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

Paul Blain Levy

University of Birmingham

April 2, 2023
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
We’re going to look at simply typed \( \lambda \)-calculus with arithmetic, including not just function types, but also sum and product types.

Here is the syntax of types:

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

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

Here is the syntax of types:

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

Why no brackets?

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

Example

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

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 list 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.

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

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

Two introduction rules

$$\Gamma \vdash M : A \quad \Gamma \vdash inl^{A,B} M : A + B$$

$$\Gamma \vdash M : B \quad \Gamma \vdash inr^{A,B} M : A + B$$

Elimination rule

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

$$\Gamma \vdash \text{match } M \text{ as } \{ \text{inl } x. \ N, \ \text{inr } y. \ N' \} : C$$
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

\[
\Gamma \vdash M : 0 \\
\Gamma \vdash \text{match } M \text{ as } \{\}^A : A
\]
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

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

  $$\Gamma \vdash M^1 : A$$

  $$\Gamma \vdash M^r : B$$
Typing rules for $A \times B$

Introduction rule

\[
\begin{align*}
\Gamma \vdash M : A & \quad \Gamma \vdash N : B \\
\hline
\Gamma \vdash \langle M, N \rangle : A \times B
\end{align*}
\]

Two options for elimination

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

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

- **Projection product.** Two elimination rules

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

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

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

Introduction rule

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

Two options for elimination

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

  \[
  \Gamma \vdash M : 1 \quad \Gamma \vdash N : C
  \]

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

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

**Theorem**

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

The term \((x + y) + \text{let} (y \text{ be } 3). (x + y)\) has binding diagram

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

- Terms are \(\alpha\)-equivalent when they have the same binding diagram.

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

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

Example

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

\[
\begin{align*}
M &= \lambda y_{\text{nat}}. y + 3 \\
M' &= 7 \\
N &= x(5 + y) \\
N[M/x, M'/y] &= (\lambda z_{\text{nat}}. z + 3)(5 + 7)
\end{align*}
\]
Types denote sets

- Every type $A$ denotes a set $[A]$.
- For example, $[\text{nat} \to \text{nat}]$ is the set of functions $\mathbb{N} \to \mathbb{N}$. 
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 $\llbracket A \rrbracket$. 
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 $[A]$.

- 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 \( \{\text{inl } x \mid x \in X\} \cup \{\text{inr } y \mid y \in Y\} \).

\[
\begin{align*}
\llbracket \text{bool} \rrbracket & = \mathbb{B} = \{\text{true, false}\} \\
\llbracket \text{nat} \rrbracket & = \mathbb{N} \\
\llbracket A \rightarrow B \rrbracket & = \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \\
\llbracket 1 \rrbracket & = 1 = \{\langle \rangle \} \\
\llbracket A + B \rrbracket & = \llbracket A \rrbracket + \llbracket B \rrbracket \\
\llbracket A \times B \rrbracket & = \llbracket A \rrbracket \times \llbracket B \rrbracket \\
\llbracket 0 \rrbracket & = \emptyset
\end{align*}
\]
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 $[M]$, or more precisely $[[\Gamma \vdash M : A]]$. It’s a function from $[[\Gamma]]$ to $[[A]]$.

Example

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

denotes the function

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

$$\rho \mapsto \lambda z \in \mathbb{N} \rightarrow \mathbb{N}. z(\rho_x + \rho_y)$$
\[ \Gamma \vdash 17 : \text{nat} \]
\[ [17] : \rho \mapsto 17 \]
\[ \begin{array}{c}
\Gamma \vdash M : \text{nat} \\
\Gamma \vdash M' : \text{nat}
\end{array} \]
\[ \Gamma \vdash M + M' : \text{nat} \]
\[ [M + M'] : \rho \mapsto [M] \rho + [M'] \rho \]
More semantic equations

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

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

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

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

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

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

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

Semantic Coherence

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

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

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

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

Weakening Lemma

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

$$[\Gamma' \vdash M : A] \rho = [\Gamma \vdash M](\rho \upharpoonright \Gamma)$$
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 \(\alpha\)-equivalent terms have the same denotation.

**Substitution Lemma**

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

\[[N[M/x, M'/y]] : \rho \mapsto [N](\rho, x \mapsto [M]\rho, y \mapsto [M']\rho)\]
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 $A \rightarrow B$

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

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

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
Two $\beta$-laws for bool

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

\[
\Gamma \vdash \text{match true as } \{ \text{true. } N, \text{ false. } N' \} = N : C
\]
More $\beta$-laws

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

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

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

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

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

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

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

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

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

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

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

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

Introduction outside an elimination may be inserted.
η-laws

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

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

Introduction outside an elimination may be inserted.

η-law for projection product $A \times B$, everything is a tuple

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

η-law for projection unit $1$, everything is a tuple

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

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

\[
\begin{align*}
\Gamma \vdash M : \text{bool} & \quad \Gamma, z : \text{bool} \vdash N : C \\
\Gamma \vdash N[M/z] = & \quad z \not\in \Gamma \\
\text{match } M \text{ as } \{ \text{true. } N[\text{true}/z], \text{false. } N[\text{false}/z] \} & : C
\end{align*}
\]
η-law for bool, everything is true or false

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

η-law for \( A + B \), everything is inl or inr

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

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

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

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

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

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

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

$\eta$-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:

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

$$\Gamma \vdash \lambda x_A. M = \lambda x_A. M' : A \to B$$
Closure Theorems

- $\beta\eta$ is closed under weakening. But not conversely, e.g.
  \[
  z : 0 \vdash \text{true} =_{\beta\eta} \text{false} : \text{bool}
  \]
  but not \[
  \vdash \text{true} =_{\beta\eta} \text{false} : \text{bool}
  \]

- $\beta\eta$ is closed under substitution.

Soundness theorem

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

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

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

$$\frac{}{\Gamma \vdash A \rightarrow B}$$

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

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

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

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 $=_{\beta \eta}$. 
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 $=_{\beta\eta}$.

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

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

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 \(\Gamma, z : \texttt{bool} \vdash N : C\) is sent to \(N[\text{true}/z]\) and \(N[\text{false}/z]\).
- These are inverse up to \(=_{\beta\eta}\).

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

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

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

$$\Gamma, z : A + B \vdash C$$

natural in $\Gamma$ and $C$. 
The connective $+$ is leftist, having a reversible rule

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

\quad \frac{}{\Gamma, z : A + B \vdash C}
$$

natural in $\Gamma$ and $C$.

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

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

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

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

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

It also has a reversible rule

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

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

The connective $\times$ has a reversible rule

$$\Gamma \vdash A \quad \Gamma \vdash B$$

$$\Gamma \vdash A \times B$$

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

It also has a reversible rule

$$\Gamma, x : A, y : B \vdash C$$

$$\Gamma, z : A \times B \vdash C$$

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

The connective $\times$ has a reversible rule

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

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

It also has a reversible rule

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

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.

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

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

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

Here is its term syntax:

\[
\begin{align*}
\text{in}_0(M, M') \\
\text{in}_1(M, M', M'') \\
\text{match } M \text{ as } \{ \text{in}_0(x, x'). N, \text{in}_1(y, y', y''). N' \}
\end{align*}
\]
Most general rightist connective

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

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

This gives a rightist connective.

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

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

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

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

This gives a rightist connective.

$$
\begin{array}{c}
\Gamma, A, A' \vdash B \\
\Gamma, C, C', C'' \vdash D
\end{array}
\quad \Gamma \vdash \prod \{^0 A, A' \vdash B; ^1 C, C', C'' \vdash D\}
$$

Here is its term syntax:

$$\lambda \{^0 (x, x').M, ^1 (y, y', y'').M'\}
\quad M^0(N, N')
\quad M^1(N, N', N'')$$
Jumbo \( \lambda \)-calculus

Type syntax

\[
A ::= \sum_{i<n} \{ \overrightarrow{A_i} \} \quad | \quad \prod_{i<n} \{ \overrightarrow{A_i} \vdash B_i \} \quad (n \in \mathbb{N} \text{ or } n = \infty)
\]

Term syntax, with type annotations omitted

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

Type syntax

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

Term syntax, with type annotations omitted

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

Includes both pattern-match product $A \times B$ and projection product $A \sqcap B$. 
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 \( N[M/x, M'/y] \)?
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 \( P[N/x] \)?
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?
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.
To evaluate `let (x be M, y be M'). N`, do we

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

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

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

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

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

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 $\text{inl } T$.
- **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 $\text{inl } T$, 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 $M$, $\lambda x.M$

To evaluate

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

This is defined inductively, for example

$$
\frac{M \Downarrow \lambda x_A. P \quad N \Downarrow T \quad P[T/x] \Downarrow T'}{
M \ N \Downarrow T'}
$$
We write $M \downarrow T$ to mean that $M$ evaluates to $T$. This is defined inductively, for example

\[
M \downarrow \lambda x_A. P \quad N \downarrow T \quad P[T/x] \downarrow T' \\
\hline
MN \downarrow T'
\]

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

Moreover $\vdash T : A$ and $[M] = [T]$. 
Big-step semantics for call-by-name

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

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

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

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

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

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

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

Printing

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

$$\Gamma \vdash M : B \quad c \in A$$

$$\Gamma \vdash \text{print} \; c \; . \; M : B$$

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

1. Evaluate

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

in CBV and CBN.

2. Evaluate

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

in CBV and CBN.

3. Evaluate

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

in CBV and CBN.
Big-step semantics for errors

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

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

Here are the rules for application:

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

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?

**\( \eta \)-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 \text{true} or \text{false}.

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

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

because we can apply the context

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

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

**$\eta$-law for $A \rightarrow B$ and $A \Pi B$**

Any term $\Gamma \vdash M : A \rightarrow B$ 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).
Summary

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.
Our first attempt.

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

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

Our first attempt.

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

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

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

\[ 
\begin{align*}
\Gamma, x : A \vdash M : B \\
\overline{\Gamma \vdash \lambda x \in A. M : A \to B}
\end{align*}
\]

\[ 
\begin{align*}
&\quad \Gamma \vdash \lambda x \in A. M : A \to B \\
&\quad \Gamma \notin \lambda x \in A. [\lambda x A. M] : \rho \mapsto \text{inl} \ \lambda a \in \lbrack A \rbrack. [\lbrack M \rbrack(\rho, x \mapsto a)]
\end{align*}
\]

\[ 
\begin{align*}
\Gamma \vdash M : A \to B & \quad \Gamma \vdash N : A \\
\overline{\Gamma \vdash MN : B}
\end{align*}
\]

\[ 
\begin{align*}
\quad \Gamma \notin \lambda x \in A. [\lambda x A. M] : \rho \mapsto \text{match} \ [\lbrack M \rbrack] \rho \text{ as }& \begin{cases}
\quad \text{inl } f. & \text{match} \ [\lbrack N \rbrack] \rho \text{ as } \begin{cases}
\text{inl } x. & f(x) \\
\text{inr } e. & \text{inr } e
\end{cases} \\
\text{inr } e. & \text{inr } e
\end{cases}
\end{align*}
\]
More term constructors

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

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

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

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

To prove the soundness of the denotational semantics, we need a substitution lemma.
Can we obtain \([N [M/x]]\) from \([M]\) and \([N]\)?

Not in CBV.

Example that rules out a general substitution lemma

Define

\[

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

\([N] = [N']\) because \(N = \eta N'\)

\([N [M/x]] \neq [N' [M/x]]\)
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 \(\vdash x : \text{bool} \vdash N, N' : \text{bool}\).

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

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

$[N[M/x]] \neq [N'[M/x]]$

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

\[
\text{match true as } \{\text{true.false, false.true}\}
\]
Denotational semantics of values

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

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

Likewise for substitution of values into values.
If $M \downarrow V$ then $\llbracket M \rrbracket \varepsilon = \text{inl} (\llbracket V \rrbracket^{\text{val}} \varepsilon)$.

If $M \nmid e$ then $\llbracket M \rrbracket \varepsilon = \text{inr} e$.

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

- A value $\Gamma \vdash^v V : A$ denotes a function $[V] : [\Gamma] \to [A]$.

Key typing rules

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

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

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

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

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

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

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

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

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 $N[V/x]$. 
Equational theory

\( \beta \)-laws

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

\( \eta \)-laws

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

Sequencing laws

\[
\begin{align*}
\text{(return } V) \text{ to } x. \ M &= M[V/x] \\
M &= M \text{ to } x. \text{ return } x \\
(M \text{ to } x. \ N) \text{ to } y. \ P &= M \text{ to } x. (N \text{ to } y. \ P)
\end{align*}
\]
Term $\Gamma \vdash M : A$ to computation $\Gamma \vdash^c \hat{M} : A$.

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

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

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

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

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

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

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

The type \( TA \) has a reversible rule

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

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

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

The type $T A$ has a reversible rule

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

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

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

\[
\begin{align*}
\llbracket \text{bool} \to (\text{bool} \to \text{bool}) \rrbracket_* &= (B + E) \to ((B + E) \to (B + E)) \\
\llbracket \text{bool} + \text{bool} \rrbracket_* &= ((B + E) + (B + E)) + E \\
\llbracket \text{bool} \Pi \text{bool} \rrbracket_* &= (B + E) \times (B + E)
\end{align*}
\]

Thus we define

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

Each term $\Gamma \vdash M : B$ should denote a function $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \to [B]_*$. 
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 \( (B + E) \to ((B + E) \to (B + E)) \)
and \( \text{error CRASH} \simeq \text{CBN} \lambda x. \lambda y. \text{error CRASH} \)

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 \( \text{error CRASH} \sim_{\text{CBN}} \lambda x. \lambda y. \text{error CRASH} \)
- 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.
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 \((\text{bool} + E) \rightarrow ((\text{bool} + E) \rightarrow (\text{bool} + E))\)
- and \( \text{error CRASH} \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error CRASH} \)
- so the answer should be \( \lambda x. \lambda y. \text{inr CRASH} \).

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

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

$$\lambda x.\lambda y.\text{inr CRASH},$$

$$\lambda x.\lambda y.\text{inr BANG})$$

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

$$\text{inr CRASH},$$

$$\text{inr BANG})$$

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

$$(\text{inr CRASH, inr CRASH}),$$

$$(\text{inr BANG, inr BANG}))$$
CBN semantics of errors

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

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

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

\([A] = (X, c, b)\)

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

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

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

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

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

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

\[ \text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl true. } & \llbracket N \rrbracket \rho \\ \text{inl false. } & \llbracket N' \rrbracket \rho \\ \text{inr CRASH. } & c \\ \text{inr BANG. } & b \end{cases} \]

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 $X$.**
  
  
  $$F^E X \overset{\text{def}}{=} (X + E, \text{inr CRASH}, \text{inr BANG})$$

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

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

- **Product of a family of $E$-pointed sets.**
  
  $$\prod_{i \in I} (X_i, c_i, b_i) \overset{\text{def}}{=} (\prod_{i \in I} X_i, \lambda i. c_i, \lambda i. b_i)$$

- **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.

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

A typing context denotes a set.

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

A term $\Gamma \vdash M : B$ denotes a function $[\Gamma] \longrightarrow [B]$. 
A type denotes a set.

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

A typing context denotes a set.

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

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

- A value type denotes a set.
- A computation type denotes an \( E \)-pointed set.
Call-By-Push-Value Types

Two kinds of type:

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

value type

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

computation type

\[
B ::= \quad FA \mid A \to B \mid 1_{\Pi} \mid B \Pi B \mid \Pi_{i \in \mathbb{N}} B_i
\]
Two kinds of type:

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

**value type**

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

**computation type**

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

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

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

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

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

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

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 $A$.

\[
\frac{\Gamma \triangleright^\nu V : A}{\Gamma \vdash^c \text{return} V : FA}\]

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

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

The type $FA$

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

$$
\Gamma \vdash^v V : A \\
\Gamma \vdash^c \text{return } V : FA
$$

$$
\Gamma \vdash^c M : FA \\
\Gamma, x : A \vdash^c N : B \\
\Gamma \vdash^c M \text{ to } x. \ N : B
$$

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{inl } a. & \llbracket N \rrbracket(\rho, x \mapsto a) \\
\text{inr CRASH. } c & \\
\text{inr BANG. } b & 
\end{cases} \\
\text{match } \llbracket M \rrbracket \rho \text{ as } & \\
\text{where } \llbracket B \rrbracket = (Y, c, b)
\end{align*}
The type $UB$

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

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

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

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

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

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

An identifier is a value.

$$\Gamma \vdash^v x : A$$

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

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

\[
\begin{align*}
\Gamma \vdash^\nu V &: A & \quad \Gamma \vdash^\nu V &: A' \\
\Gamma \vdash^\nu \text{inl } V &: A + A' & \quad \Gamma \vdash^\nu \text{inr } V &: A + A'
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash^\nu V &: A & \quad \Gamma \vdash^\nu V' &: A' \\
\Gamma \vdash^\nu \langle V, V' \rangle &: A \times A'
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash^\nu V &: A \times A' & \quad \Gamma, x : A, y : A' \vdash^c M &: B \\
\Gamma \vdash^c \text{match } V \text{ as } \langle x, y \rangle . M &: B
\end{align*}
\]

The rules for 1 are similar.
Functions

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

$$\Gamma \vdash^c \lambda x. M : A \to B$$

$$\Gamma \vdash^c M : A \to B \quad \Gamma \vdash^v V : A$$

$$\Gamma \vdash^c MV : B$$

$$\Gamma \vdash^c M : B \quad \Gamma \vdash^c M' : B'$$

$$\Gamma \vdash^c \lambda \{^1. M, \ r. M'\} : B \sqcap B'$$

$$\Gamma \vdash^c M : B \sqcap B'$$

$$\Gamma \vdash^c M^1 : B$$

$$\Gamma \vdash^c M : B \sqcap B'$$

$$\Gamma \vdash^c M^r : B'$$
Functions

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

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

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

\[ \Gamma \vdash^c MV : B \]

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

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

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

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

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

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

It is often convenient to write applications operand-first, as \( V^1.M \) and \( ^1.M \) and \( ^r.M \).
Definitional interpreter for call-by-push-value

The terminals are **computations**:

- `return V`
- `\lambda x. M`
- `\lambda \{^l M, ^r M' \}`
The terminals are **computations**: \( \text{return } V \quad \lambda x. M \quad \lambda\{^1. M, \ r. M'\} \)

To evaluate

- **return \( V \)**: return \( \text{return } V \).
- **\( M \) to \( x. \ N \)**: evaluate \( M \). If this returns \( \text{return } V \), then evaluate \( N[V/x] \).
- **\( \lambda x. N \)**: return \( \lambda x. N \).
- **\( MV \)**: evaluate \( M \). If this returns \( \lambda x. N \), evaluate \( N[V/x] \).
- **\( \lambda\{^1. M, \ r. M'\} \)**: return \( \lambda\{^1. M, \ r. M'\} \).
- **\( M^1 \)**: evaluate \( M \). If this returns \( \lambda\{^1. N, \ r. N'\} \), evaluate \( N \).
- **let (\( x \) 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

- 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

- \(V \) \(= \) thunk force \(V\)
- \(M[V/z] \) \(= \) match \(V\) 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 CBP

A CBV type translates into a value type.

\[
\begin{align*}
A \rightarrow B & \quad \mapsto \quad U(A \rightarrow FB) \\
TB & \quad \mapsto \quad UFB
\end{align*}
\]
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 \( x : A, y : B \vdash^c M : FC \).
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 \( x : A, y : B \vdash^c M : FC \).

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

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

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.

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

A CBN type translates into a computation type.

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

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

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

We’ve seen
- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.

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

Moggi’s $TA$ is $UFA$. 
We’ve seen
- the call-by-push-value calculus
- its operational semantics
- denotational semantics for errors.

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

Moggi’s $TA$ is $UFA$.

But
We’ve seen

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

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

Moggi’s $TA$ is $UFA$.

But

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

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

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

Moggi’s $TA$ is $UFA$.

**But**

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

An operational semantics due to Felleisen and Friedman (1986). And Landin, Krivine, Streicher and Reus, Bierman, Pitts, …

It is suitable for sequential languages whether CBV, CBN or CBPV.

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

Initially, K is empty.

Some authors make K into a single context, called an “evaluation context”.
To evaluate $M \to x. \ N$: evaluate $M$. If this returns return $V$, then evaluate $N[V/x]$.

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

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

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

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

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

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

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

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

Revisiting some equations:

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

error $e = \lambda x. error e$

print $c$.

$\lambda x. M = \lambda x. print c. M$
Those function rules again

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

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

Paul Blain Levy (University of Birmingham)  \( \lambda \)-calculus, effects and call-by-push-value  
April 2, 2023  100 / 128
Those function rules again

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

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

We can read \textit{V'} as an instruction “push \textit{V}”.

We can read \textit{λx} as an instruction “pop \textit{x}”.

Revisiting some equations:

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

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

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

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

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

- A computation of type $FA$ aims to return a value of type $A$.
- A computation of type $A \to B$ 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 $\text{to } x. \ N$. 

Paul Blain Levy (University of Birmingham)  \(\lambda\)-calculus, effects and call-by-push-value  April 2, 2023 102 / 128
Example program of type $F\text{nat}$ (with complex values)

```ml
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 $\Gamma \vdash^c P : C$

\[\Gamma \quad P \quad C \quad \text{nil} \quad C\]

Transitions

\[\begin{array}{l}
\Gamma \quad M \text{ to } x. \ N \quad B \quad K \quad C \quad \leadsto \\
\Gamma \quad M \quad FA \quad \text{to } x. \ N :: K \quad C
\end{array}\]

\[\begin{array}{l}
\Gamma \quad \text{return } V \quad FA \quad \text{to } x. \ N :: K \quad C \quad \leadsto \\
\Gamma \quad N[V/x] \quad B \quad K \quad C
\end{array}\]

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

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

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

Transitions

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

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

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

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

### Typing a stack

\[
\begin{align*}
\Gamma & \vdash^k \text{nil} : C \rightarrow C \\
\Gamma & \vdash^k K : B \rightarrow C \\
\Gamma & \vdash^k 1 :: K : B \sqcap B' \rightarrow C
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : A & \vdash^c M : B \\
\Gamma & \vdash^k K : B \rightarrow C \\
\Gamma & \vdash^k \text{to } x. \ M :: K : FA \rightarrow C
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash^v V : A \\
\Gamma & \vdash^k K : B \rightarrow C \\
\Gamma & \vdash^k V :: K : A \rightarrow B \rightarrow C
\end{align*}
\]
Typing rules, read off from the CK-machine

### Typing a stack

- $\Gamma \vdash^k \text{nil} : C \Rightarrow C$
- $\Gamma \vdash^k K : B \Rightarrow C$
- $\Gamma \vdash^k \mathbf{1} :: K : B \Pi B' \Rightarrow C$
- $\Gamma, x : A \vdash^c M : B$
- $\Gamma \vdash^k K : B \Rightarrow C$
- $\Gamma \vdash^k \text{to x. } M :: K : FA \Rightarrow C$

### Typing a CK-configuration

- $\Gamma \vdash^c M : B$
- $\Gamma \vdash^k K : B \Rightarrow C$
- $\Gamma \vdash^{ck} (M, K) : C$
- $\Gamma \vdash^v V : A$
- $\Gamma \vdash^k K : B \Rightarrow C$
- $\Gamma \vdash^k V :: K : A \rightarrow B \Rightarrow C$
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 $\Gamma \vdash^c M \bullet K : C$. 
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 $\Gamma \vdash^k K \bowtie L : B \Rightarrow D$. 
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.
Special Stacks

Continuations

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

Top-Level Stack

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

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 $\Gamma \vdash^k K : B \Longrightarrow C$

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

What should $K$ denote?
Stacks denote homomorphisms

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

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

What should $K$ denote?

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

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

Paul Blain Levy (University of Birmingham) 
λ-calculus, effects and call-by-push-value 
April 2, 2023 108 / 128
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 $M \bullet K$. 
Stacks denote homomorphisms

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

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

What should $K$ denote?

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

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

This function should be 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 $\llbracket K \rrbracket$ 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 \( \Gamma \vdash_{\text{ck}} (M, K) : C \) denote?

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

Properties:

1. If \((M, K) \leadsto (M', K')\) then \([ (M, K) ] = [ (M', K') ] \).
2. \([\text{error CRASH}, K]\] \(\rho = c'\).
3. \([\text{error BANG}, K]\] \(\rho = b'\).
Adjunction between values and stacks

We have an adjunction between the category of values (sets and functions) and the category of stacks ($E$-pointed sets and homomorphisms).

$$\text{Set} \quad \xleftarrow{\bot} \quad E/\text{Set} \quad \xrightarrow{F^E}$$

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

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

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

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

$$
\Gamma \vdash^c M : B
$$

$$
\frac{}{\Gamma \vdash^{sc} (s, M) : B}
$$

$s \in S$
How can we give a denotational semantics for call-by-push-value with state?

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

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 $[A]$, a semantic domain for values.

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

We complete the story with an adequacy theorem:

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

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:

$$[[ (s, M) ]] : [[\Gamma ]] \rightarrow [[B]]$$
Intrinsic semantics of state

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:

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

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

$$[[M]] : S \times [[\Gamma]] \rightarrow [[B]]$$
An SC-configuration of type $FA$ will terminate as $s$, return $V$.

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

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

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

An SC-configuration of type $B \Pi B'$ will pop $1$ and then behave in $B$, or pop $r$ and then behave in $B'$.

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

A value $\Gamma \vdash^v V : U B$ 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 $s, M \mapsto s, M \bullet K$.

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

What should $K$ denote?

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


This gives an adjunction

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

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

For call-by-value we recover

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

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 α` means let $\alpha$ be the current stack.
- `changestk α` means change the current stack to $\alpha$. 
Naming and changing the current stack

Extend the language with two instructions:

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

Execution takes places in a bigger language.

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

\[
\frac{\Gamma \; \text{changepstk } K. \; M \quad B' \quad L \quad C \mid \Delta}{\Gamma \; M \quad B \quad K \quad C \mid \Delta}
\]

Similar to Crolard's syntax. Numerous variations in the literature.
We have typing judgements:

$$\Gamma \vdash^y V : A \mid \Delta \quad \Gamma \vdash^c M : B \mid \Delta$$

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

We have typing judgements:

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

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

Example typing rules

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

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

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

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:

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

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

Example typing rules

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

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

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

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

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

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

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

\textbf{Maybe} we can build a denotational semantics where a computation type $B$ denotes an Eilenberg-Moore algebra $\mathbb{B}_{\text{alg}}$, 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 \texttt{nil},
would denote an element of \([B]\).
Intrinsic semantics of control

The denotation of \( B \) is a semantic domain for stacks from \( B \).

That means: a hypothetical extremely closed stack from \( B \), with no free identifiers and no \texttt{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 $B \Pi B'$ is a tagged stack $^1 :: K$ or $^r :: K$.

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

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

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

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

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

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

That gives an adjunction

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

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

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

For call-by-value we recover

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

This is standard.
Abbreviate \( \neg X \overset{\text{def}}{=} X \to R \).

For call-by-value we recover

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

This is standard.

For call-by-name we recover

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

This is Streicher and Reus' semantics for a CBN language with control operators.
For a set $E$, the adjunction

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

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

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

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

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

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

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

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

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