λ-calculus, effects and call-by-push-value

Paul Blain Levy

University of Birmingham

April 16, 2021
Outline

1 Pure λ-calculus
   - Syntax
   - Denotational semantics
   - The $\beta\eta$-theory
   - Reversible rules
   - Operational semantics

2 Adding Effects
   - Outline
   - Errors and printing, operationally

3 Call-by-value with errors
   - Denotational semantics
   - Substitution and values
   - Fine-grain call-by-value

4 Call-by-name with errors

5 Call-by-push-value

6 Stacks

7 State

8 Control
Types

We’re going to look at simply typed λ-calculus with arithmetic, including not just function types, but also sum and product types.

Here is the syntax of types:

\[ A ::= \text{bool} \mid \text{nat} \mid A \rightarrow A \mid 1 \mid A \times A \mid 0 \mid A + A \mid \sum_{i \in \mathbb{N}} A_i \mid \prod_{i \in \mathbb{N}} A_i \quad \text{(optional extra)} \]
We’re going to look at simply typed $\lambda$-calculus with arithmetic, including not just function types, but also sum and product types.

Here is the syntax of types:

$$A ::= \text{bool} \mid \text{nat} \mid A \rightarrow A \mid 1 \mid A \times A \mid 0 \mid A + A$$

$$\mid \sum_{i \in \mathbb{N}} A_i \mid \prod_{i \in \mathbb{N}} A_i \quad (\text{optional extra})$$

Why no brackets?

- You might expect $A ::= \cdots \mid (A)$.
- But our definition is abstract syntax.
- This means a type—or a term—is a tree of symbols, not a string of symbols.
Typing Judgement

Example

\[ x : \text{nat}, \ y : \text{nat} \vdash \lambda z_{\text{nat} \rightarrow \text{nat}}. \ z (x + x) : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \]

In English:

Given declarations of \( x : \text{nat} \) and \( y : \text{nat} \),

\( \lambda z_{\text{nat} \rightarrow \text{nat}}. \ z (x + x) \) is a term of type \((\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}\).

The typing judgement takes the form \( \Gamma \vdash M : A \).

- \( \Gamma \) is a typing context, a finite set of typed distinct identifiers.
- \( M \) is a term.
- \( A \) is a type.
Identifiers

The most basic typing rules, not associated with any particular type.

Free identifier

\[ \Gamma \vdash x : A \] \in \Gamma

Multiple local declaration, e.g. of two identifiers

\[ \Gamma \vdash M : A \quad \Gamma \vdash M' : B \quad \Gamma, x : A, y : B \vdash N : C \]

\[ \Gamma \vdash \text{let} \ (x \ \text{be} \ M, \ y \ \text{be} \ M'). \ N : C \]
Typing rules for $A \rightarrow B$

Introduction rule

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x_A. M : A \rightarrow B}
\]

Elimination rule

\[
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}
\]

Type annotations in terms

- For $\Gamma$ and $M$, there’s at most one $A$ such that $\Gamma \vdash M : A$
- and at most one derivation of $\Gamma \vdash M : A$.
- This is because of our type annotations.
- Some formulations omit some or all of these.
Typing rules for bool

Two introduction rules:

\[ \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{false} : \text{bool} \]

Elimination rule

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

It’s a pretentious notation for if \( M \) then \( N \) else \( N' \).
Typing rules for arithmetic

These are *ad hoc* rules.

\[
\frac{}{\Gamma \vdash 17 : \text{nat}} \quad \frac{\Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat}}{\Gamma \vdash M + M' : \text{nat}}
\]
Typing rules for $A + B$

Two introduction rules

\[
\begin{align*}
\Gamma &\vdash M : A \\
\Gamma &\vdash \text{inl}^{A, B} M : A + B \\
\Gamma &\vdash M : B \\
\Gamma &\vdash \text{inr}^{A, B} M : A + B
\end{align*}
\]

Elimination rule

\[
\begin{align*}
\Gamma &\vdash M : A + B \\
\Gamma, x : A &\vdash N : C \\
\Gamma, y : B &\vdash N' : C
\end{align*}
\]

\[
\Gamma \vdash \text{match } M \text{ as } \{ \text{inl } x. N, \text{ inr } y. N' \} : C
\]
Typing rules for $A + B$

Two introduction rules

\[
\begin{align*}
\Gamma & \vdash M : A \\
\Gamma & \vdash \text{inl}^{A,B} M : A + B \\
\Gamma & \vdash M : B \\
\Gamma & \vdash \text{inr}^{A,B} M : A + B
\end{align*}
\]

Elimination rule

\[
\begin{align*}
\Gamma & \vdash M : A + B \\
\Gamma, x : A & \vdash N : C \\
\Gamma, y : B & \vdash N' : C \\
\Gamma & \vdash \text{match} \ M \ \text{as} \ \{ \text{inl} \ x. \ N, \ \text{inr} \ y. \ N' \} : C
\end{align*}
\]

Likewise for $\sum_{i \in \mathbb{N}} A_i$. 
Typing rules for 0

Zero introduction rules

Elimination rule

\[ \Gamma \vdash M : 0 \]

\[ \Gamma \vdash \text{match } M \text{ as } \{\}^A : A \]
**Typing rules for** \( A \times B \)

**Introduction rule**

\[
\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}
\]

**Two options for elimination**

- **Pattern-matching product.** Elimination rule

\[
\frac{\Gamma \vdash M : A \times B \quad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \text{match } M \text{ as } \langle x, y \rangle. \ N : C}
\]

- **Projection product.** Two elimination rules

\[
\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M^1 : A} \quad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M^r : B}
\]
Typing rules for $A \times B$

**Introduction rule**

\[
\Gamma \vdash M : A \quad \Gamma \vdash N : B \\
\hline
\Gamma \vdash \langle M, N \rangle : A \times B
\]

**Two options for elimination**

- **Pattern-matching product.** Elimination rule

\[
\Gamma \vdash M : A \times B \quad \Gamma, x : A, y : B \vdash N : C \\
\hline
\Gamma \vdash \text{match } M \text{ as } \langle x, y \rangle. \ N : C
\]

- **Projection product.** Two elimination rules

\[
\begin{align*}
\Gamma \vdash M : A \times B & \\
\hline
\Gamma \vdash M^1 : A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : A \times B & \\
\hline
\Gamma \vdash M^r : B
\end{align*}
\]

$\prod_{i \in \mathbb{N}} A_i$ is a projection product.
Typing rules for 1

Introduction rule

\[ \Gamma \vdash \langle \rangle : 1 \]

Two options for elimination

- **Pattern-match unit.** Elimination rule
  \[\Gamma \vdash M : 1 \quad \Gamma \vdash N : C \quad \Gamma \vdash \text{match } M \text{ as } \langle \rangle . N : C\]

- **Projection unit.** Zero elimination rules
Weakening is admissible

**Theorem**

If $\Gamma \vdash M : A$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash M : A$. 
Terms are $\alpha$-equivalent when they have the same binding diagram.

$$M \equiv_\alpha N \iff \text{BD}(M) = \text{BD}(N)$$

- The collection of binding diagrams forms an initial algebra [FPT; AR].
- We’ll skate over this issue. It’s not specific to $\lambda$-calculus.
Substitution is an operation on binding diagrams, not on terms.

Example

\[ M = \lambda y. \text{nat}. y + 3 \]
\[ M' = 7 \]
\[ N = x(5 + y) \]
\[ N[\lambda y. \text{nat}. y + 3/5 + 7] = (\lambda z. \text{nat}. z + 3)(5 + 7) \]
Substitution is an operation on binding diagrams, not on terms.

**Multiple substitution, e.g. for two identifiers**

If $\Gamma \vdash M : A$ and $\Gamma \vdash M' : B$ and $\Gamma, x : A, y : B \vdash N : C$, we define $\Gamma \vdash N[M/x, M'/y] : C$.

**Example**

\[
\begin{align*}
M &= \lambda y_{\text{nat}}. y + 3 \\
M' &= 7 \\
N &= x (5 + y) \\
N[M/x, M'/y] &= (\lambda z_{\text{nat}}. z + 3) (5 + 7)
\end{align*}
\]
- Every type $A$ denotes a set $[A]$. 

- For example, $[\text{nat} \rightarrow \text{nat}]$ is the set of functions $\mathbb{N} \rightarrow \mathbb{N}$. 
Types denote sets

- Every type \( A \) denotes a set \([A]\).
- For example, \([\text{nat} \to \text{nat}]\) is the set of functions \( \mathbb{N} \to \mathbb{N} \).
- \([A]\) is a semantic domain for terms of type \( A \).
- This means: a closed term of type \( \vdash M : A \) denotes an element of \([A]\).
Types denote sets

- Every type $A$ denotes a set $[A]$.
- For example, $[\text{nat} \rightarrow \text{nat}]$ is the set of functions $\mathbb{N} \rightarrow \mathbb{N}$.
- $[A]$ is a **semantic domain** for terms of type $A$.
- This means: a closed term of type $\vdash M : A$ denotes an element of $[A]$.
- For example, $\lambda x_{\text{nat}}. x + 3$ denotes $\lambda a \in \mathbb{N}. a + 3$. 
Notation

For sets $X$ and $Y$,

- $X \rightarrow Y$ is the set of functions from $X$ to $Y$.
- $X \times Y$ is $\{\langle x, y \rangle \mid x \in X, y \in Y\}$.
- $X + Y$ is $\{\text{inl } x \mid x \in X\} \cup \{\text{inr } y \mid y \in Y\}$.

\[
\begin{align*}
[\text{bool}] &= \mathbb{B} = \{\text{true, false}\} \\
[\text{nat}] &= \mathbb{N} \\
[A \rightarrow B] &= [A] \rightarrow [B] \\
[1] &= 1 = \{\langle \rangle \} \\
[A + B] &= [A] + [B] \\
[A \times B] &= [A] \times [B] \\
[0] &= \emptyset
\end{align*}
\]
Let $\Gamma$ be a typing context.

- A **semantic environment** $\rho$ for $\Gamma$ provides an element $\rho_x \in [A]$ for each $(x : A) \in \Gamma$.
- $[\Gamma]$ is the set of semantic environments for $\Gamma$.

$$[\Gamma] \overset{\text{def}}{=} \prod_{(x:A) \in \Gamma} [A]$$
Semantics of typing judgement

Given a typing judgement $\Gamma \vdash M : A$, we shall define $[M]$, or more precisely $[\Gamma \vdash M : A]$. It’s a function from $[\Gamma]$ to $[A]$.

**Example**

$x : \text{nat}, y : \text{nat} \vdash \lambda z_{\text{nat}} \rightarrow_{\text{nat}} z(x + y) : (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}$

denotes the function

$$[x : \text{nat}, y : \text{nat}] \rightarrow (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$$

$$\rho \mapsto \lambda z \in \mathbb{N} \rightarrow \mathbb{N}. z(\rho_x + \rho_y)$$
Semantics of terms

\[ \Gamma \vdash 17 : \text{nat} \]
\[ \llbracket 17 \rrbracket : \rho \mapsto 17 \]

\[ \Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat} \]
\[ \Gamma \vdash M + M' : \text{nat} \]
\[ \llbracket M + M' \rrbracket : \rho \mapsto \llbracket M \rrbracket \rho + \llbracket M' \rrbracket \rho \]
More semantic equations

\[ (x : A) \in \Gamma \]

\[ [x] : \rho \mapsto \rho_x \]

\[ \Gamma, x : A \vdash M : B \]

\[ \Gamma \vdash \lambda x_A. M : A \rightarrow B \]

\[ [\lambda x_A. M] : \rho \mapsto \lambda a \in [A]. [M](\rho, x \mapsto a) \]
More semantic equations

\[
\Gamma \vdash M : A \\
\Gamma \vdash \text{inl}^{A,B} M : A + B \\
[[\text{inl}^{A,B} M]] : \rho \mapsto \text{inl} [[M]\rho]
\]

\[
\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash N' : C \\
\Gamma \vdash \text{match } M \text{ as } \{\text{inl } x. N, \text{inr } y. N'\} : C
\]

\[
[[\text{match } M \text{ as } \{\text{inl } x. N, \text{inr } y. N'\}]] : \rho \mapsto \text{match } [[M]\rho \text{ as } \{\text{inl } a. [[N](\rho, x \mapsto a), \text{inr } b. [[N'](\rho, y \mapsto b)\}}
\]
Basic properties

Semantic Coherence

If type annotations are omitted, then $\Gamma \vdash M : A$ can have more than one derivation.

We must prove that $\llbracket \Gamma \vdash M : A \rrbracket$ doesn’t depend on the derivation.
Basic properties

**Semantic Coherence**

If type annotations are omitted, then $\Gamma \vdash M : A$ can have more than one derivation.

We must prove that $\llbracket \Gamma \vdash M : A \rrbracket$ doesn’t depend on the derivation.

**Weakening Lemma**

If $\Gamma \vdash M : A$ and $\Gamma \subseteq \Gamma'$ then

$$\llbracket \Gamma' \vdash M : A \rrbracket \rho = \llbracket \Gamma \vdash M \rrbracket (\rho \upharpoonright \Gamma)$$
Substitution

We can give denotational semantics of binding diagrams.

\[ [\Gamma \vdash M : A] = [\Gamma \vdash BD(M) : A] \]

So \( \alpha \)-equivalent terms have the same denotation.
We can give denotational semantics of binding diagrams.

\[ [\Gamma \vdash M : A] = [\Gamma \vdash \text{BD}(M) : A] \]

So \( \alpha \)-equivalent terms have the same denotation.

Substitution Lemma

For binding diagrams \( \Gamma \vdash M : A \) and \( \Gamma \vdash M' : B \) and \( \Gamma, x : A \vdash N : C \),
we can recover \([N[M/x, M'/y]]\) from \([N]\) and \([M]\) and \([M']\).

\[ [N[M/x, M'/y]] : \rho \longmapsto [N](\rho, x \longmapsto [M]\rho, y \longmapsto [M']\rho) \]
The $\beta$-law for $A \to B$

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

Introduction inside an elimination may be removed.
The $\beta$-law for $A \to B$

$$\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : B$$

$$\Gamma \vdash (\lambda x_A. N) M = N[M/x] : B$$

Introduction inside an elimination may be removed.

Two $\beta$-laws for projection product $A \times B$

$$\Gamma \vdash M : A \quad \Gamma \vdash N : A'$$

$$\Gamma \vdash \langle M, N \rangle^1 = M : A$$

Zero $\beta$-laws for projection unit 1
More $\beta$-laws

Two $\beta$-laws for $\text{bool}$

\[
\begin{align*}
\Gamma \vdash N : C & \quad \Gamma \vdash N' : C \\
\Gamma \vdash \text{match true as } \{\text{true. } N, \text{false. } N'\} & = N : C
\end{align*}
\]
Two $\beta$-laws for bool

\[
\begin{align*}
\Gamma \vdash N : C & \quad \Gamma \vdash N' : C \\
\hline
\Gamma \vdash \text{match true as } \{ \text{true. } N, \text{ false. } N' \} = N : C
\end{align*}
\]

Two $\beta$-laws for $A + B$

\[
\begin{align*}
\Gamma \vdash M : A & \quad \Gamma, x : A \vdash N : C & \quad \Gamma, y : B \vdash N' : C \\
\hline
\Gamma \vdash \text{match } \text{inl}^{A,B} M \text{ as } \{ \text{inl } x. N, \text{ inr } y. N' \} = N[M/x] : C
\end{align*}
\]
More $\beta$-laws

Two $\beta$-laws for bool

$$
\Gamma \vdash N : C \quad \Gamma \vdash N' : C
$$

$$
\Gamma \vdash \text{match true as } \{\text{true. } N, \text{ false. } N'\} = N : C
$$

Two $\beta$-laws for $A + B$

$$
\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash N' : C
$$

$$
\Gamma \vdash \text{match inl}^{A,B} M \text{ as } \{\text{inl } x. \, N, \text{ inr } y. \, N'\} = N[M/x] : C
$$

Zero $\beta$-laws for 0
\[
\begin{align*}
\Gamma \vdash M : A & \quad \Gamma \vdash M' : B \quad \Gamma, x : A, y : B \vdash N : C \\
\Gamma \vdash \text{let } (x \text{ be } M, \ y \text{ be } M'). \ N = N[M/x, M'/y] & : C
\end{align*}
\]
**η-laws**

η-law for $A \rightarrow B$, everything is $\lambda$

$$\Gamma \vdash M : A \rightarrow B$$

$$\Gamma \vdash M = \lambda x_{A}. M \, x : A \rightarrow B \quad x \notin \Gamma$$

Introduction outside an elimination may be inserted.
$$\eta$$-laws

$$\eta$$-law for \(A \to B\), everything is \(\lambda\)

\[
\Gamma \vdash M : A \to B \\
\frac{}{\Gamma \vdash M = \lambda x : A. M x : A \to B} \\
x \notin \Gamma
\]

Introduction outside an elimination may be inserted.

$$\eta$$-law for projection product \(A \times B\), everything is a tuple

\[
\Gamma \vdash M : A \times B \\
\frac{}{\Gamma \vdash M = \langle M^1, M^r \rangle : A \times B}
\]

$$\eta$$-law for projection unit \(1\), everything is a tuple

\[
\Gamma \vdash M : 1 \\
\frac{}{\Gamma \vdash M = \langle \rangle : 1}
\]
More $\eta$-laws

$\eta$-law for bool, everything is true or false

\[
\frac{\Gamma \vdash M : \text{bool} \quad \Gamma, z : \text{bool} \vdash N : C}{\Gamma \vdash N[M/z] = \text{match } M \text{ as } \{N[\text{true}/z], N[\text{false}/z]\} : C} \quad z \notin \Gamma
\]
More $\eta$-laws

$\eta$-law for bool, everything is true or false

$$
\Gamma \vdash M : \text{bool} \quad \Gamma, z : \text{bool} \vdash N : C
$$

$$\Gamma \vdash N[M/z] = \text{match } M \text{ as } \{N[\text{true}/z], N[\text{false}/z]\} : C, \quad z \notin \Gamma
$$

$\eta$-law for $A + B$, everything is inl or inr

$$
\Gamma \vdash M : A + B \quad \Gamma, z : A + B \vdash N : C
$$

$$\Gamma \vdash N[M/z] = \text{match } M \text{ as } \{\text{inl } x. N[\text{inl } x/z], \text{inr } y. N[\text{inr } y/z]\} : C, \quad z \notin \Gamma
$$
More $\eta$-laws

$\eta$-law for $\text{bool}$, everything is true or false

$$\begin{align*}
\Gamma \vdash M : \text{bool} & \quad \Gamma, z : \text{bool} \vdash N : C \\
\Gamma \vdash N[M/z] = \text{match } M \text{ as } \{N[\text{true}/z], N[\text{false}/z]\} : C
\end{align*}$$ $z \notin \Gamma$

$\eta$-law for $A + B$, everything is $\text{inl}$ or $\text{inr}$

$$\begin{align*}
\Gamma \vdash M : A + B & \quad \Gamma, z : A + B \vdash N : C \\
\Gamma \vdash N[M/z] = \\
\text{match } M \text{ as } \{\text{inl } x. N[\text{inl } x/z], \text{inr } y. N[\text{inr } y/z]\} : C
\end{align*}$$ $z \notin \Gamma$

$\eta$-law for $0$, nothing exists

$$\begin{align*}
\Gamma \vdash M : 0 & \quad \Gamma, z : 0 \vdash N : C \\
\Gamma \vdash N[M/z] = \text{match } M \text{ as } \{} : C
\end{align*}$$ $z \notin \Gamma$
We define $\Gamma \vdash M =_{\beta\eta} M' : A$ inductively as follows.

All the $\beta$- and $\eta$-laws are taken as axioms, and it is a congruence i.e. an equivalence relation preserved by each term constructor. For example:

\[
\Gamma \vdash \lambda x : A. M = \lambda x : A. M' : A \to B
\]

\[
\Gamma, x : A \vdash M = M' : B
\]

\[
\Gamma \vdash \lambda x_A. M = \lambda x_A. M' : A \to B
\]
Properties of $\equiv_{\beta\eta}$

**Closure Theorems**
- $\equiv_{\beta\eta}$ is closed under weakening. But not conversely, e.g.

  $$z : 0 \vdash \text{true} \equiv_{\beta\eta} \text{false} : \text{bool}$$

  but not $$\vdash \text{true} \equiv_{\beta\eta} \text{false} : \text{bool}$$

- $\equiv_{\beta\eta}$ is closed under substitution.

**Soundness theorem**

If $\Gamma \vdash M \equiv_{\beta\eta} M' : A$ then $[M] = [M']$.

Follows from the weakening and substitution lemmas.
The connective $\rightarrow$ is rightist: it has a reversible rule

$$
\Gamma, x : A \vdash B \\
\hline
\Gamma \vdash A \rightarrow B
$$

natural in $\Gamma$—we’ll skate over naturality.
Reversible rule for $A \to B$

The connective $\to$ is rightist: it has a reversible rule

$$
\begin{array}{c}
\Gamma, x : A \vdash B \\
\hline
\Gamma \vdash A \to B
\end{array}
$$

natural in $\Gamma$—we’ll skate over naturality.

- Downwards, a term $\Gamma, x : A \vdash M : B$ is sent to $\lambda x_A. M$.
- Upwards, a term $\Gamma \vdash N : A \to B$ is sent to $N x$.
- These are inverse up to $=\beta\eta$. 

Reversible rule for $A \to B$

The connective $\to$ is rightist: it has a reversible rule

$$
\Gamma, x : A \vdash B \\
\frac{}{\Gamma \vdash A \to B}
$$

natural in $\Gamma$—we’ll skate over naturality.

- Downwards, a term $\Gamma, x : A \vdash M : B$ is sent to $\lambda x_A. M$.
- Upwards, a term $\Gamma \vdash N : A \to B$ is sent to $N x$.
- These are inverse up to $=_{\beta\eta}$.

$A \to B$ appears on the right of $\vdash$ in the conclusion.
Reversible rule for bool

The (nullary) connective bool is leftist. That means: it has a reversible rule

$$\begin{array}{c}
\Gamma \vdash C \quad \Gamma \vdash C \\
\hline
\Gamma, z : \text{bool} \vdash C
\end{array}$$

natural in $\Gamma$ and $C$—we’ll skate over naturality.

- Downwards, a pair $\Gamma \vdash M : C$ and $\Gamma \vdash M' : C$ is sent to match $z$ as $\{\text{true}.M, \text{false}.M'\}$.
- Upwards, a term $\Gamma, z : \text{bool} \vdash N : C$ is sent to $N[\text{true}/z]$ and $N[\text{false}/z]$.
- These are inverse up to $=_{\beta\eta}$.

bool appears on the left of $\vdash$ in the conclusion.
Reversible rule for $A + B$

The connective $+$ is leftist, having a reversible rule

\[
\frac{\Gamma, x : A \vdash C \quad \Gamma, y : B \vdash C}{\Gamma, z : A + B \vdash C}
\]

natural in $\Gamma$ and $C$. 

The connective $+$ is leftist, having a reversible rule

$$
\Gamma, x : A \vdash C \quad \Gamma, y : B \vdash C
\overline{\Gamma, z : A + B \vdash C}
$$
natural in $\Gamma$ and $C$.

The (nullary) connective $0$ is leftist, having a reversible rule

$$
\Gamma, z : 0 \vdash C
$$
natural in $\Gamma$ and $C$. 
The connective $\times$ has a reversible rule

\[
\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}
\]

natural in $\Gamma$, so it’s rightist.
The connective $\times$ has a reversible rule

$$
\frac{
\Gamma \vdash A \quad \Gamma \vdash B
}{
\Gamma \vdash A \times B
}$$
natural in $\Gamma$, so it’s rightist.

It also has a reversible rule

$$
\frac{
\Gamma, x : A, y : B \vdash C
}{
\Gamma, z : A \times B \vdash C
}$$
natural in $\Gamma$ and $C$, so it’s leftist.
Bipartisan connectives

The connective $\times$ has a reversible rule

$$\Gamma \vdash A \quad \Gamma \vdash B \quad \frac{}{\Gamma \vdash A \times B}$$

natural in $\Gamma$, so it’s rightist.

It also has a reversible rule

$$\Gamma, x : A, y : B \vdash C \quad \frac{}{\Gamma, z : A \times B \vdash C}$$

natural in $\Gamma$ and $C$, so it’s leftist.
Bipartisan connectives

The connective $\times$ has a reversible rule

$$
\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B}
$$

natural in $\Gamma$, so it’s rightist.

It also has a reversible rule

$$
\frac{\Gamma, x : A, y : B \vdash C}{\Gamma, z : A \times B \vdash C}
$$

natural in $\Gamma$ and $C$, so it’s leftist.

In summary, the connective $\times$ is bipartisan.
Likewise the (nullary) connective $1$. 
Most general leftist connective

The variant tuple type \[ \sum \{ 0 \ A, A'; \ 1 \ B, B', B'' \} \] denotes a sum of products

\[(\llbracket A \rrbracket \times \llbracket A' \rrbracket) + (\llbracket B \rrbracket \times \llbracket B' \rrbracket \times \llbracket B'' \rrbracket)\]

This gives a leftist connective.

\[
\Gamma, A, A' \vdash C \quad \Gamma, B, B', B'' \vdash C
\]

\[
\Gamma, \sum \{ 0 \ A, A'; \ 1 \ B, B', B'' \} \vdash C
\]
The variant tuple type $\sum \{ A, A'; B, B', B'' \}$ denotes a sum of products

$$([A] \times [A']) + ([B] \times [B'] \times [B''])$$

This gives a leftist connective.

$$\Gamma, A, A' \vdash C \quad \Gamma, B, B', B'' \vdash C$$
$$\Gamma, \sum \{ A, A'; B, B', B'' \} \vdash C$$

Here is its term syntax:

$$\text{in}_0(M, M')$$
$$\text{in}_1(M, M', M'')$$

match $M$ as \{ $\text{in}_0(x, x'). N$, $\text{in}_1(y, y', y''). N'$ \}
Most general rightist connective

The variant function type \( \prod \{ 0 \; A, A' \vdash B; \; 1 \; C, C', C'' \vdash D \} \) denotes a product of multi-ary function types

\[
(([[A] \times [A']]) \rightarrow [B]) \times (([[C] \times [C'] \times [C''']) \rightarrow [D])
\]

This gives a rightist connective.

\[
\Gamma, A, A' \vdash B \quad \Gamma, C, C', C'' \vdash D \\
\Gamma \vdash \prod \{ 0 \; A, A' \vdash B; \; 1 \; C, C', C'' \vdash D \}
\]
Most general rightist connective

The variant function type $\prod \{0 \ A, A' \vdash B; \ 1 \ C, C', C'' \vdash D\}$ denotes a product of multi-ary function types

$(([[A] \times [A']]) \rightarrow [B]) \times (([[C] \times [C'] \times [C'']) \rightarrow [D])$ 

This gives a rightist connective.

\[
\frac{\Gamma, A, A' \vdash B \ \Gamma, C, C', C'' \vdash D}{\Gamma \vdash \prod \{0 \ A, A' \vdash B; \ 1 \ C, C', C'' \vdash D\}}
\]

Here is its term syntax:

\[
\lambda\{0 (x, x').M, 1 (y, y', y'').M'\}
\]

\[
M^0(N, N')
\]

\[
M^1(N, N', N'')
\]
Jumbo $\lambda$-calculus

Type syntax

$$A ::= \sum \{ \overrightarrow{A_i} \}_{i<n} \mid \prod \{ \overrightarrow{A_i} \vdash B_i \}_{i<n} \quad (n \in \mathbb{N} \text{ or } n = \infty)$$

Term syntax, with type annotations omitted

$$M ::= x \mid \text{let } (\text{x be } \overrightarrow{M}). M$$
$$\quad \mid \text{in}_i(\overrightarrow{M})$$
$$\quad \mid \text{match } M \text{ as } \{ \text{in}_i(\overrightarrow{x}). M_i \}_{i<n}$$
$$\quad \mid \lambda \{ i(\overrightarrow{x}). M_i \}_{i<n}$$
$$\quad \mid M^i(\overrightarrow{M})$$
Jumbo $\lambda$-calculus

Type syntax

$$A ::= \sum \{ \overrightarrow{A_i} \}_{i<n} \mid \prod \{ \overrightarrow{A_i} \vdash B_i \}_{i<n} \quad (n \in \mathbb{N} \text{ or } n = \infty)$$

Term syntax, with type annotations omitted

$$M ::= x \mid \text{let } (x \text{ be } \overrightarrow{M}). M$$
$$\quad \mid \text{in}_i(\overrightarrow{M})$$
$$\quad \mid \text{match } M \text{ as } \{ \text{in}_i(\overrightarrow{x}). M_i \}_{i<n}$$
$$\quad \mid \lambda \{ i(\overrightarrow{x}). M_i \}_{i<n}$$
$$\quad \mid M^i(\overrightarrow{M})$$

Includes both pattern-match product $A \times B$ and projection product $A \Pi B$. 
Jumbo $\lambda$-calculus is the most expressive form of simply typed $\lambda$-calculus: it contains all leftist and rightist connectives as primitives.
Jumbo $\lambda$-calculus is the most expressive form of simply typed $\lambda$-calculus: it contains all leftist and rightist connectives as primitives. Modulo $\equiv_{\beta\eta}$ it is no more expressive than the non-jumbo version.
Jumbo $\lambda$-calculus is the most expressive form of simply typed $\lambda$-calculus: it contains all leftist and rightist connectives as primitives.

Modulo $\beta\eta$ it is no more expressive than the non-jumbo version.

But the $\beta$- and $\eta$-laws are not going to survive.
Evaluating terms

We want to evaluate every closed term $\vdash M : A$ to a terminal term. We want $\lambda x_A. M$ to be terminal, since $M$ is not closed. But there are many options.
Three decisions we must make

1. To evaluate `let (x be M, y be M'). N`, do we
   - evaluate `M` to `T` and `M'` to `T'`, then evaluate `N[T/x, T'/y]`?
   - just evaluate `N[M/x, M'/y]`?
Three decisions we must make

1. To evaluate `let (x be M, y be M'). N`, do we
   - evaluate `M` to `T` and `M'` to `T'`, then evaluate `N[T/x, T'/y]`?
   - just evaluate `N[M/x, M'/y]`?

2. To evaluate `M N`, we must evaluate `M` to `λx_A. P`. Do we
   - evaluate `N` to `T` (before or after evaluating `M`), then evaluate `P[T/x]`?
   - just evaluate `P[N/x]`?
Three decisions we must make

1. To evaluate \( \text{let} \ (x \ be \ M, \ y \ be \ M'). \ N, \) do we
   - evaluate \( M \) to \( T \) and \( M' \) to \( T' \), then evaluate \( N[T/x, T'/y] \)?
   - just evaluate \( N[M/x, M'/y] \)?

2. To evaluate \( MN \), we must evaluate \( M \) to \( \lambda x_A. P \). Do we
   - evaluate \( N \) to \( T \) (before or after evaluating \( M \)), then evaluate \( P[T/x] \)?
   - just evaluate \( P[N/x] \)?

3. Any terminal term of type \( A + B \) must be \( \text{inl} \ M \) or \( \text{inr} \ M \). Do we
   - deem \( \text{inl} \ T \) and \( \text{inr} \ T \) terminal only if \( T \) is terminal?
   - always deem \( \text{inl} \ M \) and \( \text{inr} \ M \) terminal?
Do we substitute **terminal** terms, or **unevaluated** terms?
One fundamental decision

Do we substitute **terminal** terms, or **unevaluated** terms?

Substituting terminal terms gives **call-by-value** or **eager** evaluation.

Substituting unevaluated terms gives **call-by-name**.
One fundamental decision

Do we substitute terminal terms, or unevaluated terms?

Substituting terminal terms gives call-by-value or eager evaluation.

Substituting unevaluated terms gives call-by-name.

Terminology: lazy and call-by-name

- “Lazy” evaluation usually means call-by-need, except in Abramsky’s “lazy λ-calculus”.
- In the untyped literature, “call-by-name” evaluation means reduction to head normal form.
Evaluation order for `let`

To evaluate `let (x be $M$, y be $M'$). $N$`, do we

- evaluate $M$ to $T$ and $M'$ to $T'$, then evaluate $N[T/x, T'/y]$? **Call-by-value**
- just evaluate $N[M/x, M'/y]$? **Call-by-name**
To evaluate $MN$, we must evaluate $M$ to $\lambda x_A. P$. Do we

- evaluate $N$ to $T$ (before or after evaluating $M$), then evaluate $P[T/x]$? **Call-by-value**
- just evaluate $P[N/x]$? **Call-by-name**
Any terminal term of type $A + B$ must be $\text{inl } M$ or $\text{inr } M$. Do we

- deem $\text{inl } T$ and $\text{inr } T$ terminal only if $T$ is terminal? **Call-by-value**
- always deem $\text{inl } M$ and $\text{inr } M$ terminal? **Call-by-name**

Consider evaluation of match $P$ as $\{\text{inl } x. N, \text{inr } y. N'\}$ to see this.
Definitional interpreter for call-by-value

CBV terminals \( T ::= \) true \mid false \mid inl \( T \) \mid inr \( T \) \mid \( \lambda x.M \)

To evaluate

- **true**: return true.
- \( M + N \): evaluate \( M \). If this returns \( m \), evaluate \( N \). If this returns \( n \), return \( m + n \).
- \( \lambda x.M \): return \( \lambda x.M \).
- \( \text{inl } M \): evaluate \( M \). If this returns \( T \), return **inl** \( T \).
- let (x be \( M \), y be \( M' \)). \( N \): evaluate \( M \). If this returns \( T \), evaluate \( M' \). If this returns \( T' \), evaluate \( N[T/x, T'/y] \).
- match \( M \) as \{true. \( N \), false. \( N' \}\}: evaluate \( M \). If this returns true, evaluate \( N \), but if it returns false, evaluate \( N' \).
- match \( M \) as \{inl \( x \). \( N \), inr \( x \). \( N' \}\}: evaluate \( M \). If this returns inl \( T \), evaluate \( N[T/x] \), but if it returns inr \( T \), evaluate \( N'[T/x] \).
- \( MN \): evaluate \( M \). If this returns \( \lambda x.P \), evaluate \( N \). If this returns \( T \), evaluate \( P[T/x] \).
In CBN the terminals are true, false, inl $M$, inr $M$, $\lambda x.M$

To evaluate

- **true**: return true.
- **$M + N$**: evaluate $M$. If this returns $m$, evaluate $N$. If this returns $n$, return $m + n$.
- **$\lambda x.M$**: return $\lambda x.M$.
- **inl $M$**: return inl $M$.
- **let (x be $M$, y be $M'$). $N$**: evaluate $N[M/x, M'/y]$.
- **match $M$ as {true. $N$, false. $N'$}**: evaluate $M$. If this returns true, evaluate $N$, but if it returns false, evaluate $N'$.
- **$MN$**: evaluate $M$. If this returns $\lambda x.P$, evaluate $P[N/x]$. 
We write $M \Downarrow T$ to mean that $M$ evaluates to $T$.

This is defined inductively, for example

$$
\begin{align*}
M \Downarrow \lambda x_A. P & \quad N \Downarrow T & \quad P[T/x] \Downarrow T' \\
\hline
M \; N \Downarrow T'
\end{align*}
$$
We write $M \Downarrow T$ to mean that $M$ evaluates to $T$. This is defined inductively, for example

$$
\frac{\begin{align*}
M \Downarrow \lambda x_A. P & \quad N \Downarrow T & \quad P[T/x] \Downarrow T'
\end{align*}}{M N \Downarrow T'}
$$

If $\vdash M : A$ then $M \Downarrow T$ for unique $T$. Moreover $\vdash T : A$ and $\llbracket M \rrbracket = \llbracket T \rrbracket$. 
We write $M \downarrow T$ to mean that $M$ evaluates to $T$. This is defined inductively, for example

$$
M \downarrow \lambda x_A. P \quad P[N/x] \downarrow T
$$

$$
\frac{M \downarrow \lambda x_A. P \quad P[N/x] \downarrow T}{M \, N \downarrow T}
$$
We write $M \Downarrow T$ to mean that $M$ evaluates to $T$. This is defined inductively, for example

$$
M \Downarrow \lambda x_A. P \quad P[N/x] \Downarrow T
$$

$$
\frac{}{MN \Downarrow T}
$$

If $\vdash M : A$ then $M \Downarrow T$ for unique $T$.

Moreover $\vdash T : A$ and $\llbracket M \rrbracket = \llbracket T \rrbracket$. 
Add effects to (jumbo) $\lambda$-calculus, with CBV or CBN evaluation.
See what equations and isomorphisms survive.
Seek a denotational semantics for each language.
The experiment

- Add effects to (jumbo) $\lambda$-calculus, with CBV or CBN evaluation.
- See what equations and isomorphisms survive.
- Seek a denotational semantics for each language.

Analyzing CBV with a microscope

- Look closely at the CBV models: there’s a pattern.
- CBV contains particles of meaning, constituting fine-grain call-by-value.
The experiment

- Add effects to (jumbo) $\lambda$-calculus, with CBV or CBN evaluation.
- See what equations and isomorphisms survive.
- Seek a denotational semantics for each language.

Analyzing CBV with a microscope

- Look closely at the CBV models: there’s a pattern.
- CBV contains particles of meaning, constituting fine-grain call-by-value.

Increasing the magnification

- Look very closely at the CBN and fine-grain CBV models: there’s a pattern.
- Both contain tiny particles of meaning, constituting call-by-push-value.
Both fine-grain call-by-value and call-by-push-value are obtained **empirically**, by observing particles of meaning within a range of denotational models.
Where this story comes from

- Plotkin: semantics of recursion for call-by-name (PCF) and call-by-value (FPC)
- Moggi: list of monads for denotational semantics
- Moggi: monadic metalanguage
- Power and Robinson: Freyd categories
- Plotkin and Felleisen: call-by-value continuation semantics
- Reynolds’ Idealized Algol, a call-by-name language with state
- O’Hearn: semantics of type identifiers in such a language
- Streicher and Reus: call-by-name continuation semantics
- Filinski: Effect-PCF
Adding computational effects

Errors

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

$$\frac{}{\Gamma \vdash \text{error}^B e : B \quad e \in E}$$

To evaluate $\text{error}^B e$: halt with error message $e$.

Printing

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

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

To evaluate $\text{print } c. M$: print $c$ and then evaluate $M$. 
Exercises

1. Evaluate

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

   in CBV and CBN.

2. Evaluate

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

   in CBV and CBN.

3. Evaluate

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

   in CBV and CBN.
Big-step semantics for errors

For call-by-value, we inductively define two big-step relations:

- $M \Downarrow T$ means $M$ evaluates to $T$.
- $M \not\Downarrow e$ means $M$ raises error $e$.

Here are the rules for application:

\[
\begin{align*}
M \not\Downarrow e & \quad & M \Downarrow \lambda x. P & \quad N \not\Downarrow e \\
\hline
M \downarrow N & \not\Downarrow e & M \downarrow N & \Downarrow e
\end{align*}
\]

\[
\begin{align*}
M \Downarrow \lambda x. P & \quad N \Downarrow T & P[T/x] \not\Downarrow e \\
\hline
M \downarrow N & \not\Downarrow e & M \downarrow N & \Downarrow e
\end{align*}
\]

Likewise for call-by-name.
A program is a closed term of type \texttt{nat} or \texttt{bool}.

Two terms $\Gamma \vdash M, M' : B$ are observationally equivalent when $C[M]$ and $C[M']$ have the same behaviour for every program with a hole $C[\cdot]$.

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

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

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,

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

because we can apply the context

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

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 $A \to B$ and $A \Pi B$

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

Any term $\Gamma \vdash M : A \Pi B$ can be expanded as $\lambda \{^1. M^1, \ r. M^r \}$.

Although these fail in CBV, they hold in CBN. Consequences:

\[
\begin{align*}
\text{error } e & \equiv_{\text{CBN}} \lambda x. \text{error } e \\
\text{error } e & \equiv_{\text{CBN}} \lambda \{^1. \text{error } e, \ r. \text{error } e \} \\
\text{print } c. \lambda x. M & \equiv_{\text{CBN}} \lambda x. \text{print } c. M \\
\text{print } c. \lambda \{^1. M, \ r. N \} & \equiv_{\text{CBN}} \lambda \{^1. \text{print } c. M, \ r. \text{print } c. N \}
\end{align*}
\]

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

In CBN, a function gets evaluated only by being applied.
The pure λ-calculus satisfies all the β- and η-laws.

With computational effects,

- CBV satisfies η for leftist connectives (tuple types), but not rightist ones (function types)
- CBN satisfies η for rightist connectives (function types), but not leftist ones (tuple types).
The pure $\lambda$-calculus satisfies all the $\beta$- and $\eta$-laws.

With computational effects,
- CBV satisfies $\eta$ for leftist connectives (tuple types), but not rightist ones (function types)
- CBN satisfies $\eta$ for rightist connectives (function types), but not leftist ones (tuple types).

Similarly for isomorphisms:
- $(A + B) + C \cong A + (B + C)$ survives in CBV but not CBN.
- $A \times B \cong A \Pi B$ survives in neither CBV nor CBN.
- $A \to (B \to C) \cong (A \Pi B) \to C$ survives in CBN but not CBV.
Naive CBV semantics

Our first attempt.

Each type $A$ denotes a set, a semantic domain for terms.

\[
\begin{align*}
[\text{bool}]^\ast & = \mathbb{B} + E \\
[\text{bool} + \text{bool}]^\ast & = (\mathbb{B} + \mathbb{B}) + E \\
[\text{bool} \times \text{bool}]^\ast & = (\mathbb{B} \times \mathbb{B}) + E
\end{align*}
\]
Naive CBV semantics

Our first attempt.

Each type $A$ denotes a set, a semantic domain for terms.

\[
\begin{align*}
\lbrack \text{bool} \rbrack_\ast &= \mathbb{B} + E \\
\lbrack \text{bool + bool} \rbrack_\ast &= (\mathbb{B} + \mathbb{B}) + E \\
\lbrack \text{bool } \times \text{bool} \rbrack_\ast &= (\mathbb{B} \times \mathbb{B}) + E
\end{align*}
\]

Not easy to make this compositional, so we abandon it.
Each type denotes a set, a semantic domain for terminals.

\[
\begin{align*}
[\text{bool}] &= \mathbb{B} \\
[A + B] &= [A] + [B] \\
[A \to B] &= [A] \to ([B] + E) \\
() \to B &= [B] + E \\
[\Gamma] &= \prod_{(x:A) \in \Gamma} [A]
\end{align*}
\]
Each type denotes a set, a semantic domain for terminals.

\[
\begin{align*}
\llbracket \text{bool} \rrbracket &= \mathbb{B} \\
\llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\
\llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow (\llbracket B \rrbracket + E) \\
\llbracket () \rightarrow B \rrbracket &= \llbracket B \rrbracket + E \\
\llbracket \Gamma \rrbracket &= \prod_{(x:A) \in \Gamma} \llbracket A \rrbracket
\end{align*}
\]

Each term $\Gamma \vdash M : B$ denotes a function $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow (\llbracket B \rrbracket + E)$. 
Semantics of term constructors

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x \in A. M : A \rightarrow B}
\]

\[
\llbracket \lambda x_A. M \rrbracket : \rho \mapsto \text{inl } \lambda a \in \llbracket A \rrbracket. \llbracket M \rrbracket(\rho, x \mapsto a)
\]

\[
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}
\]

\[
\llbracket MN \rrbracket : \rho \mapsto \text{match } \llbracket M \rrbracket\rho \text{ as } \begin{cases} \text{inl } f. & \text{match } \llbracket N \rrbracket\rho \text{ as } \begin{cases} \text{inl } x. & f(x) \\ \text{inr } e. & \text{inr } e \end{cases} \\ \text{inr } e. & \text{inr } e \end{cases}
\]
More term constructors

\[
\Gamma \vdash M : A \\
\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl}^{A,B} M : A + B}
\]

\[
\left[ \text{inl}^{A,B} M \right] : \rho \mapsto \text{match } \left[ M \right] \rho \text{ as } \begin{cases} 
\text{inl } a. & \text{inl } \text{inl } a \\
\text{inr } e. & \text{inr } e
\end{cases}
\]
More term constructors

\[ \Gamma \vdash M : A \]
\[ \Gamma \vdash \text{inl}^{A,B} M : A + B \]

\[ [\text{inl}^{A,B} M] : \rho \mapsto \text{match} [M]_\rho \text{ as } \begin{cases} \text{inl } a. & \text{inl inl } a \\ \text{inr } e. & \text{inr } e \end{cases} \]

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

Example that rules out a general substitution lemma

Define $\vdash M : \text{bool}$ and $x : \text{bool} \vdash N, N' : \text{bool}$.

\[
\begin{align*}
M & \overset{\text{def}}{=} \text{error CRASH} \\
N & \overset{\text{def}}{=} \text{true} \\
N' & \overset{\text{def}}{=} \text{match } x \text{ as } \{ \text{true.true, false.true} \} \\
\llbracket N \rrbracket & = \llbracket N' \rrbracket \text{ because } N =_{\eta \text{bool}} N' \\
\llbracket N[M/x] \rrbracket & \neq \llbracket N'[M/x] \rrbracket
\end{align*}
\]
Can we obtain \([N[M/x]]\) from \([M]\) and \([N]\)? Not in CBV.

Example that rules out a general substitution lemma

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

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

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

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

But we can give a lemma for the substitution of values.
The following terms are called values.

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

The closed values are just the terminals: we don’t allow “complex values” such as

\[
\text{match true as \{true.false, false.true\}}
\]
Each value $\Gamma \vdash V : A$ denotes a function $[V]^{\text{val}} : [\Gamma] \rightarrow [A]$.

\[
\begin{align*}
[x]^{\text{val}} & : \rho \mapsto \rho_x \\
[\text{true}]^{\text{val}} & : \rho \mapsto \text{true} \\
[\text{inl } V]^{\text{val}} & : \rho \mapsto \text{inl } [V]^{\text{val}} \rho \\
[\lambda x_A. M]^{\text{val}} & : \rho \mapsto \lambda a \in [A]. [M](\rho, x \mapsto [a])
\end{align*}
\]

We can recover $[V]$ from $[V]^{\text{val}}$.

\[
[V] : \rho \mapsto \text{inl } [V]^{\text{val}} \rho
\]
Substitution Lemma For Values

Given values $\Gamma \vdash V : A$ and $\Gamma \vdash W : B$ and a term $\Gamma, x : A, y : B \vdash M : C$ we can obtain $[M[V/x, W/y]]$ from $[V]^{\text{val}}$ and $[W]^{\text{val}}$ and $[M]$.

$$[M[V/x, W/y]] : \rho \mapsto [M](\rho, x \mapsto [V]^{\text{val}} \rho, y \mapsto [W]^{\text{val}} \rho)$$

Likewise for substitution of values into values.
Soundness of CBV Denotational Semantics

- If $M \downarrow V$ then $\llbracket M \rrbracket \varepsilon = \text{inl} (\llbracket V \rrbracket^{\text{val}} \varepsilon)$.
- If $M \not\downarrow e$ then $\llbracket M \rrbracket \varepsilon = \text{inr} e$.

Proof by induction, using the substitution lemma.
Fine-grain call-by-value has two judgements:

- A value $\Gamma \vdash^v V : A$ denotes a function $[[V]] : [[\Gamma]] \rightarrow [[A]]$.
- A computation $\Gamma \vdash^c M : A$ denotes a function $[[M]] : [[\Gamma]] \rightarrow [[A]] + E$.

Key typing rules

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

Corresponds to Power and Robinson’s notion of a Freyd category.
Semantics of returning and sequencing

\[
\Gamma \vdash^v V : A \\
\frac{\Gamma \vdash^c \text{return } V : A }{\Gamma \vdash^c \text{return } V : A}
\]

\[
[\text{return } V] : \rho \mapsto \text{inl } [V] \rho
\]

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

\[
[\text{M to x. N}] : \rho \mapsto \text{match } [M] \rho \text{ as } \begin{cases} 
\text{inl } a. & [N](\rho, x \mapsto a) \\
\text{inr } e. & \text{inr } e
\end{cases}
\]
Syntax

For connectives bool, +, → the syntax is as follows.

\[
V ::= x \mid \text{true} \mid \text{false} \\
\quad \quad \quad \quad \mid \text{inl } V \mid \text{inr } V \mid \lambda x. M \\
M ::= M \text{ to } x. M \mid \text{return } V \\
\quad \quad \quad \quad \mid \text{let } (x \text{ be } V). M \mid V V \\
\quad \quad \quad \quad \mid \text{match } V \text{ as } \{ \text{true. } M, \text{ false. } M \} \\
\quad \quad \quad \quad \mid \text{match } V \text{ as } \{ \text{inl } x. M, \text{ inr } x. M \} \\
\quad \quad \quad \quad \mid \text{error } e
\]
Syntax

For connectives `bool`, `+`, `→` the syntax is as follows.

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

\[
M ::= M \text{ to } x. M | \text{return } V \\
    | \text{let } (x \text{ be } V). M | V V \\
    | \text{match } V \text{ as } \{ \text{true. } M, \text{false. } M \} \\
    | \text{match } V \text{ as } \{ \text{inl } x. M, \text{inr } x. M \} \\
    | \text{error} e
\]

We don’t allow “complex values” such as

\[
\text{match true as } \{ \text{true. } \text{false}, \text{false. } \text{true} \}
\]

These would complicate the operational semantics.
We evaluate a closed computation $\vdash^c M : A$ to a closed value $\vdash^v V : A$. To evaluate

- **return $V$**: return $V$.
- **$M$ to $x$. $N$**, evaluate $M$. If this returns $V$, evaluate $N[V/x]$.
- **let (x be $V$, y be $W$). $M$**, evaluate $M[V/x,W/y]$.
- **$(\lambda x. M) V$**, evaluate $M[V/x]$.
Equational theory

$\beta$-laws

\[
\text{match (inl } V \text{) as } \{\text{true. } M, \text{false. } M'\} = M[V/x] \\
(\lambda x. M) V = M[V/x] \\
\text{let (x be } V, \text{ y be } W). \ M = M[V/x, W/y]
\]

$\eta$-laws

\[
M[V/z] = \text{match } V \text{ as } \{\text{inl } x. M[\text{inl } x/z], \text{ inr } y. M[\text{inr } x/z]\} \\
V = \lambda x. Vx
\]

Sequencing laws

\[
(\text{return } V \text{) to x. } M = M[V/x] \\
M = M \text{ to x. return } x \\
(M \text{ to x. } N \text{) to y. } P = M \text{ to x. } (N \text{ to y. } P)
\]
CBV to fine-grain call-by-value

Term $\Gamma \vdash M : A$ to computation $\Gamma \vdash^c \hat{M} : A$.

\[
\begin{align*}
x & \mapsto \text{return } x \\
\lambda x. M & \mapsto \text{return } \lambda x. \hat{M} \\
inl M & \mapsto \hat{M} \text{ to } x. \text{return } \text{inl } x \\
MN & \mapsto \hat{M} \text{ to } x. \hat{N} \text{ to } y. x y \\
\text{let } (x \text{ be } M, y \text{ be } M'). N & \mapsto \hat{M} \text{ to } x. \hat{M}' \text{ to } y. \hat{N}
\end{align*}
\]

Value $\Gamma \vdash V : A$ to value $\Gamma \vdash^v \check{V} : A$.

\[
\begin{align*}
x & \mapsto x \\
\lambda x. M & \mapsto \lambda x. \hat{M} \\
inl V & \mapsto \text{inl } \check{V}
\end{align*}
\]
Nullary functions

Call-by-value programmers use nullary functions to delay evaluation, and call them thunks.

\[
\begin{align*}
TA & \equiv () \to A \\
\text{thunk } M & \equiv \lambda(). M \\
\text{force } V & \equiv V() \\
\end{align*}
\]

\[ [TA] = [A] + E \]

\[ [\text{thunk } M] = [M] \]

\[ [\text{force } V] = [V] \]
Nullary functions

Call-by-value programmers use nullary functions to delay evaluation, and call them *thunks*.

\[
TA \overset{\text{def}}{=} () \rightarrow A \\
\text{thunk } M \overset{\text{def}}{=} \lambda(). M \\
\text{force } V \overset{\text{def}}{=} V() \\
\]

\[
[TA] = [A] + E \\
[\text{thunk } M] = [M] \\
[\text{force } V] = [V] \\
\]

The type \( TA \) has a reversible rule \( \frac{\Gamma \vdash^c A}{\Gamma \vdash^v TA} \).
Nullary functions

Call-by-value programmers use nullary functions to delay evaluation, and call them *thunks*.

\[
TA \overset{def}{=} () \rightarrow A \quad [TA] = [A] + E \\
thunk M \overset{def}{=} \lambda().M \quad [\text{thunk } M] = [M] \\
force V \overset{def}{=} V() \quad [\text{force } V] = [V]
\]

The type \(TA\) has a reversible rule

\[
\Gamma \vdash^c A \\
\Gamma \vdash^v TA
\]

Fine-grain CBV (unlike the *monadic metalanguage*) distinguishes computations from thunks.
Naive CBN semantics of errors

Each type denotes a set, a **semantic domain for terms**. For example:

\[
[\text{bool} \to (\text{bool} \to \text{bool})]_\ast = (\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E))
\]

\[
[\text{bool} + \text{bool}]_\ast = ((\mathbb{B} + E) + (\mathbb{B} + E)) + E
\]

\[
[\text{bool} \Pi \text{bool}]_\ast = (\mathbb{B} + E) \times (\mathbb{B} + E)
\]

Thus we define

\[
[\text{bool}]_\ast = \mathbb{B} + E
\]

\[
[A + B]_\ast = ([A]_\ast + [B]_\ast) + E
\]

\[
[A \to B]_\ast = [A]_\ast \to [B]_\ast
\]

\[
[A \Pi B]_\ast = [A]_\ast \times [B]_\ast
\]

\[
[\Gamma] = \prod_{(x:A) \in \Gamma} [A]_\ast
\]

Each term \( \Gamma \vdash M : B \) should denote a function \([M] : [\Gamma] \to [B]_\ast\).
Naive semantics: what goes wrong

Γ ⊢ error CRASH : B

denotes \( \rho \mapsto ? \)

Example:

suppose \( B = \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \) then \( B \) denotes \( (B + E) \rightarrow ((B + E) \rightarrow (B + E)) \) and \( \text{error CRASH} \simeq \text{CBN} \lambda x. \lambda y. \text{error CRASH} \)

so the answer should be \( \lambda x. \lambda y. \text{inr CRASH} \).

Intuition: go down through the function types until we hit a tuple type.

A similar problem arises with match.
Naive semantics: what goes wrong

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

\( \text{denotes } \rho \mapsto ? \)

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 \( \text{error CRASH} \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error CRASH} \)
- so the answer should be \( \lambda x. \lambda y. \text{inr CRASH} \).

Intuition: go down through the function types until we hit a tuple type.
Naive semantics: what goes wrong

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

\[ \text{denotes } \rho \mapsto ? \]

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 \( \text{error CRASH} \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error CRASH} \)
- so the answer should be \( \lambda x. \lambda y. \text{inr CRASH} \).

Intuition: go down through the function types until we hit a tuple type. A similar problem arises with \text{match}. 
Definition

An $E$-pointed set is a set $X$ with two distinguished elements $c, b \in X$.

A type should denote an $E$-pointed set, a semantic domain for terms.
Solution: \( E \)-pointed sets

Definition

An \( E \)-pointed set is a set \( X \) with two distinguished elements \( c, b \in X \).

A type should denote an \( E \)-pointed set, a semantic domain for terms. Examples:

\[
\begin{align*}
\llbracket \text{bool} \to (\text{bool} \to \text{bool}) \rrbracket &= ((\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E))), \\
&\quad \lambda x.\lambda y.\text{inr CRASH}, \\
&\quad \lambda x.\lambda y.\text{inr BANG})
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{bool} + \text{bool} \rrbracket &= (((\mathbb{B} + E) + (\mathbb{B} + E)) + E, \\
&\quad \text{inr CRASH}, \\
&\quad \text{inr BANG})
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{bool} \Pi \text{bool} \rrbracket &= ((\mathbb{B} + E) \times (\mathbb{B} + E), \\
&\quad (\text{inr CRASH, inr CRASH}), \\
&\quad (\text{inr BANG, inr BANG}))
\end{align*}
\]
CBN semantics of errors

\[
\llbracket \text{bool} \rrbracket = (\mathbb{B} + E, \text{inr CRASH}, \text{inr BANG})
\]

If \( \llbracket A \rrbracket = (X, c, b) \) and \( \llbracket B \rrbracket = (Y, c', b') \) then

\[
\llbracket A + B \rrbracket = ((X + Y) + E, \text{inr CRASH}, \text{inr BANG})
\]

and \( \llbracket A \to B \rrbracket = (X \to Y, \lambda x. c', \lambda x. b') \)

and \( \llbracket A \Pi B \rrbracket = (X \times Y, (c, c'), (b, b')) \)
CBN semantics of errors

\[
\text{[bool]} = (\mathbb{B} + E, \text{inr CRASH}, \text{inr BANG})
\]

If \([A] = (X, c, b)\) and \([B] = (Y, c', b')\)

then \([A + B] = ((X + Y) + E, \text{inr CRASH}, \text{inr BANG})\)

and \([A \rightarrow B] = (X \rightarrow Y, \lambda x. c', \lambda x. b')\)

and \([A \Pi B] = (X \times Y, (c, c'), (b, b'))\)

\[
\text{[\Gamma]} = \prod_{(x: A) \in \Gamma} \text{X}
\]

A term \(\Gamma \vdash M : B\) denotes a function \(\text{[M]} : \text{[\Gamma]} \rightarrow \text{[B]}\).
Semantics of term constructors

\[ \Gamma \vdash \text{true} : \text{bool} \]

\[ [\text{true}] : \rho \mapsto \text{inl true} \]

\[ \begin{array}{c}
\Gamma \vdash M : \text{bool} \quad \Gamma \vdash N : B \quad \Gamma \vdash N' : B \\
\Gamma \vdash \text{match } M \text{ as } \{ \text{true. } N, \text{ false. } N' \} : B
\end{array} \]

\[ [\text{match } M \text{ as } \{ \text{true. } N, \text{ false. } N' \}] : \rho \mapsto \]

match \([M]\rho\) as
\[ \left\{ \begin{array}{ll}
\text{inl true.} & [N]\rho \\
\text{inl false.} & [N']\rho \\
\text{inr CRASH.} & c \\
\text{inr BANG.} & b
\end{array} \right. \]

where \([B] = (Y, c, b)\)
More term constructors

\[
\begin{align*}
\llbracket \lambda x. M \rrbracket & : \rho \mapsto \lambda a. \llbracket M \rrbracket (\rho, x \mapsto a) \\
\llbracket M \ N \rrbracket & : \rho \mapsto \llbracket M \rrbracket \llbracket N \rrbracket \\
\llbracket x \rrbracket & : \rho \mapsto \rho_x \\
\text{error CRASH} & : \rho \mapsto c
\end{align*}
\]

Soundness/adequacy

- If \( M \Downarrow T \) then \( \llbracket M \rrbracket \varepsilon = \llbracket T \rrbracket \varepsilon \).
- If \( M \nLeftarrow \text{CRASH} \) then \( \llbracket M \rrbracket \varepsilon = c \).
- If \( M \nLeftarrow \text{BANG} \) then \( \llbracket M \rrbracket \varepsilon = b \).

Proved by induction, using the substitution lemma.
Notation for $E$-pointed sets

- Free $E$-pointed set on a set $X$.
  \[ F^E X \overset{\text{def}}{=} (X + E, \text{inr CRASH}, \text{inr BANG}) \]

- Product of two $E$-pointed sets.
  \[ (X, c, b) \times (Y, c', b') \overset{\text{def}}{=} (X \times Y, (c, c'), (b, b')) \]

- Unit $E$-pointed set.
  \[ 1\Pi \overset{\text{def}}{=} (1, ( ), ( )) \]

- Product of a family of $E$-pointed sets.
  \[ \prod_{i \in I} (X_i, c_i, b_i) \overset{\text{def}}{=} \left( \prod_{i \in I} X_i, \lambda_i. c_i, \lambda_i. b_i \right) \]

- Exponential $E$-pointed set.
  \[ X \to (Y, c, b) \overset{\text{def}}{=} \prod_{x \in X} (Y, c, b) \]
  \[ = (X \to Y, \lambda x. c, \lambda x. b) \]

- Carrier of an $E$-pointed set.
  \[ U^E (X, c, b) \overset{\text{def}}{=} X \]
Summary of call-by-name semantics

A type denotes an $E$-pointed set.

\[
\begin{align*}
[\text{bool}] &= F^E(1+1) \\
[A + B] &= F^E(U^E[A] + U^E[B]) \\
[A \to B] &= U^E[A] \to [B] \\
[A \Pi B] &= [A] \Pi [B]
\end{align*}
\]

A typing context denotes a set.

\[
[\Gamma] = \prod_{(x:A)\in\Gamma} U^E[A]
\]

A term $\Gamma \vdash^c M : B$ denotes a function $[\Gamma] \longrightarrow [B]$. 
Summary of call-by-value semantics

A type denotes a set.

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

A typing context denotes a set.

\[
[\Gamma] = \prod_{(x:A) \in \Gamma} [A]
\]

A computation $\Gamma |-^c M : B$ denotes a function $[\Gamma] \rightarrow F^E [B]$. 
Two kinds of type:

- A **value type** denotes a set.
- A **computation type** denotes an $E$-pointed set.
Call-By-Push-Value Types

Two kinds of type:

- A value type denotes a set.
- A computation type denotes an $E$-pointed set.

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

Strangely function types are computation types, and $\lambda x.M$ is a computation.
Call-By-Push-Value Types

Two kinds of type:

- A **value type** denotes a set.
- A **computation type** denotes an $E$-pointed set.

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

computation type \[ B ::= \quad FA \mid A \to B \mid 1_{\Pi} \mid B \Pi B \mid \prod_{i \in \mathbb{N}} B_i \]

**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

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

Two judgements:

- A value $\Gamma \vdash^\nu V : A$ denotes a function $[V] : [\Gamma] \rightarrow [A]$.
The type $FA$

A computation in $FA$ aims to return a value in $A$.

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

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

Sequencing in the style of Filinski’s “Effect-PCF”.
The type $FA$

A computation in $FA$ aims to return a value in $A$.

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

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

Sequencing in the style of Filinski’s “Effect-PCF”.

$$\left[ \text{return } V \right] : \rho \mapsto \text{inl } \left[V\right] \rho$$

$$\left[ M \text{ to } x. \ N \right] : \rho \mapsto \text{inl } a. \left[N\right](\rho, x \mapsto a)$$

$$\text{match } \left[M\right] \rho \text{ as } \begin{cases} \text{inr CRASH. } \ b & \text{inr BANG. } \ c \end{cases}$$

where $\left[B\right] = (Y, c, b)$
The type $UB$

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

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

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

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

\[
\begin{align*}
[\text{thunk} \ M] & = [M] \\
[\text{force} \ V] & = [V]
\end{align*}
\]
An identifier is a value.

\[ \Gamma \vdash^v x : A \in \Gamma \]

\[ \Gamma \vdash^v V : A \quad \Gamma \vdash^v W : B \quad \Gamma, x : A, y : B \vdash^c M : C \]

\[ \Gamma \vdash^c \text{let} (x \text{ be } V, y \text{ be } W).\ M : C \]
Tuples

\[\Gamma \vdash^v V : A\]

\[\Gamma \vdash^v \text{inl } V : A + A'\]

\[\Gamma \vdash^v V : A + A'\]
\[\Gamma, x : A \vdash^c M : B\]
\[\Gamma, y : A \vdash^c M' : B\]
\[\Gamma \vdash^c \text{match } V \text{ as } \{\text{inl } x. M, \text{inr } y. M'\} : B\]

\[\Gamma \vdash^v V : A\]
\[\Gamma \vdash^v V' : A'\]
\[\Gamma \vdash^v \langle V, V' \rangle : A \times A'\]

\[\Gamma \vdash^v V : A \times A'\]
\[\Gamma, x : A, y : A' \vdash^c M : B\]
\[\Gamma \vdash^c \text{match } V \text{ as } \langle x, y \rangle. M : B\]

The rules for 1 are similar.
Functions

\[ \Gamma, x : A \vdash^c M : B \quad \Gamma \vdash^c M : A \rightarrow B \quad \Gamma \vdash^c V : A \]

\[ \Gamma \vdash^c \lambda x. M : A \rightarrow B \quad \Gamma \vdash^c MV : B \]

\[ \Gamma \vdash^c M : B \quad \Gamma \vdash^c M' : B' \]

\[ \Gamma \vdash^c \lambda^{\text{l}} M, \text{ r} M' : B \uplus B' \]

\[ \Gamma \vdash^c M : B \uplus B' \]

\[ \Gamma \vdash^c M^1 : B \]

\[ \Gamma \vdash^c M^r : B' \]
Functions

\[\Gamma, x : A \vdash^c M : B\]
\[\Gamma \vdash^c \lambda x. M : A \to B\]
\[\Gamma \vdash^c M : A \to B\]
\[\Gamma \vdash^y V : A\]
\[\Gamma \vdash^c M^V : B\]

\[\Gamma \vdash^c M : B\]
\[\Gamma \vdash^c M' : B'\]
\[\Gamma \vdash^c \lambda\{^1. M, ^r. M'\} : B \pi B'\]

\[\Gamma \vdash^c M : B \pi B'\]
\[\Gamma \vdash^c M^1 : B\]
\[\Gamma \vdash^c M^r : B'\]

It is often convenient to write applications operand-first, as \(V^\cdot M\) and \(^1^\cdot M\) and \(^r^\cdot M\).
The terminals are computations: return $V$  $\lambda x.M$  $\lambda\{^l . M, ^r . M'\}$
The terminals are computations:  \( \text{return } V \quad \lambda x.M \quad \lambda\{^1.M, ^r.M'\} \)

To evaluate

- **return** \( V \): return \( \text{return } V \).
- **\( M \) to \( x.\ N \)**: evaluate \( M \). If this returns \( \text{return } V \), then evaluate \( N[V/x] \).
- **\( \lambda x.N \)**: return \( \lambda x.N \).
- **\( MV \)**: evaluate \( M \). If this returns \( \lambda x.N \), evaluate \( N[V/x] \).
- **\( \lambda\{^1.M, ^r.M'\} \)**: return \( \lambda\{^1.M, ^r.M'\} \).
- **\( M^1 \)**: evaluate \( M \). If this returns \( \lambda\{^1.N, ^r.N'\} \), evaluate \( N \).
- **let \( (x \text{ be } V, \ y \text{ be } W).\ M \)**: evaluate \( M[V/x, W/y] \).
- **force thunk** \( M \): evaluate \( M \).
- **match inl** \( V \) as \{inl x.\ M, inr y.\ M'\}**: evaluate \( M[V/x] \).
- **match** \( \langle V, V' \rangle \) as \( \langle x, y \rangle.\ M \)**: evaluate \( M[V/x, V'/y] \).
- **error** \( e \), print error message \( e \) and stop.
Equational theory

$\beta$-laws

\[
\text{force thunk } M = M
\]
\[
\text{match (inl } V) \text{ as } \{\text{true. } M, \text{false. } M'\} = M[V/x]
\]
\[
(\lambda x. M) V = M[V/x]
\]
\[
\text{let (x be } V, \text{ y be } W). M = M[V/x, W/y]
\]

$\eta$-laws

\[
V = \text{thunk force } V
\]
\[
M[V/z] = \text{match } V \text{ as } \{\text{inl } x. M[\text{inl } x/z], \text{inr } y. M[\text{inr } x/z]\}
\]
\[
M = \lambda x. M x
\]

Sequencing laws

\[
(\text{return } V) \text{ to } x. M = M[V/x]
\]
\[
M = M \text{ to } x. \text{return } x
\]
\[
(\text{M to } x. N) \text{ to } y. P = M \text{ to } x. (\text{N to } y. P)
\]
Decomposing CBV into CBPV

A CBV type translates into a value type.

\[
A \to B \quad \iff \quad U(A \to FB)
\]

\[
TB \quad \iff \quad UFB
\]
Decomposing CBV into CBPV

A CBV type translates into a value type.

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

\[
TB \iff UFB
\]

A fine-grain CBV computation \(x : A, y : B \vdash^c M : C\) translates as \(x : A, y : B \vdash^c M : FC\).
Decomposing CBV into CBPV

A CBV type translates into a value type.

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

A fine-grain CBV computation \( x : A, y : B \vdash^c M : C \)
translates as \( x : A, y : B \vdash^c M : FC \).

\[ \lambda x. M \mapsto \text{thunk } \lambda x. M \]
\[ VW \mapsto (\text{force } V) W \]
Decomposing CBV into CBPV

A CBV type translates into a value type.

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

A fine-grain CBV computation \( x : A, y : B \vdash^c M : C \) translates as \( x : A, y : B \vdash^c M : FC \).

\[ \lambda x. M \mapsto \text{thunk } \lambda x. M \]
\[ VW \mapsto (\text{force } V) W \]

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

\[ x \mapsto \text{return } x \]
\[ \lambda x. M \mapsto \text{return thunk } \lambda x. M \]
\[ MN \mapsto M \text{ to } f. N \text{ to } y. ((\text{force } f) y) \]
Decomposing CBN into CBPV

A CBN type translates into a computation type.

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

A CBN type translates into a computation type.

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

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

\[
\begin{align*}
x & \mapsto \text{force } x \\
\text{let (x be } M, \text{ y be } M') \cdot N & \mapsto \text{let (x be thunk } M, \text{ y be thunk } M') \cdot N \\
\lambda x. \ M & \mapsto \lambda x. \ M \\
M \ N & \mapsto M \ (\text{thunk } N) \\
\text{inl } M & \mapsto \text{return inl } \text{thunk } M
\end{align*}
\]
We’ve seen

- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.
Summary

We’ve seen

- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.

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

We’ve seen

- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.

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 semantics
- denotational semantics for errors.

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 semantics
- denotational semantics for errors.

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

Moggi’s $TA$ is $UFA$.

But
- our error semantics makes thunk and force invisible
We’ve seen

- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.

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

Moggi’s $TA$ is $UFA$.

But

- our error semantics makes `thunk` and `force` invisible
- we still don’t understand why a function is a computation.
CK-machine

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”.

Transitions for sequencing

To evaluate $M \to x. N$: evaluate $M$. If this returns return $V$, then evaluate $N[V/x]$.

$$\begin{array}{c}
M \to x. N \quad K \rightsquigarrow \\
M \quad \text{to x. } N :: K
\end{array}$$

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

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

\[
\begin{array}{ccc}
V'M & K & \rightsquigarrow \\
M & V :: K \\
\lambda x.N & V :: K & \rightsquigarrow \\
N[V/x] & K
\end{array}
\]
Those function rules again

\[
V' M \quad K \quad \rightsquigarrow \\
M \quad V :: K
\]

\[
\lambda x. N \quad V :: K \quad \rightsquigarrow \\
N[V/x] \quad K
\]
Those function rules again

We can read $V'$ as an instruction “push $V$”.

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

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

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

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

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

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

Revisiting some equations:

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

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

\[
\text{print } c. \lambda x. M = \lambda x. \text{print } c. M
\]
A value is, a computation does.

- A value of type $UB$ is a thunk of a computation of type $B$.
- A value of type $A + A'$ is a tagged value $\text{inl } V$ or $\text{inr } V$.
- A value of type $A \times A'$ is a pair $\langle V, V' \rangle$.

- A computation of type $FA$ aims to return a value of type $A$.
- A computation of type $A \rightarrow B$ aims to pop a value of type $A$ and then behave in $B$.
- A computation of type $B \bowtie B'$ aims to pop the tag $l$ and then behave in $B$ or pop the tag $r$ and then behave in $B'$. 
What’s in a stack?

A stack consists of

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

print "hello0".
let (x be 3,
    y be thunk (  
        print "hello1".
        λz.
        print "we just popped " + z.
        return x + z  
    )).  
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>
<th>$\leadsto$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$</td>
<td>$M$</td>
<td>$FA$</td>
<td>to x. $N :: K$</td>
<td>$C$</td>
<td></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>
<th>$\leadsto$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma$</td>
<td>$N[V/x]$</td>
<td>$B$</td>
<td>$K$</td>
<td>$C$</td>
<td></td>
</tr>
</tbody>
</table>

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

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

| $\Gamma$ | $P$ | $C$ | nil | $C$ |

Transitions

$\Gamma \vdash^k K : B \Rightarrow C$ to mean that $K$ can accompany a computation of type $B$ during evaluation.
Typing rules, read off from the CK-machine

Typing a stack

\[ \Gamma \vdash^k \text{nil} : C \implies C \]

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

\[ \Gamma \vdash^k 1 :: K : B \pi B' \implies C \]

\[ \Gamma \vdash^k \text{to } x. \ M :: K : FA \implies C \]

\[ \Gamma, x : A \vdash^c M : B \]

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

\[ \Gamma \vdash^v V : A \]

\[ \Gamma \vdash^k V :: K : A \to B \implies C \]
Typing rules, read off from the CK-machine

**Typing a stack**

\[
\Gamma \vdash^k \text{nil} : C \implies C
\]

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

\[
\Gamma \vdash^k 1 :: K : B \land B' \implies C
\]

\[
\Gamma, x : A \vdash^c M : B \quad \Gamma \vdash^k K : B \Rightarrow C
\]

\[
\Gamma \vdash^k \text{to x. } M :: K : FA \Rightarrow C
\]

\[
\Gamma \vdash^v V : A \quad \Gamma \vdash^k K : B \Rightarrow C
\]

\[
\Gamma \vdash^k V :: K : A \rightarrow B \Rightarrow C
\]

**Typing a CK-configuration**

\[
\Gamma \vdash^c M : B \quad \Gamma \vdash^k K : B \Rightarrow C
\]

\[
\Gamma \vdash^{ck} (M, K) : C
\]
Given a stack $\Gamma \vdash^k K : B \Rightarrow C$, we can weaken it or substitute values.
Given a stack $\Gamma \vdash^k K : B \Rightarrow C$, we can weaken it or substitute values.

A stack $\Gamma \vdash^k K : B \Rightarrow C$ can be dismantled onto a computation $\Gamma \vdash^c M : B$, giving a computation $\Gamma \vdash^c M \bullet K : C$. 
Given a stack \( \Gamma \vdash^k K : B \rightarrow C \), we can **weaken** it or **substitute** values.

A stack \( \Gamma \vdash^k K : B \rightarrow C \) can be **dismantled** onto a computation \( \Gamma \vdash^c M : B \), giving a computation \( \Gamma \vdash^c M \bullet K : C \).

Stacks \( \Gamma \vdash^k K : B \rightarrow C \) and \( \Gamma \vdash^k L : C \rightarrow D \) can be **concatenated** to give \( \Gamma \vdash^k K \uplus L : B \rightarrow D \).
Continuations

A continuation is a stack from an $F$ type, e.g. $\text{to } x. \ M :: K$. It describes everything that will happen once a value is supplied.
A continuation is a stack from an $F$ type, e.g. `to x. M :: K`. It describes everything that will happen once a value is supplied. In CBV, all computations have $F$ type, so all stacks are continuations.
Continuations

A continuation is a stack from an $F$ type, e.g. $\text{to } x. \quad M :: K$.
It describes everything that will happen once a value is supplied.

In CBV, all computations have $F$ type, so all stacks are continuations.

Top-Level Stack

The top-level stack is $\Gamma \vdash^k \text{nil} : C \Rightarrow C$.
The top-level type is $C$. 
Special Stacks

Continuations

A continuation is a stack from an $F$ type, e.g. $\text{to } x. \ M :: K$.
It describes everything that will happen once a value is supplied.

In CBV, all computations have $F$ type, so all stacks are continuations.

Top-Level Stack

The top-level stack is $\Gamma \vdash^k \text{nil} : {C} \implies {C}$.
The top-level type is ${C}$.

If $C$ is $F\text{bool}$ (the usual situation),
then nil is the top-level continuation:
it receives a boolean and returns it to the user.
Stacks denote homomorphisms

Consider a stack $\Gamma \vdash^k K : B \implies C$

where $[B] = (X, c, b)$ and $[C] = (Y, c', b')$.

What should $K$ denote?
Stacks denote homomorphisms

Consider a stack $\Gamma \vdash^k K : B \rightarrowrightarrow C$

where $[B] = (X, c, b)$ and $[C] = (Y, c', b')$.

What should $K$ denote?

It acts on computations by $M \mapsto M \cdot K$.

So we want $[K] : [\Gamma] \times X \rightarrowrightarrow Y$. 
Stacks denote homomorphisms

Consider a stack $\Gamma \vdash^k K : B \rightarrowrightarrow C$

where $\llbracket B \rrbracket = (X, c, b)$ and $\llbracket C \rrbracket = (Y, c', b')$.

What should $K$ denote?

It acts on computations by $M \mapsto M \cdot K$.

So we want $\llbracket K \rrbracket : \llbracket \Gamma \rrbracket \times X \rightarrowrightarrow Y$.

This function should be homomorphic in its second argument:

\[
\llbracket K \rrbracket(\rho, c) = c'
\]
\[
\llbracket K \rrbracket(\rho, b) = b'
\]

because if $M$ throws an error then so does $M \cdot K$. 

Paul Blain Levy (University of Birmingham)  \(\lambda\)-calculus, effects and call-by-push-value

April 16, 2021 108 / 128
Stacks denote homomorphisms

Consider a stack \( \Gamma \vdash^k K : B \rightarrowtail C \)

where \([B] = (X, c, b)\) and \([C] = (Y, c', b')\).

What should \(K\) denote?

It acts on computations by \( M \mapsto M \bullet K \).

So we want \([K] : [\Gamma] \times X \rightarrow Y\).

This function should be homomorphomic in its second argument:

\[
[K](\rho, c) = c' \\
[K](\rho, b) = b'
\]

because if \(M\) throws an error then so does \(M \bullet K\).

We assume there’s no exception handling.
Operations on stacks

We define $\left[ K \right]$ by induction on $K$.

Then we prove
- a weakening lemma
- a substitution lemma
- a dismantling lemma
- a concatenation lemma

providing a semantic counterpart for each operation on stacks.
Soundness of CK-machine

What should a CK-configuration $\Gamma \vdash^{ck} (M, K) : C$ denote?
Soundness of CK-machine

What should a CK-configuration $\Gamma \vdash^\text{ck} (M, K) : C$ denote?

$$\llbracket (M, K) \rrbracket : [\Gamma] \rightarrow [C]$$

$$\rho \mapsto [K](\rho, [M] \rho)$$

Properties:

1. If $(M, K) \leadsto (M', K')$ then $\llbracket (M, K) \rrbracket = \llbracket (M', K') \rrbracket$.
2. $\llbracket \text{error CRASH}, K \rrbracket \rho = c'$.
3. $\llbracket \text{error BANG}, K \rrbracket \rho = b'$.
We have an adjunction between the category of values (sets and functions) and the category of stacks ($E$-pointed sets and homomorphisms).

\[
\begin{array}{c}
\text{Set} \\ \\
\overset{\bot}{\rightarrow} \\
\underset{U^E}{\leftarrow} \\
\end{array}
\begin{array}{c}
F^E \\
E/\text{Set}
\end{array}
\]

This resolves the exception monad $X \longmapsto X + E$ on $\text{Set}$. 
Consider CBPV extended with two storage cells: 
1 stores a natural number, and 1' stores a boolean.
Consider CBPV extended with two storage cells: 
\( l \) stores a natural number, and \( l' \) stores a boolean.

\[
\begin{align*}
\Gamma \vdash^V V : \text{nat} & \quad \Gamma \vdash^c M : B \\
\Gamma \vdash^c l := V. M : B \\
\Gamma, x : \text{nat} \vdash^c M : B \\
\Gamma \vdash^c \text{read } l \text{ as } x. M : B
\end{align*}
\]
Consider CBPV extended with two storage cells: $1$ stores a natural number, and $1'$ stores a boolean.

\[
\begin{align*}
\Gamma \vdash^v V : \text{nat} & \quad \Gamma \vdash^c M : B \\
\hline
\Gamma \vdash^c 1 := V. M : B \\
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \text{nat} \vdash^c M : B \\
\hline
\Gamma \vdash^c \text{read } 1 \text{ as } x. M : B \\
\end{align*}
\]

A state is $1 \mapsto n, 1' \mapsto b$.

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 \textit{SC-configuration}.

We can type these using

\[
\frac{\Gamma \vdash^c M : B}{\Gamma \vdash^{sc} (s, M) : B} \quad s \in S
\]
Denotational semantics of state

How can we give a denotational semantics for call-by-push-value with state?

- Algebra semantics.
- Intrinsic semantics.
Moggi’s monad for state is $S \rightarrow (S \times -)$. Its Eilenberg-Moore algebras were characterized by Plotkin and Power.
Moggi’s monad for state is $S \rightarrow (S \times -)$.
Its Eilenberg-Moore algebras were characterized by Plotkin and Power.

A value type $A$ denotes a set $\llbracket A \rrbracket$, a semantic domain for values.

A computation type $B$ denotes an Eilenberg-Moore algebra $\llbracket B \rrbracket_{\text{alg}},$ a semantic domain for computations.
Moggi’s monad for state is $S \rightarrow (S \times -)$.
Its Eilenberg-Moore algebras were characterized by Plotkin and Power.

A value type $A$ denotes a set $\llbracket A \rrbracket$, a semantic domain for values.

A computation type $B$ denotes an Eilenberg-Moore algebra $\llbracket B \rrbracket_{\text{alg}}$, a semantic domain for computations.

We complete the story with an adequacy theorem:

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

This requires an SC-configuration to have a denotation.
Intrinsic semantics of state

A value type $A$ denotes a set $\llbracket A \rrbracket$, a semantic domain for values.

A computation type $B$ denotes a set $\llbracket B \rrbracket$, a semantic domain for SC-configurations.
Intrinsic semantics of state

A value type $A$ denotes a set $\llbracket A \rrbracket$, a **semantic domain for values**.

A computation type $B$ denotes a set $\llbracket B \rrbracket$, a **semantic domain for SC-configurations**.

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

\[
\llbracket (s, M) \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket B \rrbracket
\]
A value type $A$ denotes a set $\llbracket A \rrbracket$, a semantic domain for values.

A computation type $B$ denotes a set $\llbracket B \rrbracket$, a semantic domain for SC-configurations.

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

$$\llbracket (s, M) \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket B \rrbracket$$

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

$$\llbracket M \rrbracket : S \times \llbracket \Gamma \rrbracket \rightarrow \llbracket B \rrbracket$$
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 $x : A$ and then behave in $B$.

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

An SC-configuration of type $B \Pi B'$ will pop $l$ and then behave in $B$, or pop $r$ and then behave in $B'$.

$$[B \Pi B'] = [B] \times [B']$$

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

$$[UB] = S \rightarrow [B]$$
State: the value/stack adjunction

Consider a stack \( \Gamma \vdash^k K : B \Rightarrow C \)

What should \( K \) denote?
Consider a stack $\Gamma \vdash^k K : B \Rightarrow C$

What should $K$ denote?

It acts on SC-configurations by $s, M \mapsto s, M \bullet K$.

So we want $[K] : [[\Gamma]] \times [[B]] \to [[C]]$. 


Consider a stack $\Gamma \vdash^k K : B \implies C$

What should $K$ denote?

It acts on SC-configurations by $s, M \mapsto \downarrow s, M \bullet K$.

So we want $\llbracket K \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket B \rrbracket \rightarrow \llbracket C \rrbracket$.

This gives an adjunction

$$\begin{array}{c}
\text{Set} & \xleftarrow{S \times \_} & \xrightarrow{\_ \rightarrow S} & \text{Set} \\
\downarrow \qquad \downarrow & & & \\
\text{Set} & \xleftarrow{\_ \rightarrow S} & \xrightarrow{S \rightarrow \_} & \text{Set}
\end{array}$$

between values and stacks.
For call-by-value we recover

\[
\begin{align*}
\llbracket \text{bool}_{\text{CBV}} \rrbracket &= 1 + 1 \\
\llbracket A \to_{\text{CBV}} B \rrbracket &= \llbracket U(A \to FB) \rrbracket \\
&= S \to (\llbracket A \rrbracket \to (S \times \llbracket B \rrbracket))
\end{align*}
\]

This is standard.
State in call-by-value and call-by-name

For call-by-value we recover

\[
\begin{align*}
[	ext{bool}_{\text{CBV}}] &= 1 + 1 \\
[A \rightarrow_{\text{CBV}} B] &= [U(A \rightarrow FB)] \\
&= S \rightarrow ([A] \rightarrow (S \times [B]))
\end{align*}
\]

This is standard.

For call-by-name we recover

\[
\begin{align*}
[	ext{bool}_{\text{CBN}}] &= [F(1 + 1)] \\
&= S \times (1 + 1) \\
[A \rightarrow_{\text{CBN}} B] &= [UA \rightarrow B] \\
&= (S \rightarrow [A]) \rightarrow [B]
\end{align*}
\]

This is O’Hearn’s semantics of types for a stateful CBN language.
Naming and changing the current stack

Extend the language with two instructions:

- `letstk \( \alpha \)` means *let \( \alpha \) be the current stack*.
- `changestk \( \alpha \)` means *change the current stack to \( \alpha \).*
Naming and changing the current stack

Extend the language with two instructions:

- letstk $\alpha$ means let $\alpha$ be the current stack.
- changestk $\alpha$ means change the current stack to $\alpha$.

Execution takes place in a bigger language.

\[
\begin{array}{l}
\Gamma \quad \text{letstk } \alpha. \ M \quad \frac{B}{K} \quad \frac{C}{\Delta} \\
\Rightarrow \quad \Gamma \quad M[K/\alpha] \quad \frac{B}{K} \quad \frac{C}{\Delta}
\end{array}
\]

\[
\begin{array}{l}
\Gamma \quad \text{changestk } K. \ M \quad \frac{B'}{L} \quad \frac{C}{\Delta} \\
\Rightarrow \quad \Gamma \quad M \quad \frac{B}{K} \quad \frac{C}{\Delta}
\end{array}
\]

Similar to Crolard's syntax. Numerous variations in the literature.
Typing judgements for control

We have typing judgements:

\[ \Gamma \vdash^v V : A \mid \Delta \quad \Gamma \vdash^c M : B \mid \Delta \]

The stack context \( \Delta \) consists of declarations \( \alpha : B \), meaning \( \alpha \) is a stack from \( B \).
Typing judgements for control

We have typing judgements:

\[ \Gamma \vdash^v V : A \mid \Delta \quad \Gamma \vdash^c M : B \mid \Delta \]

The stack context \( \Delta \) consists of declarations \( \alpha : B \), meaning \( \alpha \) is a stack from \( B \).

Example typing rules

\[
\frac{\Gamma \vdash^c M : B \mid \Delta, \alpha : B}{\Gamma \vdash^c \text{letstk} \ \alpha. \ M \mid \Delta}
\]

\[
\frac{\Gamma \vdash^c M : B \mid \Delta}{\Gamma \vdash^c \text{changestk} \ \alpha. \ M : B' \mid \Delta \quad (\alpha : B) \in \Delta}
\]
Typing judgements for execution language

During execution, the top-level type $C$ must be indicated:

$$
\begin{align*}
\Gamma \vdash^v V : A & \quad \Delta \\
\Gamma \vdash^c M : B & \quad [C] \quad \Delta \\
\Gamma \vdash^k K : B & \quad \rightarrow C \quad \Delta \\
\Gamma \vdash^{ck} (M, K) : C & \quad | \quad \Delta
\end{align*}
$$

Typically $\Gamma$ and $\Delta$ would be empty and $C = F \text{bool}$. 
Typing judgements for execution language

During execution, the top-level type $C$ must be indicated:

$$
\Gamma \vdash^v V : A [C] \Delta \quad \Gamma \vdash^c M : B [C] \Delta
\Gamma \vdash^k K : B \rightarrow C \mid \Delta \quad \Gamma \vdash^{ck} (M, K) : C \mid \Delta
$$

Typically $\Gamma$ and $\Delta$ would be empty and $C = F \text{bool}$.

Example typing rules

$$
\Gamma \vdash^k \alpha : B \rightarrow C \mid \Delta \quad (\alpha : B) \in \Delta
$$

$$
\Gamma \vdash^k K : B \rightarrow C \mid \Delta \quad \Gamma \vdash^c M : B [C] \Delta
\Gamma \vdash^c \text{changestk } K. M : B' [C] \Delta
$$
Fix a set $R$, the semantic domain for CK-configurations.

That means: a hypothetical extremely closed CK-configuration, with no free identifiers and no nil, would denote an element of $R$. 
Fix a set $R$, the semantic domain for CK-configurations.

That means: a hypothetical extremely closed CK-configuration, with no free identifiers and no nil, would denote an element of $R$.

Moggi’s monad for control operators (“continuations”) is $(- \rightarrow R) \rightarrow R$. 
Fix a set $R$, the semantic domain for CK-configurations.

That means: a hypothetical extremely closed CK-configuration, with no free identifiers and no $\texttt{nil}$, would denote an element of $R$.

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

Maybe we can build a denotational semantics where a computation type $B$ denotes an Eilenberg-Moore algebra $[B]_{\text{alg}}$, a semantic domain for computations.
The denotation of $\mathcal{B}$ is a semantic domain for stacks from $\mathcal{B}$.

That means: a hypothetical extremely closed stack from $\mathcal{B}$, with no free identifiers and no nil, would denote an element of $[[\mathcal{B}]]$. 
Intrinsic semantics of control

The denotation of $B$ is a semantic domain for stacks from $B$. That means: a hypothetical extremely closed stack from $B$, with no free identifiers and no nil, would denote an element of $[B]$. The behaviour of a computation $\Gamma \vdash^c M : B \mid \Delta$ depends on the environment, current stack and stack environment:

$$[M] : [\Gamma] \times [B] \times [\Delta] \rightarrow R$$

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

$$[V] : [\Gamma] \times [\Delta] \rightarrow [A]$$
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 $B \Pi B'$ is a tagged stack $^1 :: K$ or $^r :: K$.

$$[B \Pi B'] = [B] + [B']$$

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

$$[UB] = [B] \to R$$
The semantics of a term in the execution language depends not only on the environment and the stack environment but also on the top-level stack.
The semantics of a term in the execution language depends not only on the environment and the stack environment but also on the top-level stack.

In particular, a stack $\Gamma \vdash^k K : B \leadsto C \mid \Delta$ denotes

$$[K] : [\Gamma] \times [C] \times [\Delta] \rightarrow [B]$$
The semantics of a term in the execution language depends not only on the environment and the stack environment but also on the top-level stack.

In particular, a stack $\Gamma \vdash^k K : B \rightarrow C | \Delta$ denotes

$$\llbracket K \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket C \rrbracket \times \llbracket \Delta \rrbracket \rightarrow \llbracket B \rrbracket$$

That gives an adjunction

$$\text{Set} \quad \overset{-\rightarrow R} \longrightarrow \quad \text{Set}^{\text{op}}$$

between values and stacks.
Abbreviate \( \neg X \overset{\text{def}}{=} X \rightarrow R \).
Control in call-by-value and call-by-name

Abbreviate \( \neg X \overset{\text{def}}{=} X \rightarrow R \).

For call-by-value we recover

\[
\begin{align*}
\llbracket \text{bool}_{\text{CBV}} \rrbracket &= 1 + 1 \\
\llbracket A \rightarrow_{\text{CBV}} B \rrbracket &= \llbracket U(A \rightarrow FB) \rrbracket \\
&= \neg(\llbracket A \rrbracket \times \neg\llbracket B \rrbracket)
\end{align*}
\]

This is standard.
Control in call-by-value and call-by-name

Abbreviate $\neg X \overset{\text{def}}{=} X \to R$.

For call-by-value we recover

\[
\begin{align*}
[\text{bool}_{\text{CBV}}] &= 1 + 1 \\
[A \to_{\text{CBV}} B] &= [U(A \to FB)] \\
&= \neg([A] \times \neg[B])
\end{align*}
\]

This is standard.

For call-by-name we recover

\[
\begin{align*}
[\text{bool}_{\text{CBN}}] &= [F(1 + 1)] \\
&= \neg(1 + 1) \\
[A \to_{\text{CBN}} B] &= [UA \to B] \\
&= \neg[A] \times [B]
\end{align*}
\]

This is Streicher and Reus’ semantics for a CBN language with control operators.
For a set $E$, the adjunction $\text{Set} \xrightarrow{\perp} E/\text{Set}$

models call-by-push-value with errors.
Summary: adjunctions between values and stacks

For a set \( E \), the adjunction

\[
\begin{array}{c}
\text{Set} \\
\xrightarrow{F^E} \\
\downarrow \\
\xleftarrow{UE} \\
\text{E/Set}
\end{array}
\]

models call-by-push-value with errors.

For a set \( S \), the adjunction

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

models call-by-push-value with state.
Summary: adjunctions between values and stacks

For a set $E$, the adjunction $\text{Set} \xymatrix{ & F^E \ar[r] & E/\text{Set} \ar[l] }$ models call-by-push-value with errors.

For a set $S$, the adjunction $\text{Set} \xymatrix{ & S \times - \ar[r] & \text{Set} \ar[l] }$ models call-by-push-value with state.

For a set $R$, the adjunction $\text{Set} \xymatrix{ & - \rightarrow R \ar[r] & \text{Set}^{\text{op}} \ar[l] }$ models call-by-push-value with control.
Summary: adjunctions between values and stacks

For a set $E$, the adjunction $\text{Set} \xleftarrow{U^E} E/\text{Set} \xrightarrow{F^E} \text{Set}$ models call-by-push-value with errors.

For a set $S$, the adjunction $\text{Set} \xleftarrow{S\to-} \text{Set} \xrightarrow{S\times-} \text{Set}$ models call-by-push-value with state.

For a set $R$, the adjunction $\text{Set} \xleftarrow{-\to R} \text{Set}^{\text{op}} \xrightarrow{-\to R} \text{Set}$ models call-by-push-value with control.