\[ \Gamma, x : A \vdash^c M : B \quad \Gamma \mid B \vdash^k K : C \]
\[ \Gamma \mid FA \vdash^k \text{to } x. M :: K : C \]
\[ \Gamma \vdash^v V : A \quad \Gamma \mid B \vdash^k K : C \]
\[ \Gamma \mid A \rightarrow B \vdash^k V :: K : C \]

Typing a CK-configuration

\[ \Gamma \vdash^c M : B \quad \Gamma \mid B \vdash^k K : C \]
\[ \Gamma \vdash^{ck} (M, K) : C \]
A **continuation** is a stack from an $F$ type. For example:

$$\Gamma | F A \vdash^k \text{to } x. \; M :: K : C$$

It describes what happens next once it receives a value.
Special Stacks

Continuations

A continuation is a stack from an $F$ type. For example:

$$\Gamma \mid FA \vdash^k \text{to} x. \ M :: K : C$$

It describes what happens next once it receives a value. In CBV, all computations have $F$ type, so all stacks are continuations.
Special Stacks

Continuations
A **continuation** is a stack from an $F$ type. For example:

$$\Gamma \mid FA \vdash^k \text{to } x. \ M :: K : C$$

It describes what happens next once it receives a value. In CBV, all computations have $F$ type, so all stacks are continuations.

Top-Level Stack
The **top-level stack** is

$$\Gamma \mid C \vdash^k \text{nil} : C$$

and $C$ is the **top-level type**.
Continuations

A **continuation** is a stack from an $F$ type. For example:

$$
\Gamma \mid FA \vdash^k \text{x. } M :: K : C
$$

It describes what happens next once it receives a value.
In CBV, all computations have $F$ type, so all stacks are continuations.

Top-Level Stack

The **top-level stack** is

$$
\Gamma \mid C \vdash^k \text{nil} : C
$$

and $C$ is the **top-level type**.
If $C$ is an $F$ type, then nil is the **top-level continuation**: it receives a value and returns it to the user.
Suppose $[C] = (Y, \phi)$. The behaviour of $\Gamma \vdash^{ck} (M, K) : C$ depends on the environment:

$$
[\Gamma] \xrightarrow{[\langle M, K \rangle]} Y
$$

to be preserved by each transition.
Suppose $\llbracket C \rrbracket = (Y, \phi)$. The behaviour of $\Gamma \vdash^{\text{ck}} (M, K) : C$ depends on the environment:

\[
\begin{array}{c}
\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (M,K) \rrbracket} Y
\end{array}
\]

to be preserved by each transition.

Suppose also $\llbracket B \rrbracket = (X, \theta)$. A stack $\Gamma \mid B \vdash^k K : C$ transforms computations to CK-configurations. So we get a function

\[
\begin{array}{c}
\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y
\end{array}
\]
	homomorphic in its second argument.
Denotational semantics of stacks

Suppose \([C] = (Y, \phi)\). The behaviour of \(\Gamma \vdash^{ck} (M, K) : C\) depends on the environment:

\[
[\Gamma] \xrightarrow{[\Gamma, K]} Y
\]

to be preserved by each transition.

Suppose also \([B] = (X, \theta)\). A stack \(\Gamma | B \vdash^{k} K : C\) transforms computations to CK-configurations. So we get a function

\[
[\Gamma] \times X \xrightarrow{[K]} Y
\]

homomorphic in its second argument.

because if \(M\) raises an error or prints, then so does \(M, K\).
Denotational semantics of stacks

Suppose $\lbrack C \rbrack = (Y, \phi)$. The behaviour of $\Gamma \vdash^{ck} (M, K) : C$ depends on the environment:

$$
\begin{array}{c}
\lbrack \Gamma \rbrack \xrightarrow{\lbrack (M, K) \rbrack} Y
\end{array}
$$

to be preserved by each transition.

Suppose also $\lbrack B \rbrack = (X, \theta)$. A stack $\Gamma \mid B \vdash^K K : C$ transforms computations to CK-configurations. So we get a function

$$
\begin{array}{c}
\lbrack \Gamma \rbrack \times X \xrightarrow{\lbrack K \rbrack} Y
\end{array}
$$

homomorphic in its second argument.

because if $M$ raises an error or prints, then so does $M, K$.

We assume there’s no exception handling.
We have an adjunction between the category of values (sets and functions) and the category of stacks ($T$-algebras and homomorphisms).

\[
\begin{array}{c}
\text{Set} \\ \downarrow \scriptstyle{\perp} \\ \text{Set}^T \\
\end{array}
\xrightarrow{F^T} \xleftarrow{U^T}
\]

This resolves the monad $T$ on $\text{Set}$. 
Consider CBPV extended with two storage cells: 
1 stores a natural number, and \( 1' \) stores a boolean.
State

Consider CBPV extended with two storage cells: $l$ stores a natural number, and $l'$ stores a boolean.

\[
\begin{align*}
\Gamma \vdash^c M : B \\
\Gamma \vdash^c l := n. M : B \\
\Gamma \vdash^c M_n : B \quad (\forall n \in \mathbb{N}) \\
\Gamma \vdash^c \text{read } l \text{ as } \{n. M_n\}_{n \in \mathbb{N}} : B
\end{align*}
\]
Consider CBPV extended with two storage cells: \( l \) stores a natural number, and \( l' \) stores a boolean.

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

A state is \( l \mapsto n, l' \mapsto b \).

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.

We can type these using

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

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:

$$\text{If } s, M \Downarrow s', T \text{ then } [s, M]_{\epsilon} = [s', T]_{\epsilon}$$

This requires an SC-configuration to have a denotation.
Semantics of SC-configurations

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 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 $B$.

The behaviour of an SC-configuration $\Gamma \vdash_{sc} (s, M) : B$ depends on the environment:

$$[\Gamma] \xrightarrow{[(s,M)]} [B]$$
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 $B$.

The behaviour of an SC-configuration $\Gamma \vdash^{sc} (s, M) : B$ depends on the environment:

$$[[\Gamma]] \xrightarrow{[[s, M]]} [[B]]$$

The behaviour of a computation $\Gamma \vdash^c M : B$ depends on the state and environment:

$$S \times [[\Gamma]] \xrightarrow{[[M]]} [[B]]$$
State: semantics of types

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 $\Gamma \vdash V : UB$ can be forced in any state $s$, giving an SC-configuration $s, \text{force } V$.

$$[UB] = S \rightarrow [B]$$
State: semantics of types

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 $s, \text{force } V$.

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

We recover standard semantics for CBV, and O’Hearn’s semantics for CBN.
The SCK-machine

We replace $\vdash^{ck}$ with $\vdash^{sck}$.

\[
\Gamma \vdash^c M : B \quad \Gamma \mid B \vdash^k K : C \\
\overline{\Gamma \vdash^{sck} (s, M, K) : C} \quad s \in S
\]
We replace $\vdash^{ck}$ with $\vdash^{sck}$.

\[
\begin{align*}
\Gamma \vdash^c M : B & \quad \Gamma \mid B \vdash^k K : C \\
\hline
\Gamma \vdash^{sck} (s, M, K) : C
\end{align*}
\]

$s \in S$

The behaviour of an SCK-configuration $\Gamma \vdash^{sck} (s, M, K) : C$ depends on the environment:

\[
\begin{array}{c}
\llbracket \Gamma \rrbracket \\
\xrightarrow{[(s,M,K)]} \\
\llbracket C \rrbracket
\end{array}
\]

to be preserved by each transition.
The SCK-machine

We replace $\vdash^{ck}$ with $\vdash^{sck}$.

$$\Gamma \vdash^c M : B \quad \Gamma \mid B \vdash^k K : C \quad s \in S$$

The behaviour of an SCK-configuration $\Gamma \vdash^{sck} (s, M, K) : C$ depends on the environment:

$$\llbracket \Gamma \rrbracket \xrightarrow{(s, M, K)} \llbracket C \rrbracket$$

to be preserved by each transition.

A stack $\Gamma \mid B \vdash^k K : C$ transforms SC-configuration behaviours to SCK-configuration behaviours:

$$\llbracket \Gamma \rrbracket \times \llbracket B \rrbracket \xrightarrow{[K]} \llbracket C \rrbracket$$
We’ve seen that a stack $\overline{\text{B} \vdash^k K : \text{C}}$ denotes a function $[\text{B}] \xrightarrow{[K]} [\text{C}]$. 
We’ve seen that a stack \( |B| \vdash^k K : C \) denotes a function \( [B] \xrightarrow{[K]} [C] \).

We have an adjunction

\[
\begin{array}{ccc}
\text{Set} & \xleftarrow{S \times -} & \xrightarrow{- \times S} & \text{Set} \\
\downarrow & & & & \\
S \rightarrow - & & & & S \rightarrow -
\end{array}
\]

between values and stacks.
Catching and throwing a stack

We extend CBPV with Crolard’s instructions for changing the stack.

- **catch** $\alpha$ means “let $\alpha$ be the current stack”.
- **throw** $K$ means “change the current stack to $K$”.

\[
\begin{array}{c}
\Gamma \quad \text{catch } \alpha. \ M \quad B \quad K \quad \Delta \quad \rightsquigarrow \\
\Gamma \quad M[K/\alpha] \quad B \quad K \quad \Delta
\end{array}
\]

\[
\begin{array}{c}
\Gamma \quad \text{throw } K. \ M \quad B' \quad K' \quad \Delta \quad \rightsquigarrow \\
\Gamma \quad M \quad B \quad K \quad \Delta
\end{array}
\]
Typing judgements for control

The stack context $\Delta$ consists of stack names $\alpha : B$.

$\Gamma \vdash^n \bot$ indicates “no top-level type”.

<table>
<thead>
<tr>
<th>Type</th>
<th>Judgement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>$\Gamma \vdash^v V : A</td>
</tr>
<tr>
<td>Computation</td>
<td>$\Gamma \vdash^c M : B</td>
</tr>
<tr>
<td>Stack</td>
<td>$\Gamma</td>
</tr>
<tr>
<td>CK-Configuration</td>
<td>$\Gamma</td>
</tr>
</tbody>
</table>

Creating nil at start of evaluation

Initial configuration to evaluate $\Gamma \vdash^c P : C | \Delta$

$\Gamma$ $P$ $C$ nil $\Delta$, nil : $C$

Typically $\Gamma$ and $\Delta$ would be empty and $C = F$ bool.
Fix a set $R$, the set of behaviours of CK-configurations.
Fix a set $R$, the set of behaviours of CK-configurations.

Moggi’s monad for control operators ("continuations") is $(\neg \to R) \to R.$
Fix a set \( R \), the set of behaviours of CK-configurations.

Moggi’s monad for control operators (“continuations”) is \((\rightarrow R) \rightarrow R\).

Maybe we can use algebras for this to build a denotational semantics of control.
Informally a computation type $[B]$ denotes the set of stacks from $B$. 
Informally a computation type $[B]$ denotes the set of stacks from $B$. The behaviour of a computation $\Gamma \vdash^c M : B \mid \Delta$ depends on the environment, stack environment and current stack:

$$[\Gamma] \times [\Delta] \times [B] \xrightarrow{[M]} R$$
Informally a computation type $[B]$ denotes the set of stacks from $B$.

The behaviour of a computation $\Gamma \vdash^c M : B | \Delta$ depends on the environment, stack environment and current stack:

$$[\Gamma] \times [\Delta] \times [B] \xrightarrow{[M]} R$$

A value $\Gamma \vdash^v V : A | \Delta$ denotes

$$[\Gamma] \times [\Delta] \xrightarrow{[V]} [A]$$

A stack $\Gamma \vdash^n k K | \Delta$ denotes

$$[\Gamma] \times [\Delta] \xrightarrow{[K]} [B]$$

A CK-configuration $\Gamma \vdash^{nck} \text{nil.}(M, K) : \Delta$ denotes

$$[\Gamma] \times [\Delta] \xrightarrow{[(M, K)]} R$$

to be preserved by each transition.
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.

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

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

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

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.

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

We recover standard continuation semantics for CBV, and Streicher and Reus’ semantics for CBN.
Control: the value/stack adjunction

A stack with top-level type

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

denotes a function

\[ [\Gamma] \times [\Delta] \times [C] \xrightarrow{[\alpha. K]} [B] \]
A stack with top-level type

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

denotes a function

\[
[\Gamma] \times [\Delta] \times [C] \xrightarrow{[\alpha. K]} [B]
\]

So we have an adjunction

\[
\text{Set} \xrightarrow{\rightarrow \mathcal{R}} \text{Set}^{\text{op}} \xleftarrow{\rightarrow \mathcal{R}} \text{Set}
\]

between values and stacks with top-level type.
Summary of models

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

\[
\begin{array}{c}
\text{Set} \\
\downarrow \quad \perp \\
\uparrow \quad U^T \\
\text{Set}^T
\end{array}
\]

\[F^T \quad \quad \quad \quad \quad U^T \]

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

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

$$\begin{array}{ccc}
\text{Set} & \overset{F^T}{\longrightarrow} & \text{Set}^T \\
\bot & \Downarrow & \Uparrow
\end{array}$$

This is useful for modelling CBPV with errors and printing.

For a set $S$ we have an adjunction

$$\begin{array}{ccc}
\text{Set} & \overset{S \times -}{\longrightarrow} & \text{Set} \\
\bot & \Downarrow & \Uparrow
\end{array}$$

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

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

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

This is useful for modelling CBPV with errors and printing.

For a set $S$ we have an adjunction

$$\text{Set} \xrightarrow{S \times - \bot} \text{Set} \xleftarrow{S \rightarrow}$$

This is useful for modelling CBPV with state.

For a set $R$ we have an adjunction

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

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

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

\[
\text{Set} \quad \quad \text{Set}^T
\]

\[
\begin{array}{c}
F^T \\
\downarrow \\
U^T
\end{array}
\]

This is useful for modelling CBPV with errors and printing.

For a set $S$ we have an adjunction

\[
\text{Set} \quad \quad \text{Set}
\]

\[
\begin{array}{c}
S \times - \\
\downarrow \\
S \rightarrow -
\end{array}
\]

This is useful for modelling CBPV with state.

For a set $R$ we have an adjunction

\[
\text{Set} \quad \quad \text{Set}^{\text{op}}
\]

\[
\begin{array}{c}
- \rightarrow R \\
\downarrow \\
- \rightarrow R
\end{array}
\]

This is useful for modelling CBPV with control.