Outline

1 Pure $\lambda$-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 \( \lambda \)-calculus with arithmetic, including not just function types, but also sum and product types.

Here is the syntax of types:

\[
A ::= \text{bool} \mid \text{nat} \mid A \to A \mid 1 \mid A \times A \mid 0 \mid A + A \\
\mid \sum_{i \in \mathbb{N}} A_i \mid \prod_{i \in \mathbb{N}} A_i \quad \text{(optional extra)}
\]
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:

\[
A ::= \text{bool} \mid \text{nat} \mid A \to A \mid 1 \mid A \times A \mid 0 \mid A + A \\
| \sum_{i \in \mathbb{N}} A_i \mid \prod_{i \in \mathbb{N}} A_i \quad \text{(optional extra)}
\]

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

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

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

Introduction rule

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

Elimination rule

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

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 \texttt{bool}

Two introduction rules:

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

Elimination rule

\[
\Gamma \vdash M : \texttt{bool} \quad \Gamma \vdash N : B \quad \Gamma \vdash N' : B
\]

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

It’s a pretentious notation for if $M$ then $N$ else $N'$. 

Paul Blain Levy (University of Birmingham) $\lambda$-calculus, effects and call-by-push-value July 6, 2018 7 / 128
Typing rules for arithmetic

These are \textit{ad hoc} rules.

\[
\begin{align*}
\Gamma \vdash 17 : \text{nat} \\
\hline
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : \text{nat} & \quad \Gamma \vdash M' : \text{nat} \\
\hline
\Gamma \vdash M + M' : \text{nat}
\end{align*}
\]
Typing rules for $A + B$

Two introduction rules

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

Elimination rule

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

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

Two introduction rules

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

Elimination rule

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

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

Zero introduction rules

Elimination rule

\[ \Gamma \vdash M : 0 \]

\[ \Gamma \vdash \operatorname{match} M \text{ as } \{\}^A : A \]
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

\[
\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M^1 : A}
\]

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

Introduction rule

$$
\Gamma \vdash M : A \quad \Gamma \vdash N : B \\
\Gamma \vdash \langle M, N \rangle : A \times B
$$

Two options for elimination

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

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

- **Projection product.** Two elimination rules

  $$
  \Gamma \vdash M : A \times B \\
  \Gamma \vdash M^1 : A \\
  \Gamma \vdash M^r : B
  $$

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

Introduction rule

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

Two options for elimination

- **Pattern-match unit.** Elimination rule

  \[
  \Gamma \vdash M : 1 \quad \Gamma \vdash N : C \\
  \Gamma \vdash \text{match } M \text{ as } \langle \rangle . N : C
  \]

- **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

Substitution is an operation on binding diagrams, not on terms.

Example

\[ M = \lambda y \text{nat}. y + 3 \]
\[ M' = 7 \]
\[ N = x(5 + y) \]
\[ N[\frac{M}{x}, \frac{M'}{y}] = (\lambda z \text{nat}. z + 3)(5 + 7) \]
Substitution

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 $\llbracket A \rrbracket$.

For example, $\llbracket \text{nat} \to \text{nat} \rrbracket$ is the set of functions $\mathbb{N} \to \mathbb{N}$. 
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 $\lambda a \in \mathbb{N}. a + 3$. 
Semantics of types

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 $\{\text{inl } x \mid x \in X\} \cup \{\text{inr } y \mid y \in Y\}$.

\[
\begin{align*}
[\text{bool}] & = \mathbb{B} = \{\text{true}, \text{false}\} \\
[\text{nat}] & = \mathbb{N} \\
[A \rightarrow B] & = [A] \rightarrow [B] \\
[1] & = 1 = \{\langle \rangle \} \\
[A + B] & = [A] + [B] \\
[A \times B] & = [A] \times [B] \\
[0] & = \emptyset
\end{align*}
\]
Semantic environments

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 $\llbracket M \rrbracket$, or more precisely $\llbracket \Gamma \vdash M : A \rrbracket$. It's a function from $\llbracket \Gamma \rrbracket$ to $\llbracket A \rrbracket$.

Example

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

denotes the function

\[
\llbracket x : \text{nat}, y : \text{nat} \rrbracket \quad \longrightarrow \quad (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}
\]

$\rho \quad \longmapsto \quad \lambda z \in \mathbb{N} \to \mathbb{N}. z(\rho_x + \rho_y)$
Semantics of terms

\[
\begin{align*}
\Gamma \vdash 17 : \text{nat} \\
[17] : \rho \mapsto 17 \\
\Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat} \\
\Gamma \vdash M + M' : \text{nat} \\
[M + M'] : \rho \mapsto [M] \rho + [M'] \rho
\end{align*}
\]
More semantic equations

\[ \Gamma 
\vdash \begin{array}{l}
(x : A) \in \Gamma \\
\chi : \rho \mapsto \rho_x
\end{array}
\]

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

\[ \Gamma \vdash \lambda x_A \cdot M : A \to B \]

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

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

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

\[
\llbracket \text{match } M \text{ as } \{\text{inl } x. N, \text{inr } y. N'\} \rrbracket : \rho \mapsto \text{match } \llbracket M \rrbracket \rho \text{ as } \{\text{inl } a. \llbracket N \rrbracket (\rho, x \mapsto a), \text{inr } b. \llbracket N' \rrbracket (\rho, x \mapsto b)\}
\]
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 $\left[ \Gamma \vdash M : A \right]$ doesn’t depend on the derivation.

### Weakening Lemma

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

$$\left[ \Gamma' \vdash M : A \right] \rho = \left[ \Gamma \vdash M \right] (\rho \upharpoonright \Gamma)$$
Substitution

**Binding Diagrams**

- 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.
We can give denotational semantics of binding diagrams.

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

So α-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 \([M]\) and \([N]\).

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

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

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 bool

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

Two \( \beta \)-laws for \( \text{bool} \)

\[
\Gamma \vdash N : C \quad \Gamma \vdash N' : C
\]

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

Two \( \beta \)-laws for \( A + B \)

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

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

Two $\beta$-laws for bool

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

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

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

Zero $\beta$-laws for 0
$$\Gamma \vdash M : A \quad \Gamma \vdash M' : B \quad \Gamma, x : A, y : B \vdash N : C$$

$$\Gamma \vdash \text{let } (x \text{ be } M, \ y \text{ be } M'). \ N = N[M/x, M'/y] : C$$
$\eta$-laws

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

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

Introduction outside an elimination may be inserted.
η-laws

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

\[
\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash M = \lambda x_{A}. M\ x : A \to B} \quad x \not\in \Gamma
\]

Introduction outside an elimination may be inserted.

η-law for projection product \( A \times B \), everything is \( \lambda \)

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

η-law for projection unit 1, everything is \( \lambda \)

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

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

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

$z \notin \Gamma$
More $\eta$-laws

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

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

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

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

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

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

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

$z \notin \Gamma$

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

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

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

$z \notin \Gamma$

$\eta$-law for 0, nothing exists

$$\Gamma \vdash M : 0 \quad \Gamma, z : 0 \vdash N : C$$

$$\Gamma \vdash N[M/z] = \text{match } M \text{ as } \{\} : C$$

$z \notin \Gamma$
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:

\[
\Gamma, x : A \vdash M = M' : B \\
\Gamma \vdash \lambda x_A. M = \lambda x_A. M' : A \to B
\]
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 $\llbracket M \rrbracket = \llbracket M' \rrbracket$.

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

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

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

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

Reversible rule for $A \rightarrow B$

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

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

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 $\Gamma \vdash N : A \rightarrow B$ is sent to $N x$.
- These are inverse up to $=_{\beta\eta}$.

$A \rightarrow B$ appears on the right of $\vdash$ in the conclusion.
The (nullary) connective bool is leftist. That means: it has a reversible rule

\[
\frac{\Gamma \vdash C \quad \Gamma \vdash C}{\Gamma, z : \text{bool} \vdash C}
\]

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.
The connective $+ \hspace{1em}$ is leftist, having a reversible rule

$$
\Gamma, x : A \vdash C \quad \Gamma, y : B \vdash C \\
\hline
\Gamma, z : A + B \vdash C
$$

natural in $\Gamma$ and $C$.  

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

$$
\frac{
}{
\Gamma, z : 0 \vdash C
}
$$

natural in $\Gamma$ and $C$. 
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.
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.
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

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

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

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

This gives a leftist connective.

$$\Gamma, A, A' \vdash C \quad \Gamma, B, B', B'' \vdash C$$

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

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

This gives a leftist connective.

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

Here is its term syntax:

\[
\text{match } M \text{ as } \{ \text{in}_0(x, x'). N, \text{in}_1(y, y', y''). N' \}
\]
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

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

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

\[
\left( ([A] \times [A']) \to [B] \right) \times \left( ([C] \times [C'] \times [C'']) \to [D] \right)
\]

This gives a rightist connective.

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

Here is its term syntax:

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

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

\[
M^1 (N, N', N'')
\]
Type syntax

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

Term syntax, with type annotations omitted

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

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

$$\mid \text{in}_i(\overrightarrow{M})$$

$$\mid \text{match } M \text{ as } \{ \text{in}_i(\overrightarrow{x}). M_i \}_{i<n}$$

$$\mid \lambda\{^i(\overrightarrow{x}). M_i \}_{i<n}$$

$$\mid M^i(\overrightarrow{M})$$

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 $=_{\beta\eta}$ 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 \( \text{let} (x \text{ be } M, y \text{ be } M'). N, \) 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?
Three decisions we must make

1. To evaluate \( \text{let (x be } M, \ y \text{ be } M'). \ N \), 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 \( P[N/x] \)?
Three decisions we must make

1. To evaluate \( \text{let (x be } M, \ y \text{ be } M'). N \), 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?
One fundamental decision

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 $\text{let (x be } M, \ y \text{ be } M'). \ N$, 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**
Terminal terms of type $A + B$

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 $T ::= \text{true} \mid \text{false} \mid \text{inl } T \mid \text{inr } T \mid \lambda x. M$

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 $\text{inl } T$.
- $\text{let (x be } M, \ y be } M'). N$: evaluate $M$. If this returns $T$, evaluate $M'$. If this returns $T'$, evaluate $N[T/x, T'/y]$.
- $\text{match } M \text{ as } \{ \text{true. } N, \ \text{false. } N' \} \text{: evaluate } M$. If this returns $\text{true}$, evaluate $N$, but if it returns $\text{false}$, evaluate $N'$.
- $\text{match } M \text{ as } \{ \text{inl } x. \ N, \ \text{inr } x. \ N' \} \text{: evaluate } M$. If this returns $\text{inl } T$, evaluate $N[T/x]$, but if it returns $\text{inr } T$, evaluate $N'[T/x]$.
- $MN$: evaluate $M$. If this returns $\lambda x. P$, evaluate $N$. If this returns $T$, evaluate $P[T/x]$.
Definitional interpreter for call-by-name

In CBN the terminals are `true, false, inl M, inr M, λx.M`

To evaluate

- **true**: return `true`.
- **M + N**: evaluate `M`. If this returns `m`, evaluate `N`. If this returns `n`, return `m + n`.
- **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 `λx.P`, evaluate `P[N/x]`.

Paul Blain Levy (University of Birmingham)  
λ-calculus, effects and call-by-push-value  
July 6, 2018 47 / 128
Big-step semantics for call-by-value

We write $M \Downarrow T$ to mean that $M$ evaluates to $T$.

This is defined inductively, for example

\[
\frac{M \Downarrow \lambda x_A. P \quad N \Downarrow T \quad P[T/x] \Downarrow T'}{M \; N \Downarrow T'}
\]
We write $M \Downarrow T$ to mean that $M$ evaluates to $T$.

This is defined inductively, for example

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

Therefore $M N \Downarrow T'$.

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

Moreover $\vdash T : A$ and $[M] = [T]$. 

We write $M \downarrow T$ to mean that $M$ evaluates to $T$. This is defined inductively, for example

\[
\begin{align*}
M &\downarrow \lambda x_A. P \\
P[N/x] &\downarrow T
\end{align*}
\]

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

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

Therefore

$$
MN \Downarrow T
$$

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

Moreover $\vdash T : A$ and $\llbracket M \rrbracket = \llbracket T \rrbracket$. 
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.
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

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

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

\[ \Gamma \vdash M : B \]

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

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

1. Evaluate
   \[
   \text{let (x be error CRASH). 5}
   \]
   in CBV and CBN.

2. Evaluate
   \[
   (\lambda x. (x + x))(\text{print "hello"}. 4)
   \]
   in CBV and CBN.

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

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

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

Likewise for call-by-name.
Observational equivalence

A program is a closed term of type \( \text{nat} \) or \( \text{bool} \).

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,

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

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 $\lambda \{^{l}.M^{l}, ^{r}.M^{r}\}$.

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

- $\text{error } e \simeq_{\text{CBN}} \lambda x. \text{error } e$
- $\text{error } e \simeq_{\text{CBN}} \lambda \{^{l}.\text{error } e, ^{r}.\text{error } e\}$
- $\text{print } c. \lambda x. M \simeq_{\text{CBN}} \lambda x. \text{print } c. M$
- $\text{print } c. \lambda \{^{l}.M, ^{r}.N\} \simeq_{\text{CBN}} \lambda \{^{l}.\text{print } c. M, ^{r}.\text{print } c. N\}$

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

In CBN, a function gets evaluated only by being applied.
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).
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.
- $A \rightarrow (B \rightarrow C) \cong (A \Pi B) \rightarrow C$ survives in CBN but not CBV.
Naive CBV semantics

Our first attempt.

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

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

Our first attempt.

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

$$
[\text{bool}]_* = B + E
$$
$$
[\text{bool} + \text{bool}]_* = (B + B) + E
$$
$$
[\text{bool} \times \text{bool}]_* = (B \times B) + E
$$

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

\[
\begin{align*}
[\text{bool}] &= \mathbb{B} \\
[A + B] &= [A] + [B] \\
[A \to B] &= [A] \to ([B] + E) \\
[() \to B] &= [B] + E \\
[\Gamma] &= \prod_{(x:A) \in \Gamma} [A]
\end{align*}
\]
CBV denotational semantics

Each type denotes a set, a **semantic domain for terminals**.

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

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

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

$$\Gamma \vdash \lambda x \in A. M : A \rightarrow B$$

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

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

$$\Gamma \vdash MN : B$$

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

\[ \Gamma \vdash M : A \]

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

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

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

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

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

\[
M \overset{\text{def}}{=} \text{error CRASH} \\
N \overset{\text{def}}{=} \text{true} \\
N' \overset{\text{def}}{=} \text{match } x \text{ as } \{\text{true. true, false. true}\} \\
[N] = [N'] \quad \text{because } N =_{\eta_{\text{bool}}} N' \\
[N[M/x]] \neq [N'[M/x]]
\]
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 \(\vdash x : \text{bool}\) \(\vdash N, N' : \text{bool}\).

\[
\begin{align*}
M & \overset{\text{def}}{=} \text{error CRASH} \\
N & \overset{\text{def}}{=} \text{true} \\
N' & \overset{\text{def}}{=} \text{match } x \text{ as \{true.true, false.true\}} \\
[N] & = [N'] \quad \text{because } N =_{\eta_{\text{bool}}} N' \\
[N[M/x]] & \neq [N'[M/x]]
\end{align*}
\]

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

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

- $[x]^{val} : \rho \mapsto \rho_x$
- $[true]^{val} : \rho \mapsto true$
- $[inl V]^{val} : \rho \mapsto inl [V]^{val} \rho$
- $[\lambda x_A . M]^{val} : \rho \mapsto \lambda a \in [A]. [M](\rho, x \mapsto [a])$

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

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

Given values $\Gamma \vdash V : A$ and $\Gamma \vdash^v W : B$ and a term $\Gamma, x : A, y : B \vdash M : C$

we can obtain $[M[V/x, W/y]]$ from $[V]^{val}$ and $[W]^{val}$ and $[M]$.

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

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 $\llbracket V \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$.
- A computation $\Gamma \vdash^c M : A$ denotes a function $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket + E$.

Key typing rules

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

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

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

\[ \begin{align*}
\Gamma \vdash^v V : A \\
\Gamma \vdash^c \text{return } V : A \\
[\text{return } V] : \rho \mapsto \text{inl } [V]\rho \\
\Gamma \vdash^c M : A \quad \Gamma, x : A \vdash^c N : B \\
\Gamma \vdash^c M \text{ to } x. N : B \\
[\text{match } M] : \rho \mapsto \text{match } [M]\rho \text{ as } \left\{ \begin{array}{l}
\text{inl } a. \quad [N](\rho, x \mapsto a) \\
\text{inr } e. \quad \text{ inr } e
\end{array} \right.
\end{align*} \]
Syntax

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

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

We don't allow "complex values" such as

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

These would complicate the operational semantics.
Syntax

For connectives bool, +, \( \rightarrow \) the syntax is as follows.

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

We don’t allow “complex values” such as

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

These would complicate the operational semantics.
We evaluate a closed computation $\vdash^c M : A$ to a closed value $\vdash^y V : A$. 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 $M[V/x]$.
- **match inl $V$ as {inl $x$. $N$, inr $x$. $N'$}:** evaluate $N[V/x]$. 
Equational theory

\(\beta\)-laws

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

\(\eta\)-laws

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

Sequencing laws

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

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

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

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.

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

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

\[ \Gamma \vdash^c A \]
\[ \Gamma \vdash^v TA \]
Call-by-value programmers use nullary functions to delay evaluation, and call them thunks.

\[
TA \overset{\text{def}}{=} () \rightarrow A \quad [TA] = [A] + E
\]

\[
\text{thunk } M \overset{\text{def}}{=} \lambda().M \quad [\text{thunk } M] = [M]
\]

\[
\text{force } V \overset{\text{def}}{=} V() \quad [\text{force } V] = [V]
\]

The type \( TA \) has a reversible rule

\[
\Gamma \vdash^c A \quad \overline{\quad} \quad \Gamma \vdash^v TA
\]

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:

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

Thus we define

\[ [\text{bool}]_* = \mathbb{B} + E \]
\[ [A + B]_* = ([A]_* + [B]_*) + E \]
\[ [A \to B]_* = [A]_* \to [B]_* \]
\[ [A \Pi B]_* = [A]_* \times [B]_* \]
\[ [\Gamma] = \prod_{(x:A) \in \Gamma} [A]_* \]

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

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

denotes \(\rho \mapsto ?\)

Example: suppose \(B = \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})\) then \(B\) denotes \((B + E) \rightarrow ((B + E) \rightarrow (B + E))\)

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

A similar problem arises with match.

Paul Blain Levy (University of Birmingham)
\(\lambda\)-calculus, effects and call-by-push-value
July 6, 2018 77 / 128
Naive semantics: what goes wrong

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

denotes \( \rho \mapsto ? \)

Example:
- suppose \( B = \text{bool} \to (\text{bool} \to \text{bool}) \)
- then \( B \) denotes \( (\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E)) \)
- 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

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

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 \( \text{error CRASH} \sim_{\text{CBN}} \lambda x. \lambda y. \text{error CRASH} \)
- 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.
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.
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:

\[
\begin{align*}
[\text{bool} \to (\text{bool} \to \text{bool})] & = ((\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E))), \\
& \quad \lambda x.\lambda y.\text{inr CRASH}, \\
& \quad \lambda x.\lambda y.\text{inr BANG}) \\
[\text{bool} + \text{bool}] & = (((\mathbb{B} + E) + (\mathbb{B} + E)) + E, \\
& \quad \text{inr CRASH}, \\
& \quad \text{inr BANG}) \\
[\text{bool} \Pi \text{bool}] & = ((\mathbb{B} + E) \times (\mathbb{B} + E), \\
& \quad (\text{inr CRASH}, \text{inr CRASH}), \\
& \quad (\text{inr BANG}, \text{inr BANG}))
\end{align*}
\]
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'))
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 \( [A \Pi B] = (X \times Y, (c, c'), (b, b')) \)

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

\( \text{[\Gamma]} = (X, c, b) \)

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

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

\[
\begin{array}{c}
\Gamma \vdash \text{true} : \rho \mapsto \text{inl true} \\
\end{array}
\]

\[
\Gamma \vdash M : \text{bool} \quad \Gamma \vdash N : B \quad \Gamma \vdash N' : B
\]

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

\[
\begin{array}{c}
\text{match } [M]_{\rho} \text{ as } \begin{cases}
\text{inl true.} & [N]_{\rho} \\
\text{inl false.} & [N']_{\rho} \\
\text{inr CRASH.} & c \\
\text{inr BANG.} & b
\end{cases} \quad \text{where } [B] = (Y, c, b)
\end{array}
\]
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 \( M \not\Downarrow \text{CRASH} \) then \( \llbracket M \rrbracket \varepsilon = c \).
- If \( M \not\Downarrow \text{BANG} \) 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.
  \[ (X, c, b) \prod (Y, c', b') \overset{\text{def}}{=} (X \times Y, (c, c'), (b, b')) \]
- Unit $E$-pointed set.
  \[ 1_\prod \overset{\text{def}}{=} (1, ()), () \]
- 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.
  \[ X \rightarrow (Y, c, b) \overset{\text{def}}{=} \prod_{x \in X} (Y, c, b) \]
  \[ = (X \rightarrow Y, \lambda x. c, \lambda x. b) \]
- 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.

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

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.

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

A typing context denotes a set.

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

A computation \( \Gamma \vdash^c M : B \) denotes a function \([\Gamma] \to F^E[B]\).
Two kinds of type:

- A **value type** denotes a set.
- A **computation type** denotes an $E$-pointed set.
Two kinds of type:

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

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

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

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

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 $\Gamma \vdash^v V : A$ 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”.

Paul Blain Levy (University of Birmingham)  λ-calculus, effects and call-by-push-value  July 6, 2018  87 / 128
The type $FA$

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

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

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

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

$$\llbracket \text{return } V \rrbracket : \rho \mapsto \text{inl } \llbracket V \rrbracket \rho$$

$$\llbracket M \to x. N \rrbracket : \rho \mapsto$$

match $\llbracket M \rrbracket \rho$ as

$$\begin{cases} 
\text{inl } a. & \llbracket N \rrbracket (\rho, x \mapsto a) \\
\text{inr } \text{CRASH}. & c \\
\text{inr } \text{BANG}. & b 
\end{cases}$$

where $\llbracket B \rrbracket = (Y, c, b)$
The type $UB$

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

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

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

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

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

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

\[
\Gamma \vdash^c \text{let} (x \text{ be } V, y \text{ be } W). M : C
\]
The rules for 1 are similar.
It is often convenient to write applications operand-first, as \( V \ M \), \( l \ M \), and \( r \ M \).
Functions

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

\[
\Gamma \vdash^c M : A \to B \\
\Gamma \vdash^v V : A \\
\Gamma \vdash^c MV : B
\]

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

\[
\Gamma \vdash^c M : B \mathbin\Pi B' \\
\Gamma \vdash^c M^1 : B
\]

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

It is often convenient to write applications operand-first, as \(V \cdot M\) and \({^1}\cdot M\) and \({^r}\cdot M\).
The terminals are \textit{computations}: \begin{align*}
\text{return } V & \quad \lambda x. M & \quad \lambda \{^1 M, \text{r. } M'\}
\end{align*}
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 (\( x \) be \( V \), \( y \) be \( W \)). \( M \)**: evaluate \( M[V/x, W/y] \).
- **force thunk \( M \)**: evaluate \( M \).
- **match \( \text{inl } V \) as \{\( \text{inl } x. M, \text{inr } y. M'\)\}:** 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

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

\(\eta\)-laws

\[
\begin{align*}
V &= \text{thunk force } V \\
M[V/z] &= \text{match } V \text{ as } \{ \text{inl } x. M[\text{inl } x/z], \text{inr } y. M[\text{inr } x/z] \} \\
M &= \lambda x. Mx
\end{align*}
\]

Sequencing laws

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

A CBV type translates into a value type.

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

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

A CBV type translates into a value type.

\[ A \to B \leftrightarrow U(A \to FB) \]
\[ TB \leftrightarrow UFB \]

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.

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

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

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

A CBV type translates into a value type.

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

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 \( x : A, y : B \vdash^c M : FC \)

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

A CBN type translates into a computation type.

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

A CBN type translates into a computation type.

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

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

\[
\begin{align*}
x & \quad \mapsto \quad \text{force } x \\
\text{let (x be } M, \text{ y be } M'). N & \quad \mapsto \quad \text{let (x be thunk } M, \text{ y be thunk } M'). N \\
\lambda x . M & \quad \mapsto \quad \lambda x . M \\
M \; N & \quad \mapsto \quad M \; (\text{thunk } N) \\
\text{inl } M & \quad \mapsto \quad \text{return inl thunk } M
\end{align*}
\]
Summary

We’ve seen

- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.
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.
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 $\mathcal{T}A$ 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 $M \text{ to } x.\ N$: evaluate $M$. If this returns return $V$, then evaluate $N[V/x]$.

\[
M \text{ to } x.\ N \quad K \quad \leadsto
\]

\[
M \quad \text{to } x.\ N :: K
\]

\[
\text{return } V \quad \text{to } x.\ N :: K \quad \leadsto
\]

\[
N[V/x] \quad K
\]
Transitions for application

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

$$
\begin{array}{c}
V' M & K \\
\Rightarrow \\
M & V :: K
\end{array}
$$

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

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

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

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

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

Revisiting some equations:

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

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

error \( e = \lambda x. \text{error} e \)

\( \text{print} \ c. \lambda x. M = \lambda x. \text{print} c. M \)
Those function rules again

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

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

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

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

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

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

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

Revisiting some equations:

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

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

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

\[
\text{print} \; c. \; \lambda x. M \; = \; \lambda x. \; \text{print} \; c. \; M
\]
Values and Computations

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 $B \Pi B'$ 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 `to x. N`. 
Example program of type $F \text{nat}$ (with complex values)

print "hello0".
let (x be 3,
    y be thunk (print "hello1".
                $\lambda z$. 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>$\Rightarrow$</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>$\Rightarrow$</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 $C = F\ \text{bool}$. 
Typing the CK-machine

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

$$\begin{array}{|c|c|c|}
\hline
\Gamma & P & C \\
\hline
\text{nil} & C \\
\hline
\end{array}$$

Transitions

$$\begin{array}{|c|c|c|c|}
\hline
\Gamma & M \text{ to } x. N & B & K & C \\
\hline
\Gamma & M & FA & \text{ to } x. N :: K & C \\
\hline
\Gamma & \text{return } V & FA & \text{ to } x. N :: K & C \\
\Gamma & N[V/x] & B & K & C \\
\hline
\end{array}$$

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

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

Typing a stack

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash^k \text{nil} : C \rightarrow C$</td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash^k K : B \rightarrow C$</td>
<td>$\Gamma, x : A \vdash^c M : B$</td>
</tr>
<tr>
<td>$\Gamma \vdash^k \text{to } x. M :: K : FA \rightarrow C$</td>
<td>$\Gamma \vdash^k K : B \rightarrow C$</td>
</tr>
<tr>
<td>$\Gamma \vdash^k 1 :: K : B \Pi B' \rightarrow C$</td>
<td>$\Gamma \vdash^v V : A$</td>
</tr>
<tr>
<td>$\Gamma \vdash^k V :: K : A \rightarrow B \rightarrow C$</td>
<td>$\Gamma \vdash^k K : B \rightarrow C$</td>
</tr>
</tbody>
</table>
Typing rules, read off from the CK-machine

### Typing a stack

- $\Gamma \vdash^k \text{nil} : C \rightarrow C$
- $\Gamma \vdash^k K : B \rightarrow C$
- $\Gamma \vdash^k 1 : K : B \times B' \rightarrow C$
- $\Gamma, x : A \vdash^c M : B$
- $\Gamma \vdash^k K : B \rightarrow C$
- $\Gamma \vdash^k \text{to } x. M :: K : FA \rightarrow C$
- $\Gamma \vdash^v V : A$
- $\Gamma \vdash^k K : B \rightarrow C$
- $\Gamma \vdash^k V :: K : A \rightarrow B \rightarrow C$

### Typing a CK-configuration

- $\Gamma \vdash^c M : B$
- $\Gamma \vdash^k K : B \rightarrow C$
- $\Gamma \vdash^{c^k} (M, K) : C$
Operations on Stacks

1. Given a stack $\Gamma \vdash^k K : B \Rightarrow C$, we can **weaken** it or **substitute** values.
Operations on Stacks

1. Given a stack $\Gamma \vdash^k K : B \Rightarrow C$, we can weaken it or substitute values.

2. A stack $\Gamma \vdash^k K : B \Rightarrow C$ can be dismantled onto a computation $\Gamma \vdash^c M : B$, giving a computation $\Gamma \vdash^c M \bullet K : C$. 
Operations on Stacks

1. Given a stack $\Gamma \vdash^k K : B \Rightarrow C$, we can **weaken** it or **substitute** values.

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

3. Stacks $\Gamma \vdash^k K : B \Rightarrow C$ and $\Gamma \vdash^k L : C \Rightarrow D$ can be **concatenated** to give $\Gamma \vdash^k K \oplus L : B \Rightarrow D$. 
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. $\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.
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 $\Gamma \vdash k\ \text{nil} : \underbrace{C} \Longrightarrow \underbrace{C}$. The **top-level type** is $\underbrace{C}$.
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 $\Gamma \vdash^k \text{nil} : C \Rightarrow C$.

The top-level type is $C$.

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

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

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

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 $M \mapsto M \bullet K$.

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

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

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

This function should be homomorphic in its second argument:

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

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

because if \( M \) throws an error then so does \( M \cdot K \).
Stacks denote homomorphisms

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

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 homomorphic 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 $[K]$ 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 $\Gamma \vdash_{ck} (M, K) : C$ denote?

$$\llbracket (M, K) \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket C \rrbracket \quad \rho \mapsto \llbracket K \rrbracket (\rho, \llbracket M \rrbracket \rho)$$

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).

$$\text{Set} \overset{\bot}{\underset{U^E}{\leftrightarrow}} E/\text{Set} \overset{F^E}{\rightarrow}$$

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

\[
\Gamma \vdash^v V : \text{nat} \quad \Gamma \vdash^c M : B \\
\Gamma \vdash^c l := V. M : B \\
\Gamma, x : \text{nat} \vdash^c M : B \\
\Gamma \vdash^c \text{read} l \text{ as } x. M : B
\]
Consider CBPV extended with two storage cells: 
\(l\) stores a natural number, and \(l'\) stores a boolean.

\[
\Gamma \vdash^v V : \text{nat} \quad \Gamma \vdash^c M : B \\
\Gamma \vdash^c l := V. M : B \\
\Gamma, x : \text{nat} \vdash^c M : B \\
\Gamma \vdash^c \text{read } l \text{ as } x. M : B
\]

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 \textbf{SC-configuration}.

We can type these using

$$
\Gamma \vdash^c M : B
\frac{\Gamma \vdash^{sc} (s, M) : B}{s \in S}
$$
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 $\llbracket B \rrbracket_{alg}$, 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:

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

This requires an SC-configuration to have a denotation.
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 $\Gamma \vdash_{sc} (s, M) : \llbracket B \rrbracket$ depends on the environment:

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

A computation type \( B \) denotes a set \( \lbrack B \rbrack \), a semantic domain for SC-configurations.

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

The behaviour of a computation \( \Gamma \vdash^{c} M : B \) depends on the state and environment:
\[
\lbrack M \rbrack : S \times \lbrack \Gamma \rbrack \rightarrow \lbrack B \rbrack
\]
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 $A \to B$ will pop $x : A$ and then behave in $B$.

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

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

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

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

$$[UB] = S \to [B]$$
Consider a stack $\Gamma \vdash^k K : B \implies C$

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

What should $K$ denote?

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

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

State: the value/stack adjunction

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

What should $K$ denote?

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


This gives an adjunction

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

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

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

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

For call-by-value we recover

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

This is standard.

For call-by-name we recover

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

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:
- \text{letstk } \alpha \text{ means } \text{let } \alpha \text{ be the current stack.}
- \text{changestk } \alpha \text{ means change the current stack to } \alpha.

Execution takes places in a bigger language.

\[
\Gamma \quad \text{letstk } \alpha. \ M \quad B \quad K \quad C \mid \Delta \quad \rightsquigarrow
\]
\[
\Gamma \quad M[K/\alpha] \quad B \quad K \quad C \mid \Delta
\]

\[
\Gamma \quad \text{changestk } K. \ M \quad B' \quad L \quad C \mid \Delta \quad \rightsquigarrow
\]
\[
\Gamma \quad M \quad B \quad K \quad C \mid \Delta
\]

Similar to Crolard's syntax. Numerous variations in the literature.
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:

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

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

Example typing rules

\[
\begin{align*}
\Gamma & \vdash^c \mathit{M} : B \mid \Delta, \alpha : B \\
\hline
\Gamma & \vdash^c \mathit{letstk} \, \alpha. \, \mathit{M} \mid \Delta \\
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash^c \mathit{M} : B \mid \Delta \\
\hline
\Gamma & \vdash^c \mathit{changestk} \, \alpha. \, \mathit{M} : B' \mid \Delta \quad (\alpha : B) \in \Delta \\
\end{align*}
\]
During execution, the top-level type $C$ must be indicated:

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

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:

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

Typically $\Gamma$ and $\Delta$ would be empty and $C = F\布尔$.

Example typing rules

$$
\Gamma \vdash^k \alpha : B \rightarrow C \ | \ \Delta \\
\Gamma \vdash^{ck} M : B \ [C] \ \Delta \\
\Gamma \vdash^{ck} \text{changestk } K. \ M : B' \ [C] \ \Delta
$$
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 $\texttt{nil}$, would denote an element of $R$.

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

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.
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 denotation of \( B \) is a semantic domain for \textit{stacks from} \( B \).

That means: a hypothetical \textit{extremely closed} stack from \( B \), with no free identifiers and no \textit{nil}, would denote an element of \( \llbracket B \rrbracket \).

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

\[
\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket B \rrbracket \times \llbracket \Delta \rrbracket \rightarrow R
\]

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

\[
\llbracket V \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \rightarrow \llbracket A \rrbracket
\]
Control: semantics of types

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

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

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

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

A stack from $B \Pi B'$ is a tagged stack $\downarrow :: K$ or $\uparrow :: K$.

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

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

$$[UB] = [B] \rightarrow R$$
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 $\Gamma \vdash^k K : B \Rightarrow C \mid \Delta$ denotes

$$[K] : [\Gamma] \times [C] \times [\Delta] \longrightarrow [B]$$
Semantics of the execution language

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 $\Gamma \vdash^k K : B \Longrightarrow C \mid \Delta$ denotes

$$[K] : [\Gamma] \times [C] \times [\Delta] \longrightarrow [B]$$

That gives an adjunction

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

between values and stacks.
Abbreviate $\neg X \overset{\text{def}}{=} X \rightarrow R$.
Abbreviate $\neg X \equiv X \to R$.

For call-by-value we recover

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

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

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

For call-by-value we recover

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

This is standard.

For call-by-name we recover

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

This is Streicher and Reus’ semantics for a CBN language with control operators.
For a set $E$, the adjunction $\text{Set} \xleftarrow{\perp} E/\text{Set} \xrightarrow{\perp} \text{Set}$ models call-by-push-value with errors.
For a set $E$, the adjunction $\text{Set} \xymatrix@C=50pt{\perp \ar[r]^{F^E} & \ar[l]_{U^E} E/\text{Set}}$
models call-by-push-value with errors.

For a set $S$, the adjunction $\text{Set} \xymatrix@C=50pt{\perp \ar[r]^{S \times -} & \ar[l]_{S \rightarrow -} \text{Set}}$
models call-by-push-value with state.
Summary: adjunctions between values and stacks

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

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

For a set $R$, the adjunction $\text{Set} \xrightarrow{\bot} \text{Set}^{\text{op}}$ models call-by-push-value with control.
Summary: adjunctions between values and stacks

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

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

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