Paul Blain Levy

University of Birmingham

April 18, 2017
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:

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

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

\[ \Gamma \vdash 17 : \text{nat} \]

\[ \Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat} \]

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

Two introduction rules

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

Elimination rule

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

$$
\Gamma \vdash M : 0
\hline
\Gamma \vdash \text{match } M \text{ as } \emptyset^A : A
$$
Typing rules for $A \times B$, pattern-match syntax

Introduction rule

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

Elimination rule

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

Two elimination rules

\[
\begin{align*}
\Gamma \vdash M : A \times B & \quad \Gamma \vdash M : A \times B \\
\Gamma \vdash M^1 : A & \quad \Gamma \vdash M^r : B
\end{align*}
\]
Typing rules for $A \times B$, projection syntax

Two elimination rules

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

Introduction rule

\[
\Gamma \vdash M : A \quad \Gamma \vdash N : B \quad \Gamma \vdash \lambda \{^1. M, \; ^r. N\} : A \times B
\]
Two elimination rules

\[
\Gamma \vdash M : A \times B \\
\Gamma \vdash M^1 : A \\
\Gamma \vdash M^r : B
\]

Introduction rule

\[
\Gamma \vdash M : A \\
\Gamma \vdash N : B \\
\Gamma \vdash \lambda\{^1M, ^rN\} : A \times B
\]

Likewise for \( \prod_{i \in \mathbb{N}} A_i \).
Typing rules for $\text{1}$, pattern-match and projection

**Introduction rule**

\[
\begin{array}{c}
\Gamma \vdash \langle \rangle : 1 \\
\Gamma \vdash \lambda\{\} : 1
\end{array}
\]

**Elimination rule for pattern-match syntax**

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

**Zero elimination rules for projection syntax**
Theorem

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

The term

\[(x + y) + \text{let } (y \text{ be } 3). (x + y)\]

has binding diagram

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.
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 $[A]$.

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

Paul Blain Levy (University of Birmingham)  λ-calculus, effects and call-by-push-value  April 18, 2017  17 / 129
Semantics of types

Notation

For sets $X$ and $Y$,
- $X \to Y$ 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 $\{ \text{inl } x \mid x \in X \} \cup \{ \text{inr } y \mid y \in Y \}$.

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

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

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

\[ \Gamma \vdash 17 : \text{nat} \]

\[ [17] : \rho \mapsto 17 \]

\[ \Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat} \]

\[ \Gamma \vdash M + M' : \text{nat} \]

\[ [M + M'] : \rho \mapsto [M] \rho + [M'] \rho \]
More semantic equations

\[
\begin{align*}
\Gamma \vdash x : A &\quad (x : A) \in \Gamma \\
[x] : \rho &\mapsto \rho_x \\
\Gamma, x : A \vdash M : B &\quad \Gamma \vdash \lambda x_A. M : A \to B \\
[\lambda x_A. M] : \rho &\mapsto \lambda a \in [A]. [M](\rho, x \mapsto a)
\end{align*}
\]
More semantic equations

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

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

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

$$\llbracket \Gamma' \vdash M : A \rrbracket \rho = \llbracket \Gamma \vdash M \rrbracket (\rho \restriction \Gamma)$$
We can give denotational semantics of binding diagrams.

\[ \text{We } \alpha\text{-equivalent terms have the same denotation.} \]
Binding Diagrams

- We can give denotational semantics of binding diagrams.
- \[ \llbracket \Gamma \vdash M : A \rrbracket = \llbracket \Gamma \vdash \text{BD}(M) : A \rrbracket \]
- 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 \( \llbracket N[M/x, M'/y] \rrbracket \) from \( \llbracket M \rrbracket \) and \( \llbracket N \rrbracket \).

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

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

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

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

$$\begin{align*}
\Gamma &\vdash M : A & \Gamma, x : A &\vdash N : B \\
\Gamma &\vdash (\lambda x_A. N) M = N[M/x] : B
\end{align*}$$

Introduction inside an elimination may be removed.

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

$$\begin{align*}
\Gamma &\vdash M : A & \Gamma &\vdash N : A' \\
\Gamma &\vdash \lambda\{^1. M, ^r. N\}^1 = M : A
\end{align*}$$

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

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

\[
\Gamma \vdash N : C \quad \Gamma \vdash N' : C
\]
\[
\Gamma \vdash \text{match true as } \{ \text{true.} N, \text{false.} N' \} = N : C
\]
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 $A + B$

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

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

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

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

\[
\begin{array}{c}
\frac{
\Gamma \vdash M : A \rightarrow B
}{
\Gamma \vdash M = \lambda x_{A}. M \, x : A \rightarrow B
}\end{array}
\quad x \notin \Gamma
\]

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

**\(\eta\)-law for** \(A \rightarrow B\), **everything is** \(\lambda\)

\[
\begin{align*}
\Gamma \vdash M : A \rightarrow B & \\
\Gamma \vdash M = \lambda x_A. M \, x : A \rightarrow B & \quad x \notin \Gamma
\end{align*}
\]

**Introduction outside an elimination may be inserted.**

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

\[
\begin{align*}
\Gamma \vdash M : A \times B & \\
\Gamma \vdash M = \lambda \{^l \cdot M^l, \ ^r \cdot M^r\} : A \times B
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash M : 1 & \\
\Gamma \vdash M = \lambda \{\} : 1
\end{align*}
\]
More $\eta$-laws

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

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

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

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

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

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

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

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

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

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

$\eta$-law for 0, nothing exists

\[ \Gamma \vdash M : 0 \quad \Gamma, z : 0 \vdash N : C \]
\[ \Gamma \vdash N[M/z] = \text{match } M \text{ as } \{ \} : C \]
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:

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

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

**Closure Theorems**

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

  \[
  \begin{align*}
  z:0 & \vdash \text{true} \equiv_{\beta\eta} \text{false}: \text{bool} \\
  & \vdash \text{true} \ntriangleright_{\beta\eta} \text{false}: \text{bool}
  \end{align*}
  \]

- $\equiv_{\beta\eta}$ 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

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

natural in $\Gamma$—we’ll skate over naturality.
The connective $\to$ is rightist: it has a reversible rule

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

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 $=_\beta\eta$. 
Reversible rule for $A \rightarrow B$

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

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

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 $\Gamma \vdash N : A \rightarrow B$ 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 \( \text{bool} \) 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 \{true.\( M \), 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 \( =_{\beta\eta} \).

\( \text{bool} \) appears on the **left** of \( \vdash \) in the conclusion.
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 $C$. 
Reversible rule for $A + B$

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

$$
\begin{array}{c}
\Gamma, x : A \vdash C \\
\Gamma, y : B \vdash C
\end{array}
\quad\Rightarrow\quad
\begin{array}{c}
\Gamma, z : A + B \vdash C
\end{array}
$$

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.

Likewise $1$ is bipartisan.
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.
Bipartisan connectives

The connective $\times$ has a reversible rule

$$
\Gamma \vdash A \quad \Gamma \vdash B \\
\hline
\Gamma \vdash A \times B
$$

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

It also has a reversible rule

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

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

Likewise 1 is bipartisan.
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.

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

$\Gamma, \sum \{^0 A, A'; ^1 B, B', B''\} \vdash C$

Here is its term syntax:

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

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

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.

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

$$(((\llbracket A \rrbracket \times \llbracket A' \rrbracket) \to \llbracket B \rrbracket) \times (((\llbracket C \rrbracket \times \llbracket C' \rrbracket) \times \llbracket C'' \rrbracket) \to \llbracket D \rrbracket))$$

This gives a rightist connective.

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

$$\Gamma \vdash \prod \{^0 A, A' \vdash B; ^1 C, C', C'' \vdash D\}$$

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

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

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

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 λ-calculus is the most expressive form of simply typed λ-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 $\equiv_{\beta\eta}$ 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 \( \text{let (x be } M, \ y \text{ be } M'). N, \) do we
   - evaluate \( M \) to \( T \) and \( M' \) to \( T' \), then evaluate \( N[T/x, T'/y] \)\
   - just evaluate \( N[M/x, M'/y] \)?
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 \( \text{let} \ (x \ \text{be} \ M, \ y \ \text{be} \ M'). \ N \), 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 \( M \ N \), 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 \texttt{let}

To evaluate \texttt{let} \((x \text{ be } M, \ y \text{ be } M') \cdot N\), do we

- evaluate \(M\) to \(T\) and \(M'\) to \(T'\), then evaluate \(N[T/x, T'/y]\)? \textbf{Call-by-value}
- just evaluate \(N[M/x, M'/y]\)? \textbf{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 | false | inl \( T \) | inr \( T \) | \( \lambda x.M \)

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 \( M \): 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 \( P[T/x] \).
Definitional interpreter for call-by-name

In CBN the terminals are \text{true}, \text{false}, \text{inl} \ M, \text{inr} \ M, \lambda x. M

To evaluate

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

\[
\begin{align*}
M \Downarrow \lambda x_A. P & \quad N \Downarrow T & \quad P[T/x] \Downarrow T' \\
\hline
MN \Downarrow T'
\end{align*}
\]
We write $M \Downarrow T$ to mean that $M$ evaluates to $T$.

This is defined inductively, for example

$$M \Downarrow \lambda x_A. P \quad N \Downarrow T \quad P[T/x] \Downarrow T'$$

$$\frac{}{M N \Downarrow T'}$$

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

\[
\begin{align*}
M \Downarrow \lambda x_A. P & \quad P[N/x] \Downarrow T \\
\hline
M N \Downarrow T
\end{align*}
\]
We write $M \Downarrow T$ to mean that $M$ evaluates to $T$. This is defined inductively, for example

$$
\begin{align*}
M & \Downarrow \lambda x_A. P & P[N/x] & \Downarrow T \\
\frac{}{MN \Downarrow T}
\end{align*}
$$

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

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

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.
Long story

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 $E = \{\text{CRASH}, \text{BANG}\}$ 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

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

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

1. Evaluate

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

in CBV and CBN.

2. Evaluate

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

in CBV and CBN.

3. Evaluate

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

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:

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

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

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

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?

\[\text{\( \eta \)-law for \texttt{bool} } \]

Any term \( \Gamma, z : \texttt{bool} \vdash M : B \) 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,

\[
\text{true } \not\simeq_{\text{CBN}} \text{match } z \text{ as } \{ \text{true. true, false. true} \}
\]

because we can apply the context

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

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 $A \prod B$

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

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

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

<table>
<thead>
<tr>
<th>Term</th>
<th>CBV</th>
<th>CBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>error $e$</td>
<td>$\simeq_{\text{CBV}}$</td>
<td>$\lambda x. \text{error } e$</td>
</tr>
<tr>
<td>error $e$</td>
<td>$\simeq_{\text{CBV}}$</td>
<td>$\lambda {^l. \text{error } e, ^r. \text{error } e }$</td>
</tr>
<tr>
<td>print $c. \lambda x. M$</td>
<td>$\simeq_{\text{CBV}}$</td>
<td>$\lambda x. \text{print } c. M$</td>
</tr>
<tr>
<td>print $c. \lambda {^l. M, ^r. N }$</td>
<td>$\simeq_{\text{CBV}}$</td>
<td>$\lambda {^l. \text{print } c. M, ^r. \text{print } c. N }$</td>
</tr>
</tbody>
</table>

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 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).

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.

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

Our first attempt.

Each type \( A \) denotes a set, a **semantic domain for terms**.

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

Not easy to make this compositional, so we abandon it.
CBV denotational semantics

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

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

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

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

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

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

\[\Gamma \vdash M : A\]

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

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

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

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

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 \([N[M/x]]\) from \([M]\) and \([N]\)? Not in CBV.

Example that rules out a general substitution lemma

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

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

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

$[N] = [N']$ 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.

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

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

match true as {true.false, false.true}
Denotational semantics of values

Each value $\Gamma \vdash V : A$ denotes a function $[V]^\text{val} : [\Gamma] \to [A]$.

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

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 \( \Gamma \vdash^v W : B \) 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 \([M]\).

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

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

- If \( M \Downarrow V \) then \([M] \varepsilon = \text{inl} (\[V\]^{\text{val}} \varepsilon)\).
- If \( M \not\Downarrow e \) then \([M] \varepsilon = \text{inr} e\).

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 $[V] : [\Gamma] \rightarrow [A]$.


Key typing rules

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

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

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

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

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

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

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

\[ \llbracket M \text{ to } x. N \rrbracket : \rho \mapsto \text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl } a. & \llbracket N \rrbracket (\rho, x \mapsto a) \\ \text{inr } e. & \text{inr } e \end{cases} \]
For connectives bool, +, → the syntax is as follows.

\[
V ::= \begin{align*}
x & | \text{true} | \text{false} \\
    & | \text{inl} \ V | \text{inr} \ V | \lambda x. \ M
\end{align*}
\]

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

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

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 $M[V/x]$.
- **match inl $V$ as {inl $x$. $N$, inr $x$. $N'$}**: evaluate $N[V/x]$. 

Paul Blain Levy (University of Birmingham)  λ-calculus, effects and call-by-push-value  April 18, 2017  73 / 129
Equational theory

\(\beta\)-laws

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

\(\eta\)-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

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

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

- $x \mapsto \text{return } x$
- $\lambda x. M \mapsto \text{return } \lambda x. \hat{M}$
- $\text{inl } M \mapsto \hat{M} \text{ to } x. \text{return } \text{inl } x$
- $MN \mapsto \hat{M} \text{ to } x. \hat{N} \text{ to } y. x y$
- let (x be $M$, y be $M'$). $N \mapsto \hat{M} \text{ to } x. \hat{M}' \text{ to } y. \hat{N}$

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

- $x \mapsto x$
- $\lambda x. M \mapsto \lambda x. \hat{M}$
- $\text{inl } V \mapsto \text{inl } \check{V}$
Nullary functions

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

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

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

\[
T A \overset{\text{def}}{=} () \to A \quad [T A] = [A] + E
\]

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

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

The type \( T A \) has a reversible rule:

\[
\begin{align*}
\Gamma \vdash^c A \\
\Gamma \vdash^v T A
\end{align*}
\]
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:

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

Fine-grain CBV (unlike the monadic metalanguage) distinguishes computations from thunks.
Each type denotes a set, a semantic domain for terms. For example:

\[
\begin{align*}
\llbracket \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \rrbracket \ast &= (\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E)) \\
\llbracket \text{bool} + \text{bool} \rrbracket \ast &= ((\mathbb{B} + E) + (\mathbb{B} + E)) + E \\
\llbracket \text{bool} \Pi \text{bool} \rrbracket \ast &= (\mathbb{B} + E) \times (\mathbb{B} + E)
\end{align*}
\]

Thus we define

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

Each term \( \Gamma \vdash M : B \) should denote a function \( \llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow \llbracket B \rrbracket \ast \).
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 \((B + E) \rightarrow ((B + E) \rightarrow (B + E))\)

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 \( \text{match} \).
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 \( \text{error CRASH} \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error CRASH} \)
- 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 \texttt{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*. 

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

$$[[\text{bool} \to (\text{bool} \to \text{bool})]] = ((\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E))),$$
$$\lambda x.\lambda y.\text{inr CRASH},$$
$$\lambda x.\lambda y.\text{inr BANG})$$

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

$$[[\text{bool} \pi \text{bool}]] = ((\mathbb{B} + E) \times (\mathbb{B} + E),$$
$$(\text{inr CRASH}, \text{inr CRASH}),$$
$$(\text{inr BANG}, \text{inr BANG}))$$
CBN semantics of errors

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

If \( [A] = (X, c, b) \) and \( [B] = (Y, c', b') \)
then \( [A + B] = ((X + Y) + E, \text{inr CRASH, inr BANG}) \)
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 \( \text{[[A]]} = (X, c, b) \) and \( \text{[[B]]} = (Y, c', b') \)
then \( \text{[[A + B]]} = ((X + Y) + E, \text{inr CRASH}, \text{inr BANG}) \)
and \( \text{[[A \rightarrow B]]} = (X \rightarrow Y, \lambda x. c', \lambda x. b') \)
and \( \text{[[A \Pi B]]} = (X \times Y, (c, c'), (b, b')) \)

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

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

\[ \Gamma \vdash \text{true} : \text{bool} \]
\[ \llbracket \text{true} \rrbracket : \rho \mapsto \text{inl true} \]

\[ \Gamma \vdash M : \text{bool} \quad \Gamma \vdash N : B \quad \Gamma \vdash N' : B \]
\[ \Gamma \vdash \text{match } M \text{ as } \{ \text{true. } N, \text{ false. } N' \} : B \]
\[ \llbracket \text{match } M \text{ as } \{ \text{true. } N, \text{ false. } N' \} \rrbracket : \rho \]
\[ \text{match } \llbracket M \rrbracket_\rho \text{ as } \begin{cases} \text{inl true.} & \llbracket N \rrbracket_\rho \\ \text{inl false.} & \llbracket N' \rrbracket_\rho \\ \text{inr CRASH.} & c \\ \text{inr BANG.} & b \end{cases} \]

where \( \llbracket B \rrbracket = (Y, c, b) \)
More term constructors

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

Soundness/adequacy

- If \( M \downarrow T \) then \([M]_\varepsilon = [T]_\varepsilon\).
- If \( M \not\downarrow \text{CRASH} \) then \([M]_\varepsilon = c\).
- If \( M \not\downarrow \text{BANG} \) then \([M]_\varepsilon = b\).

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

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

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

- 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.
  \[ X \rightarrow (Y, c, b) \overset{\text{def}}{=} \prod_{x \in X} (Y, c, b) = (X \rightarrow Y, \lambda x. c, \lambda x. b) \]

- 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.

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

A typing context denotes a set.

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

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

A type denotes a set.

\[
\begin{align*}
[\text{bool}] &= 1 + 1 \\
[A + B] &= [A] + [B] \\
[A \rightarrow B] &= UE([A] \rightarrow FE[B]) \\
[TB] &= UEFE[B]
\end{align*}
\]

A typing context denotes a set.

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

A computation $\Gamma \vdash^c M : B$ denotes a function $[\Gamma] \rightarrow FE[B]$. 
Call-By-Push-Value Types

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

**computation type**  
\[ B ::= F A \mid A \rightarrow B \mid 1_\Pi \mid B \Pi B \mid \Pi_{i \in \mathbb{N}} B_i \]
Two kinds of type:

- A **value type** denotes a set.
- A **computation type** denotes an \( E \)-pointed set.

**value type**

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

**computation type**

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

**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 $A$.

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

\[
\Gamma \vdash^c \mathit{return} V : FA
\]

\[
\Gamma \vdash^c M \mathit{to} x. N : B
\]

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

A computation in \( FA \) aims to return a value in \( A \).

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

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

\[
\begin{align*}
\llbracket \text{return } V \rrbracket & : \rho \mapsto \text{inl } \llbracket V \rrbracket \rho \\
\llbracket M \text{ to } x. \ N \rrbracket & : \rho \mapsto \\
\begin{cases} 
\text{match } \llbracket M \rrbracket \rho \text{ as } & \\
\text{inl } a. & \llbracket N \rrbracket (\rho, x \mapsto a) \\
\text{inr CRASH. } & c \\
\text{inr BANG. } & b \\
\end{cases}
\end{align*}
\]

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

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

$$\Gamma \vdash^c M : B \quad \Gamma \vdash^v V : UB$$
$$\Gamma \vdash^v \text{thunk } M : UB \quad \Gamma \vdash^c \text{force } V : B$$
A value in $UB$ is a thunk of a computation in $B$.

$$
\Gamma \vdash^c M : B \\
\Rightarrow \\
\Gamma \vdash^v \text{thunk } M : UB
$$

$$
\Gamma \vdash^c V : UB \\
\Rightarrow \\
\Gamma \vdash^c \text{force } V : B
$$

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

$$
[\text{force } V] = [V]
$$
Identifiers

An identifier is a value.

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

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

\[
\Gamma \vdash^c \text{let } (x \text{ be } V, y \text{ be } W). \ M : C
\]
The rules for 1 are similar.
Functions

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

\[ \frac{\Gamma \vdash^c M : A \rightarrow B \quad \Gamma \vdash^c V : A}{\Gamma \vdash^c MV : B} \]

\[ \frac{\Gamma \vdash^c M_i : B_i \quad (\forall i \in I)}{\Gamma \vdash^c \lambda \{^i. M_i\}_i \in I : \prod_{i \in I} B_i} \]

\[ \frac{\Gamma \vdash^c M : \prod_{i \in I} B_i}{\Gamma \vdash^c M^\hat{i} : B^\hat{i} \quad \hat{i} \in I} \]
Functions

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

\[
\frac{\Gamma \vdash^c M : A \rightarrow B \quad \Gamma \vdash^\nu V : A}{\Gamma \vdash^c MV : B}
\]

\[
\frac{\Gamma \vdash^c M_i : B_i \ \ (\forall i \in I)}{\Gamma \vdash^c \lambda \{^i M_i\}_i I : \prod_{i \in I} B_i}
\]

\[
\frac{\Gamma \vdash^c M : \prod_{i \in I} B_i}{\Gamma \vdash^c M^\hat{i} : B^\hat{i}} \quad \hat{i} \in I
\]

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

- `return V`
- `λx. M`
- `λ{^i. M_i}_{i ∈ I}`
Definitional interpreter for call-by-push-value

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

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 \( N[V/x] \).
- **\( \lambda \{^i. N_i\}_{i \in I} \):** return \( \lambda \{^i. N_i\}_{i \in I} \).
- **\( M \hat{\_} \):** evaluate \( M \). If this returns \( \lambda \{^i. N_i\}_{i \in I} \), evaluate \( N \hat{\_} \).
- **let \( (x \text{ be } V, \ y \text{ be } W) \). \( M \):** evaluate \( M[V/x, W/y] \).
- **force thunk \( M \):** evaluate \( M \).
- **match \( \text{in}_i V \) as \( \{\text{in}_i . M_i\}_{i \in I} \):** evaluate \( M \hat{\_}[V/x] \).
- **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 } (\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

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

Sequencing laws

- \((\text{return } V) \text{ to } x. \ M = M[V/x]\)
- \(M = M \text{ to } x. \text{ return } x\)
- \((M \text{ to } x. N) \text{ to } y. \ P = M \text{ to } x. (N \text{ to } y. \ P)\)
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.

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

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.

\[ \text{translation} \quad A \rightarrow B \quad \mapsto \quad U(A \rightarrow FB) \]

\[ TB \quad \mapsto \quad UFB \]

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

\[ \lambda x. M \quad \mapsto \quad \text{thunk} \lambda x. M \]

\[ VW \quad \mapsto \quad (\text{force } V)W \]
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 \( x : A, y : B \vdash^c M : FC \).

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

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

\[ x \mapsto \text{return} x \]
\[ \lambda x. M \mapsto \text{return thunk} \lambda x. M \]
\[ M N \mapsto M \text{ to } f. N \text{ to } y. ((\text{force } f) y) \]
A CBN type translates into a computation type.

\[
\begin{align*}
\text{bool} & \mapsto F(1 + 1) \\
A + B & \mapsto F(UA + UB) \\
A \to B & \mapsto UA \to B
\end{align*}
\]
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 \(x : UA, y : UB \vdash^c M : C\).

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

- the call-by-push-value calculus
- its operational 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 $TA$ 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 $\texttt{thunk}$ and $\texttt{force}$ 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.
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 $N[V/x]$.

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

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

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

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

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

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

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

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

Revisiting some equations:

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

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

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

\[
\text{print} c. \quad \lambda x. M = \lambda x. \text{print} c. M
\]
Those function rules again

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

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

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

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

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

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

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

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

Revisiting some equations:

\[
V \cdot \lambda x. M = M[V/x] \\
M = \lambda x. x \cdot 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 $\sum_{i \in I}A_i$ is a pair $\langle i, V \rangle$.
- A value of type $A \times A'$ is a pair $\langle V, V' \rangle$.

- A computation of type $FA$ aims to return a value of type $A$.
- A computation of type $A \rightarrow B$ aims to pop a value of type $A$ then behave in $B$.
- A computation of type $\prod_{i \in I}B_i$ aims to pop a tag $i \in I$ then behave in $B_i$. 
What’s in a stack?

A stack consists of

- arguments that are values
- arguments that are tags
- frames taking the form \texttt{to x. N.}
Example program of type $\mathcal{F} \mathfrak{n} \mathfrak{a} \mathfrak{t}$ (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 $\Gamma \vdash^c P : C$

\[
\begin{array}{cccc}
\Gamma & P & C & \text{nil} & C \\
\end{array}
\]

Transitions

\[
\begin{array}{cccc}
\Gamma & M \to x. \, N & B & K & C \\
\rightarrow & \Gamma & M \quad FA & \text{to x.} & N :: K & C \\
\Gamma & \text{return} \, V & FA & \text{to x.} & N :: K & C \\
\rightarrow & \Gamma & N[V/x] & B & K & C \\
\end{array}
\]

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

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

$\begin{array}{ccccc}
\Gamma & P & C & \text{nil} & C
\end{array}$

Transitions

$\begin{array}{cccccccc}
\Gamma & M \text{ to } x. N & B & K & C & \rightsquigarrow \\
\Gamma & M & FA & \text{ to } x. N :: K & C
\end{array}$

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

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

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

Typing a stack

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

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

\[ \Gamma \vdash^k \hat{i} :: K : \prod_{i \in I} B_i \rightarrow C \]

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

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

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

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

Typing a stack

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

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

\[ \vdash^k \hat{i} :: K : \prod_{i \in I} B_i \rightarrow C \hat{i} \in I \]

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

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

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.
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 $\Gamma \vdash^k K : B \implies C$, we can weaken it or substitute values.

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

3. Stacks $\Gamma \vdash^k K : B \implies C$ and $\Gamma \vdash^k L : C \implies D$ can be concatenated to give $\Gamma \vdash^k K + L : B \implies D$. 
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.
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.
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 $C$. 
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 $\Gamma \vdash^k \text{nil} : C \longrightarrow C$.

The top-level type is $C$.

If $C$ is an $F$ type, then $\text{nil}$ is the top-level continuation: it receives a value and returns it to the user.
Stacks denote homomorphisms

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

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

What should $K$ denote?
Stacks denote homomorphisms

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

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 \( [K] : [\Gamma] \times X \rightarrow Y \).
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 $\Gamma \vdash^k K : B \longrightarrow C$

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 $\llbracket K \rrbracket : \llbracket \Gamma \rrbracket \times X \longrightarrow Y$.

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 $M \cdot K$.

We assume there’s no exception handling.
We define $\llbracket K \rrbracket$ 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.
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?

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

Properties:

1. If $(M, K) \rightsquigarrow (M', K')$ 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).

\[
\begin{array}{c}
\text{Set} \\ \downarrow \\
U^E \\
\rightarrow \\
\rightarrow \\
\text{E/Set}
\end{array}
\quad \xrightarrow{F^E}
\]

This resolves the exception monad $X \mapsto X + E$ on Set.
State

Consider CBPV extended with two storage cells: 1 stores a natural number, and 1’ stores a boolean.
Consider CBPV extended with two storage cells: 
\( \mathbf{l} \) stores a natural number, and \( \mathbf{l}' \) stores a boolean.

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

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

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

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

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

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

We can type these using

$$
\Gamma \vdash^c M : B \\
\Gamma \vdash^{sc} (s, M) : B \\
s \in S
$$
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 $\lbrack A \rbrack$, a semantic domain for values.

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

We complete the story with an adequacy theorem:

If $s, M \Downarrow s', T$ then $\lbrack s, M \rbrack\varepsilon = \lbrack s', T \rbrack\varepsilon$

This requires an SC-configuration to have a denotation.
A value type $A$ denotes a set $[[A]]$, a semantic domain for values.

A computation type $B$ denotes a set $[[B]]$, a semantic domain for SC-configurations.
A value type $A$ denotes a set $[[A]]$, a semantic domain for values.

A computation type $B$ denotes a set $[[B]]$, a semantic domain for SC-configurations.

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

$$[[s, M]] : [[\Gamma]] \rightarrow [[B]]$$
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:

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

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 $A \to B$ will pop $x : A$, then behave in $B$.

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

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

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

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

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

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

What should $K$ denote?

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

So we want $[K] : [\Gamma] \times [B] \to [C]$. 
State: the value/stack adjunction

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

What should \( K \) denote?

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

So we want \( \llbracket K \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket B \rrbracket \longrightarrow \llbracket C \rrbracket \).

This gives an adjunction

\[
\begin{array}{c}
\text{Set} \\
\downarrow \quad S \times - \\
S \rightarrow - \\
\end{array}
\left \downarrow \right \rightarrow
\begin{array}{c}
\text{Set} \end{array}
\]

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

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

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 $\alpha$. 

Similar to Crolard's syntax. Numerous variations in the literature.
Naming and changing the current stack

Extend the language with two instructions:

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

Execution takes places in a bigger language.

\[
\begin{array}{llll}
\Gamma & \texttt{letstk }\alpha. M & B & K & C | \Delta \\
\Gamma & M[K/\alpha] & B & K & C | \Delta \\
\end{array}
\]

\[
\begin{array}{llll}
\Gamma & \texttt{changestk }K. M & B' & L & C | \Delta \\
\Gamma & M & B & K & C | \Delta \\
\end{array}
\]

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

We have typing judgements:

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

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

We have typing judgements:

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

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

Example typing rules

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

\[
\Gamma \vdash \Gamma \vdash M : B \mid \Delta \\
\quad \Gamma \vdash \text{changestk} \alpha. M : B' \mid \Delta \quad (\alpha : B) \in \Delta
\]

Paul Blain Levy (University of Birmingham)  
\( \lambda \)-calculus, effects and call-by-push-value  
April 18, 2017  
122 / 129
During execution, the top-level type $C$ must be indicated:

\[
\begin{align*}
\Gamma \vdash_V V : A & \ [C] \ \Delta \\
\Gamma \vdash_c M : B & \ [C] \ \Delta \\
\Gamma \vdash_k K : B \implies & \ C \ | \ \Delta \\
\Gamma \vdash_{ck} (M, K) : & \ C \ | \ \Delta
\end{align*}
\]

Typically $\Gamma$ and $\Delta$ would be empty and $C = F\ \text{bool}$. 
During execution, the top-level type $C$ must be indicated:

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

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

**Example typing rules**

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

That means: a hypothetical extremely closed CK-configuration, with no free identifiers and no \texttt{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 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 $\llbracket B \rrbracket_{\text{alg}}$,
a semantic domain for computations.
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 $\llbracket B \rrbracket$. 
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 $\llbracket B \rrbracket$.

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

$$\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket B \rrbracket \times \llbracket \Delta \rrbracket \to R$$

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

$$\llbracket V \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \to \llbracket A \rrbracket$$
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 $\prod_{i \in I} B_i$ is a pair $\hat{i} :: K$.

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

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

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

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

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

That gives an adjunction

$$\text{Set} \xleftarrow{\rightarrow} \text{Set}^{\text{op}}$$

between values and stacks.
Control in call-by-value and call-by-name

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

For call-by-value we recover
\[
\begin{align*}
\text{bool}_{\text{CBV}} & = 1 + 1 \\
A \rightarrow_{\text{CBV}} B & = U_{A \rightarrow_{\text{FB}} B} \\
\end{align*}
\]
This is standard.

For call-by-name we recover
\[
\begin{align*}
\text{bool}_{\text{CBN}} & = F(1 + 1) \\
A \rightarrow_{\text{CBN}} B & = U_{A \rightarrow B} \\
\end{align*}
\]
This is Streicher and Reus' semantics for a CBN language with control operators.
Abbreviate $\neg X \overset{\text{def}}{=} X \rightarrow R$.

For call-by-value we recover

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

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

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

This is standard.

For call-by-name we recover

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

This is Streicher and Reus’ semantics for a CBN language with control operators.
For a set $E$, the adjunction $\text{Set} \xleftarrow{\bot} E/\text{Set}$ models call-by-push-value with errors.
For a set $E$, the adjunction $\text{Set} \xrightarrow{\perp} E/\text{Set}$ models call-by-push-value with errors.

For a set $S$, the adjunction $\text{Set} \xrightarrow{\perp} \text{Set}$ models call-by-push-value with state.
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.
Summary: adjunctions between values and stacks

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

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

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