

# $\lambda$ -calculus, effects and call-by-push-value

Paul Blain Levy

University of Birmingham

April 18, 2026

# Outline

- 1 Pure  $\lambda$ -calculus
  - Syntax
  - Denotational semantics
  - The  $\beta\eta$ -theory
  - Reversible rules
  - Operational semantics
- 2 Adding Effects
  - Outline
  - Errors and printing, operationally
- 3 Call-by-value with errors
  - Denotational semantics
  - Substitution and values
  - Fine-grain call-by-value
- 4 Call-by-name with errors
- 5 Call-by-push-value
- 6 Stacks
- 7 State
- 8 Control

# Types

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

Here is the syntax of types:

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

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  $A ::= \dots \mid (A)$ .
- 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 list of typed distinct identifiers.
- $M$  is a term.
- $A$  is a type.

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

Free identifier

$$\frac{}{\Gamma \vdash \mathbf{x} : A} (\mathbf{x} : A) \in \Gamma$$

Multiple local declaration, e.g. of two identifiers

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M' : B \quad \Gamma, \mathbf{x} : A, \mathbf{y} : B \vdash N : C}{\Gamma \vdash \mathbf{let} (\mathbf{x} \mathbf{be} M, \mathbf{y} \mathbf{be} M'). N : C}$$

# 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:

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{bool}}$$

Elimination rule

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

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

# Typing rules for arithmetic

These are *ad hoc* rules.

$$\frac{}{\Gamma \vdash 17 : \text{nat}} \qquad \frac{\Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat}}{\Gamma \vdash M + M' : \text{nat}}$$

# Typing rules for $A + B$

Two introduction rules

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

Elimination rule

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

# Typing rules for $A + B$

Two introduction rules

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

Elimination rule

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

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

# Typing rules for 0

Zero introduction rules

Elimination rule

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \text{match } M \text{ as } \{ \}^A : A}$$

# Typing rules for $A \times B$

Introduction rule

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

Two options for elimination

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

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

- **Projection product.** Two elimination rules

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

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M^r : B}$$

# Typing rules for $A \times B$

Introduction rule

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

Two options for elimination

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

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

- **Projection product.** Two elimination rules

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

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M^r : B}$$

$\prod_{i \in \mathbb{N}} A_i$  is a projection product.

# Typing rules for 1

## Introduction rule

$$\frac{}{\Gamma \vdash \langle \rangle : 1}$$

## Two options for elimination

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

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

- **Projection unit.** Zero elimination rules

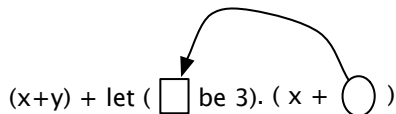
## Theorem

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

# Binding diagrams (Quine, Bourbaki)

## 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.

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

- The collection of binding diagrams forms an initial algebra [FPT; AR].
- We'll skate over this issue. It's not specific to  $\lambda$ -calculus.

# Substitution

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

# 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

$$M = \lambda y_{\text{nat}}. y + 3$$

$$M' = 7$$

$$N = x(5 + y)$$

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

# Types denote sets

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

# Types denote sets

- Every type  $A$  denotes a set  $\llbracket A \rrbracket$ .
- For example,  $\llbracket \text{nat} \rightarrow \text{nat} \rrbracket$  is the set of functions  $\mathbb{N} \rightarrow \mathbb{N}$ .
- $\llbracket A \rrbracket$  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  $\llbracket A \rrbracket$ .
- For example,  $\llbracket \text{nat} \rightarrow \text{nat} \rrbracket$  is the set of functions  $\mathbb{N} \rightarrow \mathbb{N}$ .
- $\llbracket A \rrbracket$  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  $\lambda a \in \mathbb{N}. a + 3$ .

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

$$\llbracket \text{bool} \rrbracket = \mathbb{B} = \{\text{true}, \text{false}\}$$

$$\llbracket \text{nat} \rrbracket = \mathbb{N}$$

$$\llbracket A \rightarrow B \rrbracket = \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket$$

$$\llbracket 1 \rrbracket = 1 = \{\langle \rangle\}$$

$$\llbracket A + B \rrbracket = \llbracket A \rrbracket + \llbracket B \rrbracket$$

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

$$\llbracket 0 \rrbracket = \emptyset$$

Let  $\Gamma$  be a typing context.

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

$$\llbracket \Gamma \rrbracket \stackrel{\text{def}}{=} \prod_{(x:A) \in \Gamma} \llbracket A \rrbracket$$

# Semantics of typing judgement

Given a typing judgement  $\Gamma \vdash M : A$ ,

we shall define  $\llbracket M \rrbracket$ , or more precisely  $\llbracket \Gamma \vdash M : A \rrbracket$ .

It's a function from  $\llbracket \Gamma \rrbracket$  to  $\llbracket A \rrbracket$ .

## Example

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

denotes the function

$$\begin{aligned} \llbracket x : \text{nat}, y : \text{nat} \rrbracket &\longrightarrow (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \\ \rho &\longmapsto \lambda z \in \mathbb{N} \rightarrow \mathbb{N}. z(\rho_x + \rho_y) \end{aligned}$$

$$\frac{}{\Gamma \vdash 17 : \text{nat}}$$

$$\llbracket 17 \rrbracket : \rho \mapsto 17$$

$$\frac{\Gamma \vdash M : \text{nat} \quad \Gamma \vdash M' : \text{nat}}{\Gamma \vdash M + M' : \text{nat}}$$

$$\llbracket M + M' \rrbracket : \rho \mapsto \llbracket M \rrbracket \rho + \llbracket M' \rrbracket \rho$$

# More semantic equations

$$\frac{}{\Gamma \vdash \mathbf{x} : A} (\mathbf{x} : A) \in \Gamma$$

$$\llbracket \mathbf{x} \rrbracket : \rho \mapsto \rho_{\mathbf{x}}$$

$$\frac{\Gamma, \mathbf{x} : A \vdash M : B}{\Gamma \vdash \lambda \mathbf{x}_A. M : A \rightarrow B}$$

$$\llbracket \lambda \mathbf{x}_A. M \rrbracket : \rho \mapsto \lambda a \in \llbracket A \rrbracket. \llbracket M \rrbracket(\rho, \mathbf{x} \mapsto a)$$

# More semantic equations

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

$$\llbracket \mathbf{inl}^{A,B} M \rrbracket : \rho \mapsto \mathbf{inl} \llbracket M \rrbracket \rho$$

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

$$\llbracket \mathbf{match} M \text{ as } \{\mathbf{inl} x. N, \mathbf{inr} y. N'\} \rrbracket : \rho \mapsto \mathbf{match} \llbracket M \rrbracket \rho \text{ as } \{\mathbf{inl} a. \llbracket N \rrbracket(\rho, x \mapsto a), \mathbf{inr} b. \llbracket N' \rrbracket(\rho, y \mapsto b)\}$$

## 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  $\llbracket \Gamma \vdash M : A \rrbracket$  doesn't depend on the derivation.

## Weakening Lemma

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

$$\llbracket \Gamma' \vdash M : A \rrbracket \rho = \llbracket \Gamma \vdash M \rrbracket (\rho \upharpoonright_{\Gamma})$$

## 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.

## 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  $\llbracket N \rrbracket$  and  $\llbracket M \rrbracket$  and  $\llbracket M' \rrbracket$ .

$$\llbracket N[M/x, M'/y] \rrbracket : \rho \mapsto \llbracket N \rrbracket(\rho, x \mapsto \llbracket M \rrbracket \rho, y \mapsto \llbracket M' \rrbracket \rho)$$

The  $\beta$ -law for  $A \rightarrow B$

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

Introduction inside an elimination may be removed.

The  $\beta$ -law for  $A \rightarrow B$

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

Introduction inside an elimination may be removed.

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

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A'}{\Gamma \vdash \langle M, N \rangle^1 = M : A}$$

Zero  $\beta$ -laws for projection unit 1

Two  $\beta$ -laws for `bool`

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

Two  $\beta$ -laws for `bool`

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

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

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

Two  $\beta$ -laws for `bool`

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

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

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

Zero  $\beta$ -laws for `0`

## $\beta$ -law for local declaration

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M' : B \quad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \mathbf{let} (x \mathbf{be} M, y \mathbf{be} M'). N = N[M/x, M'/y] : C}$$

$\eta$ -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.

$\eta$ -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.

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

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M = \langle M^l, M^r \rangle : A \times B}$$

$\eta$ -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

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

## More $\eta$ -laws

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

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

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

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

## More $\eta$ -laws

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

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

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

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

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

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

# The $\beta\eta$ -theory

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:

$$\frac{\Gamma, \mathbf{x} : A \vdash M = M' : B}{\Gamma \vdash \lambda \mathbf{x}_A. M = \lambda \mathbf{x}_A. M' : A \rightarrow B}$$

## Closure Theorems

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

$$\begin{array}{l} z : 0 \vdash \text{true} =_{\beta\eta} \text{false} : \text{bool} \\ \text{but not} \quad \vdash \text{true} =_{\beta\eta} \text{false} : \text{bool} \end{array}$$

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

## Soundness theorem

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

Follows from the weakening and substitution lemmas.

# Reversible rule for $A \rightarrow B$

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

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

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

# Reversible rule for $A \rightarrow B$

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

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

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

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

# Reversible rule for $A \rightarrow B$

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

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

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

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

$A \rightarrow B$  appears on the **right** of  $\vdash$  in the conclusion.

# Reversible rule for bool

The (nullary) connective `bool` is **leftist**.

That means: it has a reversible rule

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

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

`bool` appears on the **left** of  $\vdash$  in the conclusion.

## Reversible rule for $A + B$

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

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

natural in  $\Gamma$  and  $C$ .

## Reversible rule for $A + B$

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

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

natural in  $\Gamma$  and  $C$ .

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

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

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.

# 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

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

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

It also has a reversible rule

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

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

# Bipartisan connectives

The connective  $\times$  has a reversible rule

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

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

It also has a reversible rule

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

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

In summary, the connective  $\times$  is **bipartisan**.

Likewise the (nullary) connective  $1$ .

# Most general leftist connective

The **variant tuple type**  $\boxed{\Sigma} \{^0 A, A'; ^1 B, B', B''\}$  denotes a sum of products

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

This gives a leftist connective.

$$\frac{\Gamma, A, A' \vdash C \quad \Gamma, B, B', B'' \vdash C}{\Gamma, \boxed{\Sigma} \{^0 A, A'; ^1 B, B', B''\} \vdash C}$$

# Most general leftist connective

The **variant tuple type**  $\boxed{\Sigma} \{^0 A, A'; ^1 B, B', B''\}$  denotes a sum of products

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

This gives a leftist connective.

$$\frac{\Gamma, A, A' \vdash C \quad \Gamma, B, B', B'' \vdash C}{\Gamma, \boxed{\Sigma} \{^0 A, A'; ^1 B, B', B''\} \vdash C}$$

Here is its term syntax:

$$\begin{aligned} & \text{in}_0(M, M') \\ & \text{in}_1(M, M', M'') \\ \text{match } M \text{ as } & \{ \text{in}_0(x, x'). N, \text{in}_1(y, y', y''). N' \} \end{aligned}$$

# Most general rightist connective

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

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

This gives a rightist connective.

$$\frac{\Gamma, A, A' \vdash B \quad \Gamma, C, C', C'' \vdash D}{\Gamma \vdash \boxed{\Pi} \{^0 A, A' \vdash B; ^1 C, C', C'' \vdash D\}}$$

# Most general rightist connective

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

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

This gives a rightist connective.

$$\frac{\Gamma, A, A' \vdash B \quad \Gamma, C, C', C'' \vdash D}{\Gamma \vdash \boxed{\Pi} \{^0 A, A' \vdash B; ^1 C, C', C'' \vdash D\}}$$

Here is its term syntax:

$$\lambda\{^0(\mathbf{x}, \mathbf{x}').M, ^1(y, y', y'').M'\} \\ M^0(N, N') \\ M^1(N, N', N'')$$

Type syntax

$$A ::= \boxed{\Sigma} \{\vec{A}_i\}_{i < n} \quad | \quad \boxed{\Pi} \{\vec{A}_i \vdash B_i\}_{i < n} \quad (n \in \mathbb{N} \text{ or } n = \infty)$$

Term syntax, with type annotations omitted

$$\begin{aligned} M ::= & \quad \mathbf{x} \quad | \quad \mathbf{let} \ (\overrightarrow{\mathbf{x} \text{ be } \vec{M}}) . M \\ & \quad | \quad \mathbf{in}_i(\vec{M}) \\ & \quad | \quad \mathbf{match} \ M \ \mathbf{as} \ \{\mathbf{in}_i(\vec{\mathbf{x}}) . M_i\}_{i < n} \\ & \quad | \quad \lambda\{^i(\vec{\mathbf{x}}) . M_i\}_{i < n} \\ & \quad | \quad M^i(\vec{M}) \end{aligned}$$

Type syntax

$$A ::= \boxed{\Sigma} \{\vec{A}_i\}_{i < n} \quad | \quad \boxed{\Pi} \{\vec{A}_i \vdash B_i\}_{i < n} \quad (n \in \mathbb{N} \text{ or } n = \infty)$$

Term syntax, with type annotations omitted

$$\begin{aligned} M ::= & \quad \mathbf{x} \quad | \quad \text{let } (\overrightarrow{\mathbf{x} \text{ be } \vec{M}}). M \\ & \quad | \quad \text{in}_i(\vec{M}) \\ & \quad | \quad \text{match } M \text{ as } \{\text{in}_i(\vec{\mathbf{x}}). M_i\}_{i < n} \\ & \quad | \quad \lambda\{^i(\vec{\mathbf{x}}). M_i\}_{i < n} \\ & \quad | \quad M^i(\vec{M}) \end{aligned}$$

Includes both pattern-match product  $A \times B$   
and projection product  $A \Pi B$ .

# Jumbo vs non-jumbo

Jumbo  $\lambda$ -calculus is the most expressive form of simply typed  $\lambda$ -calculus: it contains all leftist and rightist connectives as primitives.

# Jumbo vs non-jumbo

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 vs non-jumbo

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  $M N$ , 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  $\lambda$ -calculus”.
- In the untyped literature, “call-by-name” evaluation means reduction to head normal form.

# Evaluation order for `let`

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

- evaluate  $M$  to  $T$  and  $M'$  to  $T'$ , then evaluate  $N[T/x, T'/y]$ ?  
**Call-by-value**
- just evaluate  $N[M/x, M'/y]$ ? **Call-by-name**

# Evaluation order for application

To evaluate  $M N$ , we must evaluate  $M$  to  $\lambda_{\mathbf{x}_A}. P$ . Do we

- evaluate  $N$  to  $T$  (before or after evaluating  $M$ ), then evaluate  $P[T/\mathbf{x}]$ ? **Call-by-value**
- just evaluate  $P[N/\mathbf{x}]$ ? **Call-by-name**

# Terminal terms of type $A + B$

Any terminal term of type  $A + B$  must be  $\text{inl } M$  or  $\text{inr } M$ . Do we

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

Consider evaluation of  $\text{match } P$  as  $\{\text{inl } x. N, \text{inr } y. N'\}$  to see this.

# Definitional interpreter for call-by-value

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

To evaluate

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

# Big-step semantics for call-by-value

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

This is defined inductively, for example

$$\frac{M \Downarrow \lambda \mathbf{x}_A. P \quad N \Downarrow T \quad P[T/\mathbf{x}] \Downarrow T'}{MN \Downarrow T'}$$

# Big-step semantics for call-by-value

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

This is defined inductively, for example

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

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

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

# 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

$$\frac{M \Downarrow \lambda \mathbf{x}_A. P \quad P[N/\mathbf{x}] \Downarrow T}{MN \Downarrow T}$$

# 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

$$\frac{M \Downarrow \lambda \mathbf{x}_A. P \quad P[N/\mathbf{x}] \Downarrow T}{MN \Downarrow T}$$

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

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

## The experiment

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

## The experiment

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

## Analyzing CBV with a microscope

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

# 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.

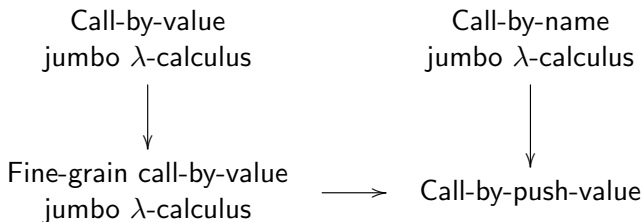
## Analyzing CBV with a microscope

- 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**.

# The big picture



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

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

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

## Printing

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

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

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

- ① Evaluate

```
let (x be error CRASH). 5
```

in CBV and CBN.

- ② Evaluate

```
( $\lambda x.(x + x)$ )(print "hello". 4)
```

in CBV and CBN.

- ③ Evaluate

```
match (print "hello". inr error CRASH) as  
  {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  $T$ .
- $M \Downarrow e$  means  $M$  raises error  $e$ .

Here are the rules for application:

$$\frac{M \Downarrow e}{MN \Downarrow e} \qquad \frac{M \Downarrow \lambda x. P \quad N \Downarrow e}{MN \Downarrow e}$$
$$\frac{M \Downarrow \lambda x. P \quad N \Downarrow T \quad P[T/x] \Downarrow e}{MN \Downarrow e}$$
$$\frac{M \Downarrow \lambda x. P \quad N \Downarrow T \quad P[T/x] \Downarrow T'}{MN \Downarrow T'}$$

Likewise for call-by-name.

# Observational equivalence

A **program** is a closed term of type `nat` or `bool`.

Two terms  $\Gamma \vdash M, M' : B$  are **observationally equivalent**

when  $\mathcal{C}[M]$  and  $\mathcal{C}[M']$  have the same behaviour

for every program with a hole  $\mathcal{C}[\cdot]$ .

**Same behaviour** means: print the same string, raise the same error, return the same boolean.

We write  $M \simeq_{\text{CBV}} M'$  and  $M \simeq_{\text{CBN}} M'$ .

# The $\eta$ -law for boolean type: has it survived?

## $\eta$ -law for bool

Any term  $\Gamma, z : \text{bool} \vdash M : B$  can be expanded as

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

Anything of boolean type is a boolean.

This holds in CBV, because  $z$  can only be replaced by `true` or `false`.

But it's broken in CBN, because  $z$  might raise an error. For example,

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

because we can apply the context

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

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  $A \amalg B$

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

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

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

$$\text{error } e \simeq_{\text{CBN}} \lambda x. \text{error } e$$

$$\text{error } e \simeq_{\text{CBN}} \lambda\{^1. \text{error } e, {}^r. \text{error } e\}$$

$$\text{print } c. \lambda x. M \simeq_{\text{CBN}} \lambda x. \text{print } c. M$$

$$\text{print } c. \lambda\{^1. M, {}^r. N\} \simeq_{\text{CBN}} \lambda\{^1. \text{print } c. M, {}^r. \text{print } c. N\}$$

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.
- $A \times B \cong A \amalg B$  survives in neither CBV nor CBN.
- $A \rightarrow (B \rightarrow C) \cong (A \amalg B) \rightarrow C$  survives in CBN but not CBV.

Our first attempt.

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

$$\begin{aligned} \llbracket \text{bool} \rrbracket_* &= \mathbb{B} + E \\ \llbracket \text{bool} + \text{bool} \rrbracket_* &= (\mathbb{B} + \mathbb{B}) + E \\ \llbracket \text{bool} \times \text{bool} \rrbracket_* &= (\mathbb{B} \times \mathbb{B}) + E \end{aligned}$$

Our first attempt.

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

$$\begin{aligned} \llbracket \text{bool} \rrbracket_* &= \mathbb{B} + E \\ \llbracket \text{bool} + \text{bool} \rrbracket_* &= (\mathbb{B} + \mathbb{B}) + E \\ \llbracket \text{bool} \times \text{bool} \rrbracket_* &= (\mathbb{B} \times \mathbb{B}) + E \end{aligned}$$

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

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

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

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

$$\begin{aligned} \llbracket \mathbf{bool} \rrbracket &= \mathbb{B} \\ \llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow (\llbracket B \rrbracket + E) \\ \llbracket () \rightarrow B \rrbracket &= \llbracket B \rrbracket + E \\ \llbracket \Gamma \rrbracket &= \prod_{(x:A) \in \Gamma} \llbracket A \rrbracket \end{aligned}$$

Each term  $\Gamma \vdash M : B$  denotes a function  $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow (\llbracket B \rrbracket + E)$ .

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

$$\llbracket \lambda \mathbf{x}_A. M \rrbracket : \rho \mapsto \text{inl } \lambda a \in \llbracket A \rrbracket. \llbracket M \rrbracket(\rho, \mathbf{x} \mapsto a)$$

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

$$\llbracket M N \rrbracket : \rho \mapsto \text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl } f. & \text{match } \llbracket N \rrbracket \rho \text{ as } \begin{cases} \text{inl } x. & f(x) \\ \text{inr } e. & \text{inr } e \end{cases} \\ \text{inr } e. & \text{inr } e \end{cases}$$

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

$$\llbracket \mathbf{inl}^{A,B} M \rrbracket : \rho \longmapsto \text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl } a. & \text{inl inl } a \\ \text{inr } e. & \text{inr } e \end{cases}$$

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

$$\llbracket \mathbf{inl}^{A,B} M \rrbracket : \rho \longmapsto \text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl } a. & \text{inl inl } a \\ \text{inr } e. & \text{inr } e \end{cases}$$

To prove the soundness of the denotational semantics, we need a substitution lemma.

# CBV Substitution Lemma: What Doesn't Work

Can we obtain  $\llbracket N[M/x] \rrbracket$  from  $\llbracket M \rrbracket$  and  $\llbracket N \rrbracket$ ?

# CBV Substitution Lemma: What Doesn't Work

Can we obtain  $\llbracket N[M/x] \rrbracket$  from  $\llbracket M \rrbracket$  and  $\llbracket N \rrbracket$ ? **Not in CBV.**

# CBV Substitution Lemma: What Doesn't Work

Can we obtain  $\llbracket N[M/x] \rrbracket$  from  $\llbracket M \rrbracket$  and  $\llbracket N \rrbracket$ ? **Not in CBV.**

## Example that rules out a general substitution lemma

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

$M \stackrel{\text{def}}{=} \text{error CRASH}$

$N \stackrel{\text{def}}{=} \text{true}$

$N' \stackrel{\text{def}}{=} \text{match } x \text{ as } \{\text{true.true, false.true}\}$

$\llbracket N \rrbracket = \llbracket N' \rrbracket$       **because  $N =_{\eta \text{bool}} N'$**

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

# CBV Substitution Lemma: What Doesn't Work

Can we obtain  $\llbracket N[M/x] \rrbracket$  from  $\llbracket M \rrbracket$  and  $\llbracket N \rrbracket$ ? **Not in CBV.**

Example that rules out a general substitution lemma

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

$$M \stackrel{\text{def}}{=} \text{error CRASH}$$
$$N \stackrel{\text{def}}{=} \text{true}$$
$$N' \stackrel{\text{def}}{=} \text{match } x \text{ as } \{\text{true.true, false.true}\}$$
$$\llbracket N \rrbracket = \llbracket N' \rrbracket \quad \text{because } N =_{\eta \text{bool}} N'$$
$$\llbracket N[M/x] \rrbracket \neq \llbracket N'[M/x] \rrbracket$$

But we can give a lemma for the substitution of **values**.

The following terms are called **values**.

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

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

$$\text{match true as } \{\text{true.false}, \text{false.true}\}$$

# Denotational semantics of values

Each value  $\Gamma \vdash V : A$  denotes a function  $\llbracket V \rrbracket^{\text{val}} : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket$ .

$$\llbracket \mathbf{x} \rrbracket^{\text{val}} : \rho \longmapsto \rho_{\mathbf{x}}$$

$$\llbracket \mathbf{true} \rrbracket^{\text{val}} : \rho \longmapsto \mathbf{true}$$

$$\llbracket \mathbf{inl } V \rrbracket^{\text{val}} : \rho \longmapsto \mathbf{inl } \llbracket V \rrbracket^{\text{val}} \rho$$

$$\llbracket \lambda \mathbf{x}_A. M \rrbracket^{\text{val}} : \rho \longmapsto \lambda a \in \llbracket A \rrbracket. \llbracket M \rrbracket(\rho, \mathbf{x} \mapsto \llbracket a \rrbracket)$$

We can recover  $\llbracket V \rrbracket$  from  $\llbracket V \rrbracket^{\text{val}}$ .

$$\llbracket V \rrbracket : \rho \longmapsto \mathbf{inl } \llbracket V \rrbracket^{\text{val}} \rho$$

# Substitution Lemma For Values

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

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

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  $M \not\Downarrow e$  then  $\llbracket M \rrbracket_{\varepsilon} = \text{inr } e$ .

Proof by induction, using the substitution lemma.

# Fine-Grain Call-By-Value

Fine-grain call-by-value has two judgements:

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

Key typing rules

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

Corresponds to Power and Robinson's notion of a **Freyd category**.

# Semantics of returning and sequencing

$$\frac{\Gamma \vdash^v V : A}{\Gamma \vdash^c \mathbf{return} V : A}$$

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

$$\frac{\Gamma \vdash^c M : A \quad \Gamma, \mathbf{x} : A \vdash^c N : B}{\Gamma \vdash^c M \mathbf{to} \mathbf{x}. N : B}$$

$$\llbracket M \mathbf{to} \mathbf{x}. N \rrbracket : \rho \mapsto \mathbf{match} \llbracket M \rrbracket \rho \mathbf{as} \begin{cases} \mathbf{inl} a. & \llbracket N \rrbracket (\rho, \mathbf{x} \mapsto a) \\ \mathbf{inr} e. & \mathbf{inr} e \end{cases}$$

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

$$\begin{aligned} V &::= x \mid \text{true} \mid \text{false} \\ &\quad \mid \text{inl } V \mid \text{inr } V \mid \lambda x. M \\ M &::= M \text{ to } x. M \mid \text{return } V \\ &\quad \mid \text{let } (\overrightarrow{x \text{ be } V}). M \mid V V \\ &\quad \mid \text{match } V \text{ as } \{\text{true}. M, \text{false}. M\} \\ &\quad \mid \text{match } V \text{ as } \{\text{inl } x. M, \text{inr } x. M\} \\ &\quad \mid \text{error } e \end{aligned}$$

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

$$\begin{aligned} V & ::= x \mid \text{true} \mid \text{false} \\ & \quad \mid \text{inl } V \mid \text{inr } V \mid \lambda x. M \\ M & ::= M \text{ to } x. M \mid \text{return } V \\ & \quad \mid \text{let } (\overrightarrow{x \text{ be } V}). M \mid V V \\ & \quad \mid \text{match } V \text{ as } \{\text{true}. M, \text{false}. M\} \\ & \quad \mid \text{match } V \text{ as } \{\text{inl } x. M, \text{inr } x. M\} \\ & \quad \mid \text{error } e \end{aligned}$$

We don't allow “complex values” such as

$$\text{match true as } \{\text{true}. \text{false}, \text{false}. \text{true}\}$$

These would complicate the operational semantics.

# Definitional interpreter for fine-grain CBV

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

$$\text{match } (\text{inl } V) \text{ as } \{\text{true}. M, \text{false}. M'\} = M[V/x]$$

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

$$\text{let } (x \text{ be } V, y \text{ be } W). M = M[V/x, W/y]$$

## $\eta$ -laws

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

$$V = \lambda x. Vx$$

## 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  $\Gamma \vdash^c \hat{M} : A$ .

$$\begin{aligned}x &\longmapsto \text{return } x \\ \lambda x. M &\longmapsto \text{return } \lambda x. \hat{M} \\ \text{inl } M &\longmapsto \hat{M} \text{ to } x. \text{return inl } x \\ M N &\longmapsto \hat{M} \text{ to } x. \hat{N} \text{ to } y. xy \\ \text{let } (x \text{ be } M, y \text{ be } M'). N &\longmapsto \hat{M} \text{ to } x. \hat{M}' \text{ to } y. \hat{N}\end{aligned}$$

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

$$\begin{aligned}x &\longmapsto x \\ \lambda x. M &\longmapsto \lambda x. \hat{M} \\ \text{inl } V &\longmapsto \text{inl } \check{V}\end{aligned}$$

# Nullary functions

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

$$\begin{array}{ll} TA & \stackrel{\text{def}}{=} () \rightarrow A & \llbracket TA \rrbracket & = & \llbracket A \rrbracket + E \\ \mathbf{thunk} M & \stackrel{\text{def}}{=} \lambda(). M & \llbracket \mathbf{thunk} M \rrbracket & = & \llbracket M \rrbracket \\ \mathbf{force} V & \stackrel{\text{def}}{=} V () & \llbracket \mathbf{force} V \rrbracket & = & \llbracket V \rrbracket \end{array}$$

# Nullary functions

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

$$\begin{array}{ll} TA & \stackrel{\text{def}}{=} () \rightarrow A & \llbracket TA \rrbracket & = \llbracket A \rrbracket + E \\ \mathbf{thunk} M & \stackrel{\text{def}}{=} \lambda(). M & \llbracket \mathbf{thunk} M \rrbracket & = \llbracket M \rrbracket \\ \mathbf{force} V & \stackrel{\text{def}}{=} V () & \llbracket \mathbf{force} V \rrbracket & = \llbracket V \rrbracket \end{array}$$

The type  $TA$  has a reversible rule 
$$\frac{\Gamma \vdash^c A}{\Gamma \vdash^v TA}$$

# Nullary functions

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

$$\begin{array}{ll} TA & \stackrel{\text{def}}{=} () \rightarrow A & \llbracket TA \rrbracket & = \llbracket A \rrbracket + E \\ \text{thunk } M & \stackrel{\text{def}}{=} \lambda(). M & \llbracket \text{thunk } M \rrbracket & = \llbracket M \rrbracket \\ \text{force } V & \stackrel{\text{def}}{=} V () & \llbracket \text{force } V \rrbracket & = \llbracket V \rrbracket \end{array}$$

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.

# Naive CBN semantics of errors

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

$$\begin{aligned} \llbracket \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \rrbracket_* &= (\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E)) \\ \llbracket \text{bool} + \text{bool} \rrbracket_* &= ((\mathbb{B} + E) + (\mathbb{B} + E)) + E \\ \llbracket \text{bool} \amalg \text{bool} \rrbracket_* &= (\mathbb{B} + E) \times (\mathbb{B} + E) \end{aligned}$$

Thus we define

$$\begin{aligned} \llbracket \text{bool} \rrbracket_* &= \mathbb{B} + E \\ \llbracket A + B \rrbracket_* &= (\llbracket A \rrbracket_* + \llbracket B \rrbracket_*) + E \\ \llbracket A \rightarrow B \rrbracket_* &= \llbracket A \rrbracket_* \rightarrow \llbracket B \rrbracket_* \\ \llbracket A \amalg B \rrbracket_* &= \llbracket A \rrbracket_* \times \llbracket B \rrbracket_* \\ \llbracket \Gamma \rrbracket &= \prod_{(x:A) \in \Gamma} \llbracket A \rrbracket_* \end{aligned}$$

Each term  $\Gamma \vdash M : B$  should denote a function  $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket B \rrbracket_*$ .

# Naive semantics: what goes wrong

$$\frac{}{\Gamma \vdash \text{error CRASH} : B}$$

denotes  $\rho \mapsto ?$

# Naive semantics: what goes wrong

$\frac{}{\Gamma \vdash \text{error CRASH} : B}$  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.

# Naive semantics: what goes wrong

$\frac{}{\Gamma \vdash \text{error CRASH} : B}$  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 `match`.

## Solution: $E$ -pointed sets

### Definition

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

A type should denote an  $E$ -pointed set, a [semantic domain for terms](#).

# Solution: $E$ -pointed sets

## Definition

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

A type should denote an  $E$ -pointed set, a [semantic domain for terms](#).

Examples:

$$\begin{aligned} \llbracket \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \rrbracket &= ((\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E))), \\ &\quad \lambda x. \lambda y. \text{inr CRASH}, \\ &\quad \lambda x. \lambda y. \text{inr BANG}) \\ \llbracket \text{bool} + \text{bool} \rrbracket &= (((\mathbb{B} + E) + (\mathbb{B} + E)) + E, \\ &\quad \text{inr CRASH}, \\ &\quad \text{inr BANG}) \\ \llbracket \text{bool} \amalg \text{bool} \rrbracket &= ((\mathbb{B} + E) \times (\mathbb{B} + E), \\ &\quad (\text{inr CRASH}, \text{inr CRASH}), \\ &\quad (\text{inr BANG}, \text{inr BANG})) \end{aligned}$$

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

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

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

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

and  $\llbracket A \amalg B \rrbracket = (X \times Y, (c, c'), (b, b'))$

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

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

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

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

and  $\llbracket A \amalg B \rrbracket = (X \times Y, (c, c'), (b, b'))$

$$\llbracket \Gamma \rrbracket = \prod_{\substack{(x:A) \in \Gamma \\ \llbracket A \rrbracket = (X, c, b)}} X$$

A term  $\Gamma \vdash M : B$  denotes a function  $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket B \rrbracket$ .

# Semantics of term constructors

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}$$

$$\llbracket \text{true} \rrbracket : \rho \mapsto \text{inl true}$$

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

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

$$\text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl true.} & \llbracket N \rrbracket \rho \\ \text{inl false.} & \llbracket N' \rrbracket \rho \\ \text{inr CRASH.} & c \\ \text{inr BANG.} & b \end{cases} \quad \text{where } \llbracket B \rrbracket = (Y, c, b)$$

## More term constructors

$$\begin{aligned} \llbracket \lambda \mathbf{x}. M \rrbracket & : \rho \mapsto \lambda a. \llbracket M \rrbracket (\rho, \mathbf{x} \mapsto a) \\ \llbracket M N \rrbracket & : \rho \mapsto \llbracket M \rrbracket \llbracket N \rrbracket \\ \llbracket \mathbf{x} \rrbracket & : \rho \mapsto \rho_{\mathbf{x}} \\ \text{error CRASH} & : \rho \mapsto c \end{aligned}$$

### Soundness/adequacy

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

Proved by induction, using the substitution lemma.

# Notation for $E$ -pointed sets

- Free  $E$ -pointed set on a set  $X$ .

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

- Product of two  $E$ -pointed sets.

$$(X, c, b) \amalg (Y, c', b') \stackrel{\text{def}}{=} (X \times Y, (c, c'), (b, b'))$$

- Unit  $E$ -pointed set.  $1_{\amalg} \stackrel{\text{def}}{=} (1, (), ())$

- Product of a family of  $E$ -pointed sets.

$$\prod_{i \in I} (X_i, c_i, b_i) \stackrel{\text{def}}{=} \left( \prod_{i \in I} X_i, \lambda i. c_i, \lambda i. b_i \right)$$

- Exponential  $E$ -pointed set.

$$\begin{aligned} X \rightarrow (Y, c, b) &\stackrel{\text{def}}{=} \prod_{x \in X} (Y, c, b) \\ &= (X \rightarrow Y, \lambda x. c, \lambda x. b) \end{aligned}$$

- Carrier of an  $E$ -pointed set.  $U^E(X, c, b) \stackrel{\text{def}}{=} X$

# Summary of call-by-name semantics

A type denotes an  $E$ -pointed set.

$$\begin{aligned} \llbracket \text{bool} \rrbracket &= F^E(1 + 1) \\ \llbracket A + B \rrbracket &= F^E(U^E \llbracket A \rrbracket + U^E \llbracket B \rrbracket) \\ \llbracket A \rightarrow B \rrbracket &= U^E \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \\ \llbracket A \amalg B \rrbracket &= \llbracket A \rrbracket \amalg \llbracket B \rrbracket \end{aligned}$$

A typing context denotes a set.

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

A term  $\Gamma \vdash M : B$  denotes a function  $\llbracket \Gamma \rrbracket \rightarrow \llbracket B \rrbracket$ .

# Summary of call-by-value semantics

A type denotes a set.

$$\begin{aligned} \llbracket \mathbf{bool} \rrbracket &= 1 + 1 \\ \llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\ \llbracket A \rightarrow B \rrbracket &= U^E(\llbracket A \rrbracket \rightarrow F^E \llbracket B \rrbracket) \\ \llbracket TB \rrbracket &= U^E F^E \llbracket B \rrbracket \end{aligned}$$

A typing context denotes a set.

$$\llbracket \Gamma \rrbracket = \prod_{(x:A) \in \Gamma} \llbracket A \rrbracket$$

A computation  $\Gamma \vdash^c M : B$  denotes a function  $\llbracket \Gamma \rrbracket \longrightarrow F^E \llbracket B \rrbracket$ .

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

computation type     $\underline{B} ::= FA \mid A \rightarrow \underline{B} \mid 1_{\perp} \mid \underline{B} \amalg \underline{B} \mid \prod_{i \in \mathbb{N}} \underline{B}_i$

# Call-By-Push-Value Types

Two kinds of type:

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

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

computation type     $\underline{B} ::= FA \mid A \rightarrow \underline{B} \mid 1_{\perp} \mid \underline{B} \amalg \underline{B} \mid \prod_{i \in \mathbb{N}} \underline{B}_i$

**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**.

# Judgements

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**

$$\mathbf{x}_0 : A_0, \dots, \mathbf{x}_{m-1} : A_{m-1}$$

# Judgements

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**

$$\mathbf{x}_0 : A_0, \dots, \mathbf{x}_{m-1} : A_{m-1}$$

Two judgements:

- A value  $\Gamma \vdash^v V : A$  denotes a function  $\llbracket V \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket$ .
- A computation  $\Gamma \vdash^c M : \underline{B}$  denotes a function  $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \longrightarrow \llbracket \underline{B} \rrbracket$ .

# The type $FA$

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

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

Sequencing in the style of Filinski's "Effect-PCF".

# The type $FA$

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

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

Sequencing in the style of Filinski's "Effect-PCF".

$$\begin{aligned} \llbracket \mathbf{return} V \rrbracket & : \rho \mapsto \text{inl } \llbracket V \rrbracket \rho \\ \llbracket M \mathbf{to} \mathbf{x}. N \rrbracket & : \rho \mapsto \\ & \text{match } \llbracket M \rrbracket \rho \text{ as } \begin{cases} \text{inl } a. & \llbracket N \rrbracket (\rho, \mathbf{x} \mapsto a) \\ \text{inr CRASH.} & c \\ \text{inr BANG.} & b \end{cases} \\ & \text{where } \llbracket \underline{B} \rrbracket = (Y, c, b) \end{aligned}$$

# The type $U\underline{B}$

A value in  $U\underline{B}$  is a **thunk** of a computation in  $\underline{B}$ .

$$\frac{\Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^v \mathbf{thunk} M : U\underline{B}}$$

$$\frac{\Gamma \vdash^v V : U\underline{B}}{\Gamma \vdash^c \mathbf{force} V : \underline{B}}$$

# The type $U\underline{B}$

A value in  $U\underline{B}$  is a **thunk** of a computation in  $\underline{B}$ .

$$\frac{\Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^v \mathbf{thunk} M : U\underline{B}} \qquad \frac{\Gamma \vdash^v V : U\underline{B}}{\Gamma \vdash^c \mathbf{force} V : \underline{B}}$$

$$\llbracket \mathbf{thunk} M \rrbracket = \llbracket M \rrbracket$$

$$\llbracket \mathbf{force} V \rrbracket = \llbracket V \rrbracket$$

An identifier is a value.

$$\frac{}{\Gamma \vdash^v \mathbf{x} : A} (\mathbf{x} : A) \in \Gamma$$

$$\frac{\Gamma \vdash^v V : A \quad \Gamma \vdash^v W : B \quad \Gamma, \mathbf{x} : A, \mathbf{y} : B \vdash^c M : \underline{C}}{\Gamma \vdash^c \mathbf{let} (\mathbf{x} \mathbf{be} V, \mathbf{y} \mathbf{be} W). M : \underline{C}}$$

$$\frac{\Gamma \vdash^v V : A}{\Gamma \vdash^v \mathbf{inl} V : A + A'}$$

$$\frac{\Gamma \vdash^v V : A'}{\Gamma \vdash^v \mathbf{inr} V : A + A'}$$

$$\frac{\Gamma \vdash^v V : A + A' \quad \Gamma, x : A \vdash^c M : \underline{B} \quad \Gamma, y : A \vdash^c M' : \underline{B}}{\Gamma \vdash^c \mathbf{match} V \text{ as } \{\mathbf{inl} x.M, \mathbf{inr} y.M'\} : \underline{B}}$$

$$\frac{\Gamma \vdash^v V : A \quad \Gamma \vdash^v V' : A'}{\Gamma \vdash^v \langle V, V' \rangle : A \times A'}$$

$$\frac{\Gamma \vdash^v V : A \times A' \quad \Gamma, x : A, y : A' \vdash^c M : \underline{B}}{\Gamma \vdash^c \mathbf{match} V \text{ as } \langle x, y \rangle.M : \underline{B}}$$

The rules for 1 are similar.

$$\frac{\Gamma, \mathbf{x} : A \vdash^c M : \underline{B}}{\Gamma \vdash^c \lambda \mathbf{x}. M : A \rightarrow \underline{B}} \qquad \frac{\Gamma \vdash^c M : A \rightarrow \underline{B} \quad \Gamma \vdash^v V : A}{\Gamma \vdash^c MV : \underline{B}}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \quad \Gamma \vdash^c M' : \underline{B}'}{\Gamma \vdash^c \lambda \{^1. M, ^r. M'\} : \underline{B} \amalg \underline{B}'}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \amalg \underline{B}'}{\Gamma \vdash^c M^1 : \underline{B}}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \amalg \underline{B}'}{\Gamma \vdash^c M^r : \underline{B}'}$$

$$\frac{\Gamma, \mathbf{x} : A \vdash^c M : \underline{B}}{\Gamma \vdash^c \lambda \mathbf{x}. M : A \rightarrow \underline{B}} \qquad \frac{\Gamma \vdash^c M : A \rightarrow \underline{B} \quad \Gamma \vdash^v V : A}{\Gamma \vdash^c MV : \underline{B}}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \quad \Gamma \vdash^c M' : \underline{B}'}{\Gamma \vdash^c \lambda \{^1. M, ^r. M'\} : \underline{B} \amalg \underline{B}'}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \amalg \underline{B}'}{\Gamma \vdash^c M^1 : \underline{B}}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \amalg \underline{B}'}{\Gamma \vdash^c M^r : \underline{B}'}$$

It is often convenient to write applications operand-first, as  $V^c M$  and  $^1 M$  and  $^r M$ .

# Definitional interpreter for call-by-push-value

The terminals are **computations**:  $\text{return } V$   $\lambda x.M$   $\lambda\{^1.M, \text{r}.M'\}$

# Definitional interpreter for call-by-push-value

The terminals are **computations**:  $\text{return } V$   $\lambda x.M$   $\lambda\{^1.M, {}^r.M'\}$

To evaluate

- **return**  $V$ : return **return**  $V$ .
- $M$  **to**  $x$ .  $N$ : evaluate  $M$ . If this returns **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 inl**  $V$  as  $\{\text{inl } x.M, \text{inr } y.M'\}$ : evaluate  $M[V/x]$ .
- **match**  $\langle V, V' \rangle$  as  $\langle x, y \rangle.M$ : evaluate  $M[V/x, V'/y]$ .
- **error**  $e$ , print error message  $e$  and stop.

# Equational theory

## $\beta$ -laws

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

## $\eta$ -laws

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

## Sequencing laws

$$\begin{aligned}(\text{return } V) \text{ to } x. M &= M[V/x] \\ M &= M \text{ to } x. \text{return } x \\ (M \text{ to } x. N) \text{ to } y. P &= M \text{ to } x. (N \text{ to } y. P)\end{aligned}$$

# 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.

$$\begin{aligned} A \rightarrow B &\longmapsto U(A \rightarrow FB) \\ TB &\longmapsto UFB \end{aligned}$$

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.

$$\begin{aligned} A \rightarrow B &\longmapsto U(A \rightarrow FB) \\ TB &\longmapsto UFB \end{aligned}$$

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

$$\begin{aligned} \lambda x. M &\longmapsto \mathbf{thunk} \lambda x. M \\ VW &\longmapsto (\mathbf{force} V) W \end{aligned}$$

# Decomposing CBV into CBPV

A CBV type translates into a value type.

$$\begin{aligned}A \rightarrow B &\longmapsto U(A \rightarrow FB) \\TB &\longmapsto UFB\end{aligned}$$

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

$$\begin{aligned}\lambda x. M &\longmapsto \text{thunk } \lambda x. M \\V W &\longmapsto (\text{force } V) W\end{aligned}$$

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

$$\begin{aligned}x &\longmapsto \text{return } x \\ \lambda x. M &\longmapsto \text{return thunk } \lambda x. M \\ M N &\longmapsto M \text{ to } f. N \text{ to } y. ((\text{force } f) y)\end{aligned}$$

# Decomposing CBN into CBPV

A CBN type translates into a computation type.

$$\begin{aligned}\text{bool} &\longmapsto F(1 + 1) \\ \underline{A} + \underline{B} &\longmapsto F(U\underline{A} + U\underline{B}) \\ \underline{A} \rightarrow \underline{B} &\longmapsto U\underline{A} \rightarrow \underline{B}\end{aligned}$$

# Decomposing CBN into CBPV

A CBN type translates into a computation type.

$$\begin{aligned}\text{bool} &\longmapsto F(1 + 1) \\ \underline{A} + \underline{B} &\longmapsto F(U\underline{A} + U\underline{B}) \\ \underline{A} \rightarrow \underline{B} &\longmapsto U\underline{A} \rightarrow \underline{B}\end{aligned}$$

A CBN term  $x : \underline{A}, y : \underline{B} \vdash M : \underline{C}$  translates as  $x : U\underline{A}, y : U\underline{B} \vdash^c M : \underline{C}$ .

$$\begin{aligned}x &\longmapsto \text{force } x \\ \text{let } (x \text{ be } M, y \text{ be } M'). N &\longmapsto \text{let } (x \text{ be } \text{thunk } M, y \text{ be } \text{thunk } M'). N \\ \lambda x. M &\longmapsto \lambda x. M \\ M N &\longmapsto M (\text{thunk } N) \\ \text{inl } M &\longmapsto \text{return inl } \text{thunk } M\end{aligned}$$

# Summary

We've seen

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

We've seen

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

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

We've seen

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

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

Moggi's  $TA$  is  $UFA$ .

# 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  $UFA$ .

But

# 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  $UFA$ .

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.

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  $\text{return } V$ , then evaluate  $N[V/x]$ .

$$\boxed{\begin{array}{ccc} M \text{ to } x. N & K & \rightsquigarrow \\ M & \text{to } x. N :: K & \end{array}}$$

$$\boxed{\begin{array}{ccc} \text{return } V & \text{to } x. N :: K & \rightsquigarrow \\ N[V/x] & K & \end{array}}$$

# Transitions for application

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

$$\boxed{\begin{array}{ccc} V'M & K & \rightsquigarrow \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{ccc} \lambda x.N & V :: K & \rightsquigarrow \\ N[V/x] & K & \end{array}}$$

# Those function rules again

$$\boxed{\begin{array}{ccc} V \cdot M & K & \rightsquigarrow \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{ccc} \lambda x.N & V :: K & \rightsquigarrow \\ N[V/x] & K & \end{array}}$$

# Those function rules again

$$\boxed{\begin{array}{ccc} V' M & K & \rightsquigarrow \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{ccc} \lambda x. N & V :: K & \rightsquigarrow \\ N[V/x] & K & \end{array}}$$

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

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

# Those function rules again

$$\boxed{\begin{array}{ccc} V' M & K & \rightsquigarrow \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{ccc} \lambda x. N & V :: K & \rightsquigarrow \\ N[V/x] & K & \end{array}}$$

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

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

Revisiting some equations:

$$\begin{aligned} V' \lambda x. M &= M[V/x] \\ M &= \lambda x. x' M && (\text{x fresh}) \\ \text{error } e &= \lambda x. \text{error } e \\ \text{print } c. \lambda x. M &= \lambda x. \text{print } c. M \end{aligned}$$

# Values and Computations

A value **is**, a computation **does**.

- A value of type  $UB$  **is** a thunk of a computation of type  $\underline{B}$ .
- A value of type  $A + A'$  **is** a tagged value  $\text{inl } V$  or  $\text{inr } V$ .
- A value of type  $A \times A'$  **is** a pair  $\langle V, V' \rangle$ .
- A computation of type  $FA$  aims to **return** a value of type  $A$ .
- A computation of type  $A \rightarrow \underline{B}$  aims  
to **pop** a value of type  $A$  and then **behave** in  $\underline{B}$ .
- A computation of type  $\underline{B} \amalg \underline{B}'$  aims  
to **pop** the tag  $\text{l}$  and then **behave** in  $\underline{B}$   
or **pop** the tag  $\text{r}$  and then **behave** in  $\underline{B}'$ .

# What's in a stack?

A stack consists of

- **arguments** that are values
- **arguments** that are tags
- **frames** taking the form `to x. N`.

## Example program of type $F \text{ nat}$ (with complex values)

```
print "hello0".
let (x be 3,
    y be thunk (
        print "hello1".
        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  $\Gamma \vdash^c P : \underline{C}$

$\Gamma$	$P$	$\underline{C}$	nil	$\underline{C}$
----------	-----	-----------------	-----	-----------------

Transitions

$\Gamma$	$M \text{ to } x. N$	$\underline{B}$	$K$	$\underline{C}$	$\rightsquigarrow$
$\Gamma$	$M$	$FA$	$\text{to } x. N :: K$	$\underline{C}$	

$\Gamma$	$\text{return } V$	$FA$	$\text{to } x. N :: K$	$\underline{C}$	$\rightsquigarrow$
$\Gamma$	$N[V/x]$	$\underline{B}$	$K$	$\underline{C}$	

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

# Typing the CK-machine

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

$\Gamma$	$P$	$\underline{C}$	nil	$\underline{C}$
----------	-----	-----------------	-----	-----------------

Transitions

$\Gamma$	$M \text{ to } x. N$	$\underline{B}$	$K$	$\underline{C}$	$\rightsquigarrow$
$\Gamma$	$M$	$FA$	$\text{to } x. N :: K$	$\underline{C}$	

$\Gamma$	$\text{return } V$	$FA$	$\text{to } x. N :: K$	$\underline{C}$	$\rightsquigarrow$
$\Gamma$	$N[V/x]$	$\underline{B}$	$K$	$\underline{C}$	

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

We write  $\Gamma \vdash^k K : \underline{B} \implies \underline{C}$  to mean that  $K$  can accompany a computation of type  $\underline{B}$  during evaluation.

# Typing rules, read off from the CK-machine

## Typing a stack

$$\frac{}{\Gamma \vdash^k \mathbf{nil} : \underline{C} \Longrightarrow \underline{C}}$$

$$\frac{\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^k \mathbf{!} :: K : \underline{B} \amalg \underline{B}' \Longrightarrow \underline{C}}$$

$$\frac{\Gamma, \mathbf{x} : A \vdash^c M : \underline{B} \quad \Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^k \mathbf{to x.} M :: K : FA \Longrightarrow \underline{C}}$$

$$\frac{\Gamma \vdash^v V : A \quad \Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^k V :: K : A \rightarrow \underline{B} \Longrightarrow \underline{C}}$$

# Typing rules, read off from the CK-machine

## Typing a stack

$$\frac{}{\Gamma \vdash^k \mathbf{nil} : \underline{C} \Longrightarrow \underline{C}}$$
$$\frac{\Gamma, \mathbf{x} : A \vdash^c M : \underline{B} \quad \Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^k \mathbf{to x. } M :: K : FA \Longrightarrow \underline{C}}$$
$$\frac{\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^k \mathbf{!} :: K : \underline{B} \amalg \underline{B}' \Longrightarrow \underline{C}}$$
$$\frac{\Gamma \vdash^v V : A \quad \Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^k V :: K : A \rightarrow \underline{B} \Longrightarrow \underline{C}}$$

## Typing a CK-configuration

$$\frac{\Gamma \vdash^c M : \underline{B} \quad \Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\text{ck}} (M, K) : \underline{C}}$$

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

# Operations on Stacks

- 1 Given a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$ , we can **weaken** it or **substitute** values.
- 2 A stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$  can be **dismantled** onto a computation  $\Gamma \vdash^c M : \underline{B}$ , giving a computation  $\Gamma \vdash^c M \bullet K : \underline{C}$ .

# Operations on Stacks

- 1 Given a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$ , we can **weaken** it or **substitute** values.
- 2 A stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$  can be **dismantled** onto a computation  $\Gamma \vdash^c M : \underline{B}$ , giving a computation  $\Gamma \vdash^c M \bullet K : \underline{C}$ .
- 3 Stacks  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$  and  $\Gamma \vdash^k L : \underline{C} \Longrightarrow \underline{D}$  can be **concatenated** to give  $\Gamma \vdash^k K \# L : \underline{B} \Longrightarrow \underline{D}$ .

# Stacks denote homomorphisms

Consider a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$

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

What should  $K$  denote?

# Stacks denote homomorphisms

Consider a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$

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

What should  $K$  denote?

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

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

# Stacks denote homomorphisms

Consider a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$

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

What should  $K$  denote?

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

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

This function should be homomorphic in its second argument:

$$\begin{aligned}\llbracket K \rrbracket(\rho, c) &= c' \\ \llbracket K \rrbracket(\rho, b) &= b'\end{aligned}$$

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

# Stacks denote homomorphisms

Consider a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$

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

What should  $K$  denote?

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

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

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  $M \bullet K$ .

**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  $\Gamma \vdash^{\text{ck}} (M, K) : \underline{C}$  denote?

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

$$\begin{aligned} \llbracket (M, K) \rrbracket & : \llbracket \Gamma \rrbracket \longrightarrow \llbracket \underline{C} \rrbracket \\ \rho & \longmapsto \llbracket K \rrbracket(\rho, \llbracket M \rrbracket \rho) \end{aligned}$$

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  $\llbracket (\text{error BANG}, K) \rrbracket \rho = b'$ .

# 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).

$$\mathbf{Set} \begin{array}{c} \xrightarrow{F^E} \\ \perp \\ \xleftarrow{U^E} \end{array} E/\mathbf{Set}$$

This resolves the exception monad  $X \mapsto X + E$  on  $\mathbf{Set}$ .

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

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

$$\frac{\Gamma \vdash^v V : \text{nat} \quad \Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^c l := V. M : \underline{B}}$$

$$\frac{\Gamma, x : \text{nat} \vdash^c M : \underline{B}}{\Gamma \vdash^c \text{read } l \text{ as } x. M : \underline{B}}$$

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

$$\frac{\Gamma \vdash^v V : \text{nat} \quad \Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^c l := V. M : \underline{B}}$$

$$\frac{\Gamma, x : \text{nat} \vdash^c M : \underline{B}}{\Gamma \vdash^c \text{read } l \text{ as } x. M : \underline{B}}$$

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

The set of states is  $S \cong \mathbb{N} \times \mathbb{B}$ .

# Big-step semantics for state

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

$$\frac{\Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^{\text{sc}} (s, M) : \underline{B}} \quad s \in S$$

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

- Algebra semantics.
- Intrinsic semantics.

# Algebra semantics for state (briefly)

Moggi's monad for state is  $S \rightarrow (S \times -)$ .

Its Eilenberg-Moore algebras were characterized by Plotkin and Power.

# Algebra semantics for state (briefly)

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

# Algebra semantics for state (briefly)

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

We complete the story with an adequacy theorem:

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

This requires an SC-configuration to have a denotation.

# Intrinsic semantics of state

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

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

i.e. a closed configuration  $\vdash^{\text{sc}} (s, M) : \underline{B}$  denotes an element of  $\underline{B}$ .

# Intrinsic semantics of state

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

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

i.e. a closed configuration  $\vdash^{\text{sc}} (s, M) : \underline{B}$  denotes an element of  $\llbracket \underline{B} \rrbracket$ .

Clearly we want

- a value  $\Gamma \vdash^v V : A$  to denote a function  $\llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket$
- a configuration  $\Gamma \vdash^{\text{sc}} (s, M) : \underline{B}$  to denote a function  $\llbracket \Gamma \rrbracket \longrightarrow \llbracket \underline{B} \rrbracket$ .

# Intrinsic semantics of state

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

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

i.e. a closed configuration  $\vdash^{\text{sc}} (s, M) : \underline{B}$  denotes an element of  $\llbracket \underline{B} \rrbracket$ .

Clearly we want

- a value  $\Gamma \vdash^v V : A$  to denote a function  $\llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket$
- a configuration  $\Gamma \vdash^{\text{sc}} (s, M) : \underline{B}$  to denote a function  $\llbracket \Gamma \rrbracket \longrightarrow \llbracket \underline{B} \rrbracket$ .

What about computations?

# Intrinsic semantics of state

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

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

i.e. a closed configuration  $\vdash^{\text{sc}} (s, M) : \underline{B}$  denotes an element of  $\llbracket \underline{B} \rrbracket$ .

Clearly we want

- a value  $\Gamma \vdash^{\text{v}} V : A$  to denote a function  $\llbracket \Gamma \rrbracket \longrightarrow \llbracket A \rrbracket$
- a configuration  $\Gamma \vdash^{\text{sc}} (s, M) : \underline{B}$  to denote a function  $\llbracket \Gamma \rrbracket \longrightarrow \llbracket \underline{B} \rrbracket$ .

What about computations?

A closed computation  $\vdash^{\text{c}} M : \underline{B}$

when supplied with a state  $s$

yields an SC-configuration  $(s, M) : \underline{B}$ .

So  $\Gamma \vdash^{\text{c}} M : \underline{B}$  denotes a function  $\llbracket \Gamma \rrbracket \times S \longrightarrow \llbracket \underline{B} \rrbracket$ .

- A value  $\Gamma \vdash^v V : U\underline{B}$   
when supplied with a state  $s$ ,  
yields a configuration  $s$ , force  $V$ .

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} S \rightarrow \llbracket \underline{B} \rrbracket$$

- A value  $\Gamma \vdash^v V : U\underline{B}$   
when supplied with a state  $s$ ,  
yields a configuration  $s$ , force  $V$ .

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} S \rightarrow \llbracket \underline{B} \rrbracket$$

- A closed SC-configuration of type  $F A$  evaluates to  $s$ , return  $V$ .

$$\llbracket F A \rrbracket \stackrel{\text{def}}{=} S \times \llbracket A \rrbracket$$

- A value  $\Gamma \vdash^v V : U\underline{B}$

when supplied with a state  $s$ ,  
yields a configuration  $s$ , force  $V$ .

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} S \rightarrow \llbracket \underline{B} \rrbracket$$

- A closed SC-configuration of type  $FA$  evaluates to  $s$ , return  $V$ .

$$\llbracket FA \rrbracket \stackrel{\text{def}}{=} S \times \llbracket A \rrbracket$$

- A closed SC-configuration  $(s, M) : A \rightarrow \underline{B}$

when supplied with a closed value  $V : A$   
yields a configuration  $(s, MV) : \underline{B}$ .

$$\llbracket A \rightarrow \underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \rightarrow \llbracket \underline{B} \rrbracket$$

## Semantics of $U$ and $F$ and $\rightarrow$ and $\Pi$

- A value  $\Gamma \vdash^v V : U\underline{B}$   
when supplied with a state  $s$ ,  
yields a configuration  $s$ , force  $V$ .

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} S \rightarrow \llbracket \underline{B} \rrbracket$$

- A closed SC-configuration of type  $FA$  evaluates to  $s$ , return  $V$ .

$$\llbracket FA \rrbracket \stackrel{\text{def}}{=} S \times \llbracket A \rrbracket$$

- A closed SC-configuration  $(s, M) : A \rightarrow \underline{B}$   
when supplied with a closed value  $V : A$   
yields a configuration  $(s, MV) : \underline{B}$ .

$$\llbracket A \rightarrow \underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \rightarrow \llbracket \underline{B} \rrbracket$$

- A closed SC-configuration  $(s, M) : \underline{B} \Pi \underline{B}'$   
yields configurations  $(s, M^l) : \underline{B}$  and  $(s, M^r) : \underline{B}'$ .

$$\llbracket \underline{B} \Pi \underline{B}' \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{B} \rrbracket \times \llbracket \underline{B}' \rrbracket$$

A closed stack  $K : \underline{B} \Longrightarrow \underline{C}$

when supplied with a closed SC-configuration  $(s, M) : \underline{B}$   
yields an SC-configuration  $(s, M \bullet K) : \underline{C}$ .

So a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$  denotes a function  $[[\Gamma]] \times [[\underline{B}]] \longrightarrow \underline{C}$ .

# Semantics of stacks

A closed stack  $K : \underline{B} \Longrightarrow \underline{C}$

when supplied with a closed SC-configuration  $(s, M) : \underline{B}$   
yields an SC-configuration  $(s, M \bullet K) : \underline{C}$ .

So a stack  $\Gamma \vdash^k K : \underline{B} \Longrightarrow \underline{C}$  denotes a function  $[[\Gamma]] \times [[\underline{B}]] \longrightarrow \underline{C}$ .

So we have an adjunction between values and stacks:

$$\mathbf{Set} \begin{array}{c} \xrightarrow{S \times -} \\ \xleftarrow[S \rightarrow -]{\perp} \\ \end{array} \mathbf{Set}$$

# State in call-by-value and call-by-name

For call-by-value we recover the standard semantics:

$$\begin{aligned}\llbracket \text{bool}_{\text{CBV}} \rrbracket &= 1 + 1 \\ \llbracket A \rightarrow_{\text{CBV}} B \rrbracket &= \llbracket U(A \rightarrow FB) \rrbracket \\ &= S \rightarrow (\llbracket A \rrbracket \rightarrow (S \times \llbracket B \rrbracket))\end{aligned}$$

# State in call-by-value and call-by-name

For call-by-value we recover the standard semantics:

$$\begin{aligned}\llbracket \text{bool}_{\text{CBV}} \rrbracket &= 1 + 1 \\ \llbracket A \rightarrow_{\text{CBV}} B \rrbracket &= \llbracket U(A \rightarrow FB) \rrbracket \\ &= S \rightarrow (\llbracket A \rrbracket \rightarrow (S \times \llbracket B \rrbracket))\end{aligned}$$

For call-by-name we recover O'Hearn's semantics:

$$\begin{aligned}\llbracket \text{bool}_{\text{CBN}} \rrbracket &= \llbracket F(1 + 1) \rrbracket \\ &= S \times (1 + 1) \\ \llbracket \underline{A} \rightarrow_{\text{CBN}} \underline{B} \rrbracket &= \llbracket U \underline{A} \rightarrow \underline{B} \rrbracket \\ &= (S \rightarrow \llbracket \underline{A} \rrbracket) \rightarrow \llbracket \underline{B} \rrbracket\end{aligned}$$

# Naming and changing the current stack

Extend the language with two instructions:

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

# Naming and changing the current stack

Extend the language with two instructions:

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

Execution takes places in a bigger language.

$$\begin{array}{l} \Gamma \text{ letstk } \alpha. M \quad \underline{B} \quad K \quad \underline{C} \mid \Delta \quad \rightsquigarrow \\ \Gamma M[K/\alpha] \quad \underline{B} \quad K \quad \underline{C} \mid \Delta \end{array}$$

$$\begin{array}{l} \Gamma \text{ changestk } K. M \quad \underline{B}' \quad L \quad \underline{C} \mid \Delta \quad \rightsquigarrow \\ \Gamma M \quad \underline{B} \quad K \quad \underline{C} \mid \Delta \end{array}$$

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

# Typing with and without `nil`

As well as the **old judgements**  
for values, computations, stacks and CK-configurations,  
we'll have corresponding **new judgements**.

# Typing with and without `nil`

As well as the [old judgements](#)  
for values, computations, stacks and CK-configurations,  
we'll have corresponding [new judgements](#).

	Nil-free $\vdash_x$		Nil-allowing $\vdash_\checkmark$
Values	$\Gamma \vdash_x^v V : A$	implies	$\Gamma \vdash_\checkmark^v V : A [\underline{C}]$
Computations	$\Gamma \vdash_x^c M : \underline{B}$	implies	$\Gamma \vdash_\checkmark^c M : \underline{B} [\underline{C}]$
Stacks	$\Gamma \vdash_x^k K : \underline{B} \Rightarrow$	implies	$\Gamma \vdash_\checkmark^k K : \underline{B} \Rightarrow \underline{C}$
CK-configurations	$\Gamma \vdash_x^{ck} (M, K)$	implies	$\Gamma \vdash_\checkmark^{ck} (M, K) : \underline{C}$

## Notes

- $\underline{C}$  is the type of `nil`.
- More on  $\Gamma$  later.

# Continuations and processes

- A **continuation from  $\underline{B}$**  is a nil-free stack  $\Gamma \vdash_{\times}^v K : \underline{B} \Rightarrow$ .
- A **process** is a nil-free CK-configuration  $\Gamma \vdash_{\times}^{\text{ck}} (M, K)$ .

# Continuations and processes

- A **continuation from  $\underline{B}$**  is a nil-free stack  $\Gamma \vdash_{\times}^v K : \underline{B} \Rightarrow$ .
- A **process** is a nil-free CK-configuration  $\Gamma \vdash_{\times}^{\text{ck}} (M, K)$ .

A typing context  $\Gamma$  is a list of declarations of

- **ordinary** identifiers  $x : A$
- and **continuation** identifiers  $\alpha : \underline{B} \Rightarrow$ .

# Typing rules for control

$$\frac{}{\Gamma \vdash_x^k \alpha : \underline{B} \Rightarrow} (\alpha : \underline{B} \Rightarrow) \in \Gamma$$

$$\frac{\Gamma \vdash_x^k K : \underline{B} \Rightarrow \quad \Gamma \vdash_x^c M : \underline{B}}{\Gamma \vdash_x^c \text{changestk } K. M : \underline{B}'}$$

$$\frac{\Gamma, \alpha : \underline{B} \Rightarrow \vdash_x^c M : \underline{B}}{\Gamma \vdash_x^c \text{letstk } \alpha. M : \underline{B}}$$

# Typing rules for control

$$\frac{}{\Gamma \vdash_{\times}^k \alpha : \underline{B} \Rightarrow} (\alpha : \underline{B} \Rightarrow) \in \Gamma$$

$$\frac{\Gamma \vdash_{\times}^k K : \underline{B} \Rightarrow \quad \Gamma \vdash_{\times}^c M : \underline{B}}{\Gamma \vdash_{\times}^c \text{changestk } K. M : \underline{B}'}$$

$$\frac{\Gamma, \alpha : \underline{B} \Rightarrow \vdash_{\times}^c M : \underline{B}}{\Gamma \vdash_{\times}^c \text{letstk } \alpha. M : \underline{B}}$$

Other than the `nil` rule, each typing rule has a  $\vdash_{\times}$  and a  $\vdash_{\checkmark}$  version.

## Typing rules for control

$$\frac{}{\Gamma \vdash_x^k \alpha : \underline{B} \Rightarrow} (\alpha : \underline{B} \Rightarrow) \in \Gamma$$

$$\frac{\Gamma \vdash_x^k K : \underline{B} \Rightarrow \quad \Gamma \vdash_x^c M : \underline{B}}{\Gamma \vdash_x^c \text{changestk } K. M : \underline{B}'}$$

$$\frac{\Gamma, \alpha : \underline{B} \Rightarrow \vdash_x^c M : \underline{B}}{\Gamma \vdash_x^c \text{letstk } \alpha. M : \underline{B}}$$

Other than the `nil` rule, each typing rule has a  $\vdash_x$  and a  $\vdash_{\checkmark}$  version.

**Theorem:** no closed continuations or processes

If  $\Gamma \vdash_x^v K : \underline{B} \Rightarrow$  or  $\Gamma \vdash_x^{ck} (M, K)$ ,

then  $\Gamma$  includes at least one continuation identifier  $\alpha : \underline{B} \Rightarrow$ .

Fix a set  $R$ , the **semantic domain for processes**.

That means: a (hypothetical) closed process  $\vdash_{\checkmark}^{\text{ck}} (M, K)$  would denote an element of  $R$ .

Fix a set  $R$ , the **semantic domain for processes**.

That means: a (hypothetical) closed process  $\vdash_{\checkmark}^{\text{ck}} (M, K)$  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 processes**.

That means: a (hypothetical) closed process  $\vdash_{\checkmark}^{\text{ck}} (M, K)$  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  $\underline{B}$  denotes an Eilenberg-Moore algebra  $\llbracket \underline{B} \rrbracket_{\text{alg}}$ , a semantic domain for computations.

## Computation type $\underline{B}$

$\llbracket \underline{B} \rrbracket$  will be a **semantic domain for continuations** from  $\underline{B}$ .

That means: a (hypothetical) closed continuation  $\vdash_{\checkmark}^k K : \underline{B} \Rightarrow$  would denote an element of  $\llbracket \underline{B} \rrbracket$ .

## Computation type $\underline{B}$

$\llbracket \underline{B} \rrbracket$  will be a **semantic domain for continuations** from  $\underline{B}$ .

That means: a (hypothetical) closed continuation  $\vdash_{\checkmark}^k K : \underline{B} \Rightarrow$  would denote an element of  $\llbracket \underline{B} \rrbracket$ .

## Context $\Gamma$

- A **semantic environment**  $\rho$  for  $\Gamma$  provides an element  $\rho_x \in \llbracket A \rrbracket$  for each  $(x : A) \in \Gamma$  and an element  $\rho_\alpha \in \llbracket \underline{B} \rrbracket$  for each  $(\alpha : \underline{B} \Rightarrow) \in \Gamma$ .
- $\llbracket \Gamma \rrbracket$  is the set of all semantic environments for  $\Gamma$ .

$$\llbracket \Gamma \rrbracket \stackrel{\text{def}}{=} \prod \left\{ \begin{array}{ll} (x : A) \in \Gamma & \mapsto \llbracket A \rrbracket \\ (\alpha : \underline{B} \Rightarrow) \in \Gamma & \mapsto \llbracket \underline{B} \rrbracket \end{array} \right.$$

Clearly we want

- a value  $\Gamma \vdash_{\times}^v V : A$  to denote a function  $[[\Gamma]] \longrightarrow [[A]]$
- a continuation  $\Gamma \vdash_{\times}^k K : \underline{B} \Rightarrow$  to denote a function  $[[\Gamma]] \longrightarrow [[\underline{B}]]$
- a process  $\Gamma \vdash_{\times}^{\text{ck}} (M, K)$  to denote a function  $[[\Gamma]] \longrightarrow R$ .

What about computations?

Clearly we want

- a value  $\Gamma \vdash_{\times}^v V : A$  to denote a function  $[[\Gamma]] \longrightarrow [[A]]$
- a continuation  $\Gamma \vdash_{\times}^k K : \underline{B} \Rightarrow$  to denote a function  $[[\Gamma]] \longrightarrow [[\underline{B}]]$
- a process  $\Gamma \vdash_{\times}^{\text{ck}} (M, K)$  to denote a function  $[[\Gamma]] \longrightarrow R$ .

What about computations?

A closed computation  $M : \underline{B}$

when supplied with a closed continuation  $K : \underline{B} \Rightarrow$

yields a process  $(M, K)$ .

# Semantics of the nil-free judgements

Clearly we want

- a value  $\Gamma \vdash_{\times}^v V : A$  to denote a function  $[[\Gamma]] \longrightarrow [[A]]$
- a continuation  $\Gamma \vdash_{\times}^k K : \underline{B} \Rightarrow$  to denote a function  $[[\Gamma]] \longrightarrow [[\underline{B}]]$
- a process  $\Gamma \vdash_{\times}^{\text{ck}} (M, K)$  to denote a function  $[[\Gamma]] \longrightarrow R$ .

What about computations?

A closed computation  $M : \underline{B}$

when supplied with a closed continuation  $K : \underline{B} \Rightarrow$

yields a process  $(M, K)$ .

So  $\Gamma \vdash_{\times}^c M : \underline{B}$  denotes a function  $[[\Gamma]] \times [[\underline{B}]] \longrightarrow R$ .

- A closed value  $V$  of type  $U\underline{B}$  when supplied with a closed continuation  $K : \underline{B} \Rightarrow$  yields a process ( $\text{force } V, K$ ).

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{B} \rrbracket \rightarrow R$$

.

- A closed value  $V$  of type  $U\underline{B}$   
when supplied with a closed continuation  $K : \underline{B} \Rightarrow$   
yields a process (`force`  $V, K$ ).

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{B} \rrbracket \rightarrow R$$

- A closed continuation  $K : FA \Rightarrow$   
when supplied with a closed value  $V : A$   
yields a process (`return`  $V, K$ ).

$$\llbracket FA \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \rightarrow R$$

- A closed value  $V$  of type  $U\underline{B}$   
when supplied with a closed continuation  $K : \underline{B} \Rightarrow$   
yields a process (`force`  $V, K$ ).

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{B} \rrbracket \rightarrow R$$

- A closed continuation  $K : FA \Rightarrow$   
when supplied with a closed value  $V : A$   
yields a process (`return`  $V, K$ ).

$$\llbracket FA \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \rightarrow R$$

- A closed continuation from  $A \rightarrow \underline{B}$  is a pair  $V :: K$ .

$$\llbracket A \rightarrow \underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \times \llbracket \underline{B} \rrbracket$$

## Semantics of $U$ and $F$ and $\rightarrow$ and $\Pi$

- A closed value  $V$  of type  $U\underline{B}$  when supplied with a closed continuation  $K : \underline{B} \Rightarrow$  yields a process (`force`  $V, K$ ).

$$\llbracket U\underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{B} \rrbracket \rightarrow R$$

- A closed continuation  $K : FA \Rightarrow$  when supplied with a closed value  $V : A$  yields a process (`return`  $V, K$ ).

$$\llbracket FA \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \rightarrow R$$

- A closed continuation from  $A \rightarrow \underline{B}$  is a pair  $V :: K$ .

$$\llbracket A \rightarrow \underline{B} \rrbracket \stackrel{\text{def}}{=} \llbracket A \rrbracket \times \llbracket \underline{B} \rrbracket$$

- A closed continuation from  $\underline{B} \Pi \underline{B}'$  is either  $^1 :: K$  or  $^r :: K$ .

$$\llbracket \underline{B} \Pi \underline{B}' \rrbracket \stackrel{\text{def}}{=} \llbracket \underline{B} \rrbracket + \llbracket \underline{B}' \rrbracket$$

# Semantics of the nil-allowing judgements

A nil-allowing expression  $\Gamma \vdash_{\checkmark} t [\underline{C}]$

(value, computation, stack or CK-configuration)

when supplied with a closed continuation  $L : \underline{C} \Rightarrow$

yields a nil-free expression  $\Gamma \vdash_{\times} t[L/\mathbf{nil}]$ .

# Semantics of the nil-allowing judgements

A nil-allowing expression  $\Gamma \vdash_{\checkmark} t [\underline{C}]$   
(value, computation, stack or CK-configuration)  
when supplied with a closed continuation  $L : \underline{C} \Rightarrow$   
yields a nil-free expression  $\Gamma \vdash_{\times} t[L/\text{nil}]$ .

We call  $L$  the **top-level continuation**.

# Semantics of the nil-allowing judgements

A nil-allowing expression  $\Gamma \vdash_{\checkmark} t \ [\underline{C}]$   
(value, computation, stack or CK-configuration)  
when supplied with a closed continuation  $L : \underline{C} \Rightarrow$   
yields a nil-free expression  $\Gamma \vdash_{\times} t[L/\text{nil}]$ .

We call  $L$  the **top-level continuation**.

- A value  $\Gamma \vdash_{\checkmark}^v V : A \ [\underline{C}]$  denotes a function  $[\Gamma] \times [\underline{C}] \longrightarrow [A]$ .
- A computation  $\Gamma \vdash_{\checkmark}^c M : \underline{B} \ [\underline{C}]$  denotes a function  $[\Gamma] \times [\underline{C}] \times [\underline{B}] \longrightarrow R$ .
- A stack  $\Gamma \vdash_{\checkmark}^k K : \underline{B} \Longrightarrow \underline{C}$  denotes a function  $[\Gamma] \times [\underline{C}] \longrightarrow [\underline{B}]$ .
- A CK-configuration  $\Gamma \vdash_{\checkmark}^{\text{ck}} (M, K) : \underline{C}$  denotes a function  $[\Gamma] \times [\underline{C}] \longrightarrow R$ .

# Semantics of the old judgements

Let's summarize:

	Old judgement	Denotes a function
Value	$\Gamma \vdash^v V : A$	$[[\Gamma]] \longrightarrow [[A]]$
Computation	$\Gamma \vdash^c M : \underline{B}$	$[[\Gamma]] \times [[\underline{B}]] \longrightarrow R$
Stack	$\Gamma \vdash^k K : \underline{B} \Rightarrow \underline{C}$	$[[\Gamma]] \times [[\underline{C}]] \longrightarrow [[\underline{B}]]$
CK-configuration	$\Gamma \vdash^{ck} (M, K) : \underline{C}$	$[[\Gamma]] \times [[\underline{C}]] \longrightarrow R$

# Semantics of the old judgements

Let's summarize:

	Old judgement	Denotes a function
Value	$\Gamma \vdash^v V : A$	$[[\Gamma]] \longrightarrow [[A]]$
Computation	$\Gamma \vdash^c M : \underline{B}$	$[[\Gamma]] \times [[\underline{B}]] \longrightarrow R$
Stack	$\Gamma \vdash^k K : \underline{B} \Rightarrow \underline{C}$	$[[\Gamma]] \times [[\underline{C}]] \longrightarrow [[\underline{B}]]$
CK-configuration	$\Gamma \vdash^{ck} (M, K) : \underline{C}$	$[[\Gamma]] \times [[\underline{C}]] \longrightarrow R$

So we have an adjunction between values and stacks:

$$\mathbf{Set} \begin{array}{c} \xrightarrow{\dashrightarrow R} \\ \perp \\ \xleftarrow{\dashrightarrow R} \end{array} \mathbf{Set}^{\text{op}}$$

# Control in call-by-value and call-by-name

For call-by-value we recover the standard semantics:

$$\begin{aligned}\llbracket \text{bool}_{\text{CBV}} \rrbracket &= 1 + 1 \\ \llbracket A \rightarrow_{\text{CBV}} B \rrbracket &= \llbracket U(A \rightarrow FB) \rrbracket \\ &= (\llbracket A \rrbracket \times (\llbracket B \rrbracket \rightarrow R)) \rightarrow R\end{aligned}$$

# Control in call-by-value and call-by-name

For call-by-value we recover the standard semantics:

$$\begin{aligned}\llbracket \text{bool}_{\text{CBV}} \rrbracket &= 1 + 1 \\ \llbracket A \rightarrow_{\text{CBV}} B \rrbracket &= \llbracket U(A \rightarrow FB) \rrbracket \\ &= (\llbracket A \rrbracket \times (\llbracket B \rrbracket \rightarrow R)) \rightarrow R\end{aligned}$$

For call-by-name we recover Streicher and Reus's semantics:

$$\begin{aligned}\llbracket \text{bool}_{\text{CBN}} \rrbracket &= \llbracket F(1 + 1) \rrbracket \\ &= (1 + 1) \rightarrow R \\ \llbracket \underline{A} \rightarrow_{\text{CBN}} \underline{B} \rrbracket &= \llbracket U \underline{A} \rightarrow \underline{B} \rrbracket \\ &= (\llbracket \underline{A} \rrbracket \rightarrow R) \times \llbracket \underline{B} \rrbracket\end{aligned}$$

# Summary: adjunctions between values and stacks

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

# Summary: adjunctions between values and stacks

For a set  $E$ , the adjunction  $\mathbf{Set} \begin{array}{c} \xrightarrow{F^E} \\ \perp \\ \xleftarrow{U^E} \end{array} E/\mathbf{Set}$

models call-by-push-value with errors.

For a set  $S$ , the adjunction  $\mathbf{Set} \begin{array}{c} \xrightarrow{S \times -} \\ \perp \\ \xleftarrow{S \rightarrow -} \end{array} \mathbf{Set}$

models call-by-push-value with state.

# Summary: adjunctions between values and stacks

For a set  $E$ , the adjunction  $\mathbf{Set} \begin{array}{c} \xrightarrow{F^E} \\ \perp \\ \xleftarrow{U^E} \end{array} E/\mathbf{Set}$

models call-by-push-value with errors.

For a set  $S$ , the adjunction  $\mathbf{Set} \begin{array}{c} \xrightarrow{S \times -} \\ \perp \\ \xleftarrow{S \rightarrow -} \end{array} \mathbf{Set}$

models call-by-push-value with state.

For a set  $R$ , the adjunction  $\mathbf{Set} \begin{array}{c} \xrightarrow{- \rightarrow R} \\ \perp \\ \xleftarrow{- \rightarrow R} \end{array} \mathbf{Set}^{\text{op}}$

models call-by-push-value with control.