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

Paul Blain Levy

University of Birmingham

April 18, 2017

Outline

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

Types

We're going to look at simply typed λ-calculus with arithmetic,

including not just function types, but also sum and product types.

Here is the syntax of types:

$$A \quad ::= \quad \texttt{bool} \mid \texttt{nat} \mid A \to A \mid 1 \mid A \times A \mid 0 \mid A + A$$

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

Types

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

Here is the syntax of types:

$$\begin{array}{rcl} A & ::= & \texttt{bool} \mid \texttt{nat} \mid A \to A \mid 1 \mid A \times A \mid 0 \mid A + A \\ & & \mid \sum_{i \in \mathbb{N}} A_i \mid \prod_{i \in \mathbb{N}} A_i \quad \text{(optional extra)} \end{array}$$

Why no brackets?

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

Typing Judgement

Example

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

In English:

Given declarations of $\mathtt{x} : \mathtt{nat}$ and $\mathtt{y} : \mathtt{nat}$,

$\lambda \mathtt{z}_{\mathtt{nat} \to \mathtt{nat}}.\ \mathtt{z}\,(\mathtt{x} + \mathtt{x})$ is a term of type $(\mathtt{nat} \to \mathtt{nat}) \to \mathtt{nat}$.

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

- Γ is a typing context, a finite set of typed distinct identifiers.
- M is a term.
- A is a type.

Identifiers

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

Free identifier

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

Multiple local declaration, e.g. of two identifiers

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

Typing rules for $A \to B$

Introduction rule

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

Elimination rule

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

Type annotations in terms

- For Γ 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:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

Elimination rule

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

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

Typing rules for arithmetic

These are *ad hoc* rules.

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

Typing rules for $A + B$

Two introduction rules

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

Elimination rule

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

Typing rules for $A + B$

Two introduction rules

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

Elimination rule

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

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

Typing rules for 0

Zero introduction rules

Elimination rule

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

Typing rules for $A \times B$, pattern-match syntax

Introduction rule

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

Elimination rule

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

Typing rules for $A \times B$, projection syntax

Two elimination rules

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

Typing rules for $A \times B$, projection syntax

Two elimination rules

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

Introduction rule

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \lambda\{{}^{\mathtt{l}}.\, M,\ {}^{\mathtt{r}}.\, N\} : A \times B}$$

Typing rules for $A \times B$, projection syntax

Two elimination rules

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

Introduction rule

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \lambda\{{}^{\mathtt{l}}.\, M,\ {}^{\mathtt{r}}.\, N\} : A \times B}$$

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

Typing rules for 1, pattern-match and projection

Introduction rule

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1} \qquad \frac{}{\Gamma \vdash \lambda\{\,\} : 1}$$

Elimination rule for pattern-match syntax

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

Zero elimination rules for projection syntax

Weakening is admissible

Theorem

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

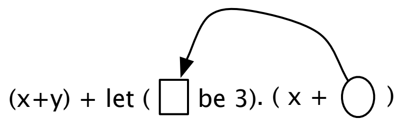
Binding diagrams

Example

The term

$$(\mathtt{x} + \mathtt{y}) + \mathtt{let}\ (\mathtt{y}\ \mathtt{be}\ 3).\ (\mathtt{x} + \mathtt{y})$$

has binding diagram



- Terms are α-equivalent when they have the same binding diagram.

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

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

Substitution

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

Substitution

Subsitution 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 $\Gamma, \mathrm{x} : A, \mathrm{y} : B \vdash N : C$,

we define $\Gamma \vdash N[M/\mathrm{x}, M'/\mathrm{y}] : C$.

Example

$$\begin{array}{rcl} M & = & \lambda \mathrm{y}_{\mathtt{nat}}.\ \mathrm{y} + 3 \\ M' & = & 7 \\ N & = & \mathrm{x}\,(5 + \mathrm{y}) \\ N[M/\mathrm{x}, M'/\mathrm{y}] & = & (\lambda \mathrm{z}_{\mathtt{nat}}.\ \mathrm{z} + 3)\,(5 + 7) \end{array}$$

Types denote sets

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

Types denote sets

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

Types denote sets

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

Semantics of types

Notation

For sets X and Y,

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

$$
\begin{array}{rcl}
[\![\texttt{bool}]\!] & = & \mathbb{B} = \{\mathsf{true}, \mathsf{false}\} \\
[\![\texttt{nat}]\!] & = & \mathbb{N} \\
[\![A \to B]\!] & = & [\![A]\!] \to [\![B]\!] \\
[\![1]\!] & = & 1 = \{\langle\,\rangle\} \\
[\![A + B]\!] & = & [\![A]\!] + [\![B]\!] \\
[\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \\
[\![0]\!] & = & \emptyset
\end{array}
$$

Semantic environments

Let Γ be a typing context.

- A semantic environment ρ for Γ provides an element $\rho_{\mathtt{x}} \in [\![A]\!]$ for each $(\mathtt{x} : A) \in \Gamma$.
- $[\![\Gamma]\!]$ is the set of semantic environments for Γ.

$$[\![\Gamma]\!] \stackrel{\text{def}}{=} \prod_{(\mathtt{x}:A)\in\Gamma} [\![A]\!]$$

Semantics of typing judgement

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

we shall define $[\![M]\!]$, or more precisely $[\![\Gamma \vdash M : A]\!]$.

It's a function from $[\![\Gamma]\!]$ to $[\![A]\!]$.

Example

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

denotes the function

$$\begin{array}{rcl} [\![\mathtt{x} : \mathtt{nat}, \mathtt{y} : \mathtt{nat}]\!] & \longrightarrow & (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ \rho & \longmapsto & \lambda z \in \mathbb{N} \to \mathbb{N}.\, z(\rho_{\mathtt{x}} + \rho_{\mathtt{y}}) \end{array}$$

Semantics of terms

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

$$[\![17]\!] \ : \ \rho \longmapsto 17$$

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

$$[\![M + M']\!] \ : \ \rho \longmapsto [\![M]\!]\rho + [\![M']\!]\rho$$

More semantic equations

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

$$[\![\mathtt{x}]\!]\ :\ \rho \longmapsto \rho_{\mathtt{x}}$$

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

$$[\![\lambda \mathtt{x}_A .\, M]\!]\ :\ \rho \longmapsto \lambda a \in [\![A]\!] .\, [\![M]\!](\rho, \mathtt{x} \mapsto a)$$

More semantic equations

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

$$[\![\texttt{inl}^{A,B}\ M]\!]\ :\ \rho \longmapsto \mathsf{inl}\ [\![M]\!]\rho$$

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

$$[\![\texttt{match}\ M\ \texttt{as}\ \{\texttt{inl x}.\ N, \texttt{inr y}.\ N'\}]\!]\ :\ \rho \longmapsto$$
$$\mathsf{match}\ [\![M]\!]\rho\ \mathsf{as}\ \{\mathsf{inl}\ a.\ [\![N]\!](\rho, \texttt{x} \mapsto a), \mathsf{inr}\ b.\ [\![N']\!](\rho, \texttt{x} \mapsto b)\}$$

Basic properties

Semantic Coherence

If type annotations are omitted,

then $\Gamma \vdash M : A$ can have more than one derivation.

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

Basic properties

Semantic Coherence

If type annotations are omitted,

then $\Gamma \vdash M : A$ can have more than one derivation.

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

Weakening Lemma

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

$$[\![\Gamma' \vdash M : A]\!]\rho = [\![\Gamma \vdash M]\!](\rho \upharpoonright_{\Gamma})$$

Substitution

Binding Diagrams

- We can give denotational semantics of binding diagrams.
- $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash \mathsf{BD}(M) : A]\!]$
- So α-equivalent terms have the same denotation.

Substitution

Binding Diagrams

- We can give denotational semantics of binding diagrams.
- $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash \mathsf{BD}(M) : A]\!]$
- So α-equivalent terms have the same denotation.

Substitution Lemma

For binding diagrams $\Gamma \vdash M : A$ and $\Gamma \vdash M' : B$ and $\Gamma, \mathtt{x} : A \vdash N : C$, we can recover $[\![N[M/\mathtt{x}, M'/\mathtt{y}]]\!]$ from $[\![M]\!]$ and $[\![N]\!]$.

$$[\![N[M/\mathtt{x}, M'/\mathtt{y}]]\!] \;:\; \rho \longmapsto [\![N]\!](\rho, \mathtt{x} \mapsto [\![M]\!]\rho, \mathtt{y} \mapsto [\![M']\!]\rho)$$

β-laws

The β-law for $A \to B$

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

Introduction inside an elimination may be removed.

β-laws

The β-law for $A \to B$

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

Introduction inside an elimination may be removed.

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

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A'}{\Gamma \vdash \lambda\{^{\mathtt{l}} .\, M,\ ^{\mathtt{r}} .\, N\}^{\mathtt{l}} = M : A}$$

Zero β-laws for projection unit 1

More β-laws

Two β-laws for `bool`

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

More β-laws

Two β-laws for `bool`

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

Two β-laws for $A + B$

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

More β-laws

Two β-laws for `bool`

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

Two β-laws for $A + B$

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

Zero β-laws for 0

β-law for local declaration

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

η-laws

η-law for $A \to B$, everything is λ

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash M = \lambda \mathtt{x}_A .\, M\, \mathtt{x} : A \to B}\ \mathtt{x} \notin \Gamma$$

Introduction outside an elimination may be inserted.

η-laws

η-law for $A \to B$, everything is λ

$$\frac{\Gamma \vdash M : A \to B}{\Gamma \vdash M = \lambda \mathtt{x}_A .\, M\, \mathtt{x} : A \to B}\ \mathtt{x} \notin \Gamma$$

Introduction outside an elimination may be inserted.

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

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash M = \lambda\{^{\mathtt{l}}.\, M^{\mathtt{l}},\ ^{\mathtt{r}}.\, M^{\mathtt{r}}\} : A \times B}$$

η-law for projection unit 1, everything is λ

$$\frac{\Gamma \vdash M : 1}{\Gamma \vdash M = \lambda\{\,\} : 1}$$

More η-laws

η-law for `bool`, everything is `true` or `false`

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

More η-laws

η-law for `bool`, everything is `true` or `false`

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

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

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

More η-laws

η-law for `bool`, everything is `true` or `false`

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

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

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

η-law for 0, nothing exists

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

The $\beta\eta$-theory

We define $\Gamma \vdash M =_{\beta\eta} M' : A$ inductively as follows.

All the β- and η-laws are taken as axioms,

and it is a congruence i.e. an equivalence relation preserved by each term constructor. For example:

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

Properties of $=_{\beta\eta}$

Closure Theorems

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

$$\begin{array}{rcl} \mathtt{z} : 0 & \vdash & \mathtt{true} =_{\beta\eta} \mathtt{false} : \mathtt{bool} \\ & \vdash & \mathtt{true} \neq_{\beta\eta} \mathtt{false} : \mathtt{bool} \end{array}$$

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

Soundness theorem

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

Follows from the weakening and substitution lemmas.

Reversible rule for $A \to B$

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

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

natural in Γ—we'll skate over naturality.

Reversible rule for $A \to B$

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

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

natural in Γ—we'll skate over naturality.

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

Reversible rule for $A \to B$

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

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

natural in Γ—we'll skate over naturality.

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

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

Reversible rule for `bool`

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

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

natural in Γ and C—we'll skate over naturality.

- Downwards, a pair $\Gamma \vdash M : C$ and $\Gamma \vdash M' : C$ is sent to $\mathtt{match\ z\ as}\ \{\mathtt{true}.\,M,\ \mathtt{false}.\,M'\}$.
- Upwards, a term $\Gamma, \mathtt{z} : \mathtt{bool} \vdash N : C$ is sent to $N[\mathtt{true}/\mathtt{z}]$ and $N[\mathtt{false}/\mathtt{z}]$.
- 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

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

natural in Γ and C.

Reversible rule for $A + B$

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

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

natural in Γ and C.

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

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

natural in Γ and C.

Bipartisan connectives

The connective × has a reversible rule

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

natural in Γ, so it's rightist.

Bipartisan connectives

The connective × has a reversible rule

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

natural in Γ, so it's rightist.

It also has a reversible rule

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

natural in Γ and C, so it's leftist.

Bipartisan connectives

The connective × has a reversible rule

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

natural in Γ, so it's rightist.

It also has a reversible rule

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

natural in Γ and C, so it's leftist.

Likewise 1 is bipartisan.

Most general leftist connective

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

$$(\llbracket A \rrbracket \times \llbracket A' \rrbracket) + (\llbracket B \rrbracket \times \llbracket B' \rrbracket \times \llbracket B'' \rrbracket)$$

This gives a leftist connective.

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

Most general leftist connective

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

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

This gives a leftist connective.

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

Here is its term syntax:

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

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

$$\mathtt{match}\ M\ \mathtt{as}\ \{\mathtt{in}_0(\mathrm{x}, \mathrm{x}').\, N, \mathtt{in}_1(\mathrm{y}, \mathrm{y}', \mathrm{y}'').\, N'\}$$

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

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

This gives a rightist connective.

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

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

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

This gives a rightist connective.

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

Here is its term syntax:

$$\begin{aligned} \lambda\{^0(\mathrm{x}, \mathrm{x}').M,^1\,(\mathrm{y}, \mathrm{y}', \mathrm{y}'').M'\} \\ M\,^0(N, N') \\ M\,^1(N, N', N'') \end{aligned}$$

Jumbo λ-calculus

Type syntax

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

Term syntax, with type annotations omitted

$$\begin{array}{rcl} M & ::= & \mathtt{x} \mid \mathtt{let}\ (\overrightarrow{\mathtt{x}\ \mathtt{be}\ M}).\ M \\ & & \mid \mathtt{in}_i(\overrightarrow{M}) \\ & & \mid \mathtt{match}\ M\ \mathtt{as}\ \{\mathtt{in}_i(\overrightarrow{\mathtt{x}}).\ M_i\}_{i<n} \\ & & \mid \lambda\{^i(\overrightarrow{\mathtt{x}}).\ M_i\}_{i<n} \\ & & \mid M\,^i(\overrightarrow{M}) \end{array}$$

Jumbo λ-calculus

Type syntax

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

Term syntax, with type annotations omitted

$$\begin{array}{rcl} M & ::= & \mathtt{x} \mid \mathtt{let}\ (\overrightarrow{\mathtt{x}\ \mathsf{be}\ M}).\ M \\ & & \mid \mathtt{in}_i(\overrightarrow{M}) \\ & & \mid \mathtt{match}\ M\ \mathtt{as}\ \{\mathtt{in}_i(\overrightarrow{\mathtt{x}}).\ M_i\}_{i<n} \\ & & \mid \lambda\{^i(\overrightarrow{\mathtt{x}}).\ M_i\}_{i<n} \\ & & \mid M\,^i(\overrightarrow{M}) \end{array}$$

Includes both pattern-match product $(A \times B)$
and projection product $(A \mathbin{\Pi} B)$.

Jumbo vs non-jumbo

Jumbo λ-calculus is the most expressive form of simply typed λ-calculus:

it contains all leftist and rightist connectives as primitives.

Jumbo vs non-jumbo

Jumbo λ-calculus is the most expressive form of simply typed λ-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 λ-calculus is the most expressive form of simply typed λ-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 β- and η-laws are not going to survive.

Evaluating terms

We want to evaluate every closed term $\vdash M : A$ to a terminal term.

We want $\lambda \mathtt{x}_A.\, M$ to be terminal, since M is not closed.

But there are many options.

Three decisions we must make

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

Three decisions we must make

1. To evaluate $\texttt{let}\ (\texttt{x be}\ M,\ \texttt{y be}\ M').\ N$, do we
    - evaluate M to T and M' to T', then evaluate $N[T/\texttt{x}, T'/\texttt{y}]$?
    - just evaluate $N[M/\texttt{x}, M'/\texttt{y}]$?
2. To evaluate $M\ N$, we must evaluate M to $\lambda\texttt{x}_A.\,P$. Do we
    - evaluate N to T (before or after evaluating M), then evaluate $P[T/\texttt{x}]$?
    - just evaluate $P[N/\texttt{x}]$?

Three decisions we must make

1. To evaluate `let` $(\mathtt{x}\ \mathtt{be}\ M,\ \mathtt{y}\ \mathtt{be}\ M').\ N$, do we
   - evaluate M to T and M' to T', then evaluate $N[T/\mathtt{x}, T'/\mathtt{y}]$?
   - just evaluate $N[M/\mathtt{x}, M'/\mathtt{y}]$?
2. To evaluate $M\ N$, we must evaluate M to $\lambda\mathtt{x}_A.\,P$. Do we
   - evaluate N to T (before or after evaluating M), then evaluate $P[T/\mathtt{x}]$?
   - just evaluate $P[N/\mathtt{x}]$?
3. Any terminal term of type $A + B$ must be `inl` M or `inr` M. Do we
   - deem `inl` T and `inr` T terminal only if T is terminal?
   - always deem `inl` M and `inr` M terminal?

One fundamental decision

Do we substitute terminal terms, or unevaluated terms?

One fundamental decision

Do we substitute terminal terms, or unevaluated terms?

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

Substituting unevaluated terms gives call-by-name.

One fundamental decision

Do we substitute terminal terms, or unevaluated terms?

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

Substituting unevaluated terms gives call-by-name.

Terminology: lazy and call-by-name

- "Lazy" evaluation usually means call-by-need, except in Abramsky's "lazy λ-calculus".
- In the untyped literature, "call-by-name" evaluation means reduction to head normal form.

Evaluation order for `let`

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

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

Evaluation order for application

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

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

Terminal terms of type $A + B$

Any terminal term of type $A + B$ must be `inl` M or `inr` M. Do we

- deem `inl` T and `inr` T terminal only if T is terminal? Call-by-value
- always deem `inl` M and `inr` M terminal? Call-by-name

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

Definitional interpreter for call-by-value

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

To evaluate

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

Definitional interpreter for call-by-name

In CBN the terminals are $\texttt{true}, \ \texttt{false}, \texttt{inl}\ M, \ \texttt{inr}\ M, \lambda\texttt{x}.M$
To evaluate

- $\texttt{true}$: return $\texttt{true}$.
- $M + N$: evaluate M. If this returns m, evaluate N. If this returns n, return $m + n$.
- $\lambda\texttt{x}.M$: return $\lambda\texttt{x}.M$.
- $\texttt{inl}\ M$: return $\texttt{inl}\ M$.
- $\texttt{let}\ (\texttt{x be}\ M, \ \texttt{y be}\ M').\ N$: evaluate $N[M/\texttt{x}, M'/\texttt{y}]$.
- $\texttt{match}\ M\ \texttt{as}\ \{\texttt{true}.\ N, \ \texttt{false}.\ N'\}$: evaluate M. If this returns $\texttt{true}$, evaluate N, but if it returns $\texttt{false}$, evaluate N'.
- $\texttt{match}\ M\ \texttt{as}\ \{\texttt{inl x}.\ N, \ \texttt{inr x}.\ N'\}$: evaluate M. If this returns $\texttt{inl}\ P$, evaluate $N[P/\texttt{x}]$, but if it returns $\texttt{inr}\ P$, evaluate $N'[P/\texttt{x}]$.
- $M\,N$: evaluate M. If this returns $\lambda\texttt{x}.P$, evaluate $P[N/\texttt{x}]$.

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

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

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

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

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

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

Big-step semantics for call-by-name

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

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

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

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

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

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

Long story

The experiment

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

Long story

The experiment

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

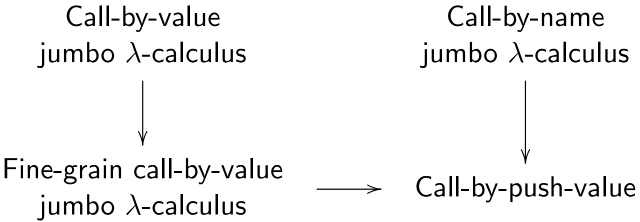
Analyzing CBV with a microscope

- 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



Call-by-value
jumbo λ-calculus

Call-by-name
jumbo λ-calculus

Fine-grain call-by-value
jumbo λ-calculus

Call-by-push-value

Both fine-grain call-by-value and call-by-push-value are obtained empirically, by observing particles of meaning within a range of denotational models.

Where this story comes from

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

Adding computational effects

Errors

Let $E = \{\mathrm{CRASH}, \mathrm{BANG}\}$ be a set of "errors". We add

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

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

Printing

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

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

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

Exercises

1. Evaluate

   $$\texttt{let}\ (\mathrm{x}\ \texttt{be}\ \texttt{error}\ \mathrm{CRASH}).\ 5$$

   in CBV and CBN.
2. Evaluate

   $$(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{x}))(\texttt{print}\ \texttt{"hello"}.\ 4)$$

   in CBV and CBN.
3. Evaluate

   $$\begin{array}{l}\texttt{match}\ (\texttt{print}\ \texttt{"hello"}.\ \texttt{inr}\ \texttt{error}\ \mathrm{CRASH})\ \texttt{as} \\ \quad \{\texttt{inl}\ \mathrm{x}.\ \mathrm{x}+1,\ \texttt{inr}\ \mathrm{y}.\ 5\}\end{array}$$

   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.
- $M \lightning e$ means M raises error e.

Here are the rules for application:

$$\frac{M \lightning e}{M\ N \lightning e} \qquad \frac{M \Downarrow \lambda \mathtt{x}.\, P \quad N \lightning e}{M\ N \lightning e}$$

$$\frac{M \Downarrow \lambda \mathtt{x}.\, P \quad N \Downarrow T \quad P[T/\mathtt{x}] \lightning e}{M\ N \lightning e}$$

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

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 $M \simeq_{\mathbf{CBV}} M'$ and $M \simeq_{\mathbf{CBN}} M'$.

The η-law for boolean type: has it survived?

η-law for `bool`

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

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

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,

$$\mathtt{true} \not\simeq_{\mathbf{CBN}} \mathtt{match}\ \mathtt{z}\ \mathtt{as}\ \{\mathtt{true}.\ \mathtt{true},\ \mathtt{false}.\ \mathtt{true}\}$$

because we can apply the context

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

Similarly the η-law for sum types is valid in CBV but not in CBN.

The η-law for functions: has it survived?

η-law for $A \to B$ and $A \,\Pi\, B$

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

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

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

$$\begin{array}{rcl}
\mathtt{error}\ e & \simeq_{\mathbf{CBN}} & \lambda \mathtt{x}.\ \mathtt{error}\ e \\
\mathtt{error}\ e & \simeq_{\mathbf{CBN}} & \lambda\{^{\mathtt{l}}.\, \mathtt{error}\ e,\ ^{\mathtt{r}}.\, \mathtt{error}\ e\} \\
\mathtt{print}\ c.\ \lambda \mathtt{x}.\ M & \simeq_{\mathbf{CBN}} & \lambda \mathtt{x}.\ \mathtt{print}\ c.\ M \\
\mathtt{print}\ c.\ \lambda\{^{\mathtt{l}}.\, M,\ ^{\mathtt{r}}.\, N\} & \simeq_{\mathbf{CBN}} & \lambda\{^{\mathtt{l}}.\, \mathtt{print}\ c.\ M,\ ^{\mathtt{r}}.\, \mathtt{print}\ c.\ N\}
\end{array}$$

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

In CBN, a function gets evaluated only by being applied.

Summary

The pure λ-calculus satisfies all the β- and η-laws.

With computational effects,

- CBV satisfies η for leftist connectives (tuple types), but not rightist ones (function types)
- CBN satisfies η for rightist connectives (function types), but not leftist ones (tuple types).

Summary

The pure λ-calculus satisfies all the β- and η-laws.

With computational effects,

- CBV satisfies η for leftist connectives (tuple types), but not rightist ones (function types)
- CBN satisfies η for rightist connectives (function types), but not leftist ones (tuple types).

Similarly for isomorphisms:

- $(A + B) + C \cong A + (B + C)$ survives in CBV but not CBN.
- $A \times B \cong A \,\Pi\, B$ survives in neither CBV nor CBN.
- $A \to (B \to C) \cong (A \,\Pi\, B) \to C$ survives in CBN but not CBV.

Naive CBV semantics

Our first attempt.

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

$$\begin{array}{rcl}
[\![\mathtt{bool}]\!]_* & = & \mathbb{B} + E \\
[\![\mathtt{bool} + \mathtt{bool}]\!]_* & = & (\mathbb{B} + \mathbb{B}) + E \\
[\![\mathtt{bool} \times \mathtt{bool}]\!]_* & = & (\mathbb{B} \times \mathbb{B}) + E
\end{array}$$

Naive CBV semantics

Our first attempt.

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

$$\begin{array}{rcl} [\![\mathtt{bool}]\!]_* & = & \mathbb{B} + E \\ [\![\mathtt{bool} + \mathtt{bool}]\!]_* & = & (\mathbb{B} + \mathbb{B}) + E \\ [\![\mathtt{bool} \times \mathtt{bool}]\!]_* & = & (\mathbb{B} \times \mathbb{B}) + E \end{array}$$

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

CBV denotational semantics

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

$$
\begin{array}{rcl}
[\![\texttt{bool}]\!] & = & \mathbb{B} \\
[\![A + B]\!] & = & [\![A]\!] + [\![B]\!] \\
[\![A \to B]\!] & = & [\![A]\!] \to ([\![B]\!] + E) \\
[\![() \to B]\!] & = & [\![B]\!] + E \\
[\![\Gamma]\!] & = & \displaystyle\prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]
\end{array}
$$

CBV denotational semantics

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

$$\begin{array}{rcl} [\![\texttt{bool}]\!] & = & \mathbb{B} \\ [\![A + B]\!] & = & [\![A]\!] + [\![B]\!] \\ [\![A \to B]\!] & = & [\![A]\!] \to ([\![B]\!] + E) \\ [\![() \to B]\!] & = & [\![B]\!] + E \\ [\![\Gamma]\!] & = & \prod_{(\texttt{x}:A)\in\Gamma} [\![A]\!] \end{array}$$

Each term $\Gamma \vdash M : B$ denotes a function $[\![M]\!] \ : \ [\![\Gamma]\!] \longrightarrow ([\![B]\!] + E)$.

Semantics of term constructors

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

$$[\![\lambda \mathtt{x}_A.\, M]\!] \; : \; \rho \longmapsto \mathsf{inl}\ \lambda a \in [\![A]\!].\, [\![M]\!](\rho, \mathtt{x} \mapsto a)$$

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

$$[\![M\, N]\!] \; : \; \rho \longmapsto \mathsf{match}\ [\![M]\!]\rho\ \mathsf{as} \begin{cases} \mathsf{inl}\ f. & \mathsf{match}\ [\![N]\!]\rho\ \mathsf{as} \begin{cases} \mathsf{inl}\ x. & f(x) \\ \mathsf{inr}\ e. & \mathsf{inr}\ e \end{cases} \\ \mathsf{inr}\ e. & \mathsf{inr}\ e \end{cases}$$

More term constructors

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

$$[\![\mathtt{inl}^{A,B}\ M]\!]\ :\ \rho \longmapsto \left\{ \begin{array}{ll} \mathsf{inl}\ a. & \mathsf{inl}\ \mathsf{inl}\ a \\ \mathsf{inr}\ e. & \mathsf{inr}\ e \end{array} \right.$$

More term constructors

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

$$[\![\mathtt{inl}^{A,B}\ M]\!]\ :\ \rho \longmapsto \begin{cases} \mathsf{inl}\ a. & \mathsf{inl}\ \mathsf{inl}\ a \\ \mathsf{inr}\ e. & \mathsf{inr}\ e \end{cases}$$

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

CBV Substitution Lemma: What Doesn't Work

Can we obtain $[\![N[M/\mathtt{x}]]\!]$ from $[\![M]\!]$ and $[\![N]\!]$?

CBV Substitution Lemma: What Doesn't Work

Can we obtain $[\![N[M/\mathtt{x}]]\!]$ from $[\![M]\!]$ and $[\![N]\!]$? Not in CBV.

CBV Substitution Lemma: What Doesn't Work

Can we obtain $[\![N[M/\mathtt{x}]]\!]$ from $[\![M]\!]$ and $[\![N]\!]$? Not in CBV.

Example that rules out a general substitution lemma

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

$$\begin{array}{rcl}
M & \overset{\text{def}}{=} & \mathtt{error}\ \mathrm{CRASH} \\
N & \overset{\text{def}}{=} & \mathtt{true} \\
N' & \overset{\text{def}}{=} & \mathtt{match}\ \mathtt{x}\ \mathtt{as}\ \{\mathtt{true}.\,\mathtt{true},\ \mathtt{false}.\,\mathtt{true}\} \\
[\![N]\!] & = & [\![N']\!] \qquad \text{because } N =_{\eta\,\mathtt{bool}} N' \\
[\![N[M/\mathtt{x}]]\!] & \neq & [\![N'[M/\mathtt{x}]]\!]
\end{array}$$

CBV Substitution Lemma: What Doesn't Work

Can we obtain $[\![N[M/\mathtt{x}]]\!]$ from $[\![M]\!]$ and $[\![N]\!]$? Not in CBV.

Example that rules out a general substitution lemma

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

$$\begin{array}{rcl}
M & \stackrel{\mathrm{def}}{=} & \mathtt{error}\ \mathrm{CRASH} \\
N & \stackrel{\mathrm{def}}{=} & \mathtt{true} \\
N' & \stackrel{\mathrm{def}}{=} & \mathtt{match}\ \mathtt{x}\ \mathtt{as}\ \{\mathtt{true}.\,\mathtt{true},\ \mathtt{false}.\,\mathtt{true}\} \\
[\![N]\!] & = & [\![N']\!] \qquad \text{because } N =_{\eta\,\mathtt{bool}} N' \\
[\![N[M/\mathtt{x}]]\!] & \neq & [\![N'[M/\mathtt{x}]]\!]
\end{array}$$

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

Values

The following terms are called values.

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

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

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

Denotational semantics of values

Each value $\Gamma \vdash V : A$ denotes a function $[\![V]\!]^{\mathsf{val}} \;:\; [\![\Gamma]\!] \longrightarrow [\![A]\!]$.

$$\begin{array}{lcccl}
[\![\mathtt{x}]\!]^{\mathsf{val}} & : & \rho & \longmapsto & \rho_{\mathtt{x}} \\
[\![\mathtt{true}]\!]^{\mathsf{val}} & : & \rho & \longmapsto & \mathsf{true} \\
[\![\mathtt{inl}\ V]\!]^{\mathsf{val}} & : & \rho & \longmapsto & \mathsf{inl}\ [\![V]\!]^{\mathsf{val}}\rho \\
[\![\lambda \mathtt{x}_A.\, M]\!]^{\mathsf{val}} & : & \rho & \longmapsto & \lambda a \in [\![A]\!].\, [\![M]\!](\rho, \mathtt{x} \mapsto [\![a]\!])
\end{array}$$

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

$$[\![V]\!] \;:\; \rho \longmapsto \mathsf{inl}\ [\![V]\!]^{\mathsf{val}}\rho$$

Substitution Lemma For Values

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

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

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

Likewise for substitution of values into values.

Soundness of CBV Denotational Semantics

- If $M \Downarrow V$ then $[\![M]\!]\varepsilon = \mathsf{inl}\ ([\![V]\!]^{\mathsf{val}}\varepsilon)$.
- If $M \lightning e$ then $[\![M]\!]\varepsilon = \mathsf{inr}\ e$.

Proof by induction, using the substitution lemma.

Fine-Grain Call-By-Value

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

- A value $\Gamma \vdash^{\mathsf{v}} V : A$ denotes a function $[\![V]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!]$.
- A computation $\Gamma \vdash^{\mathsf{c}} M : A$ denotes a function $[\![M]\!] : [\![\Gamma]\!] \longrightarrow [\![A]\!] + E$.

Key typing rules

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

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

Semantics of returning and sequencing

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

$$[\![\mathtt{return}\ V]\!]\ :\ \rho \longmapsto \mathsf{inl}\ [\![V]\!]\rho$$

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

$$[\![M\ \mathtt{to}\ \mathtt{x}.\ N]\!]\ :\ \rho \longmapsto \mathsf{match}\ [\![M]\!]\rho\ \mathsf{as} \left\{ \begin{array}{ll} \mathsf{inl}\ a. & [\![N]\!](\rho, \mathtt{x} \mapsto a) \\ \mathsf{inr}\ e. & \mathsf{inr}\ e \end{array} \right.$$

Syntax

For connectives $\mathtt{bool}, +, \rightarrow$ the syntax is as follows.

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

Syntax

For connectives $\texttt{bool}, +, \to$ the syntax is as follows.

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

We don't allow "complex values" such as

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

These would complicate the operational semantics.

Definitional interpreter for fine-grain CBV

We evaluate a closed computation $\vdash^{\mathsf{c}} M : A$ to a closed value $\vdash^{\mathsf{v}} V : A$.
To evaluate

- $\mathtt{return}\ V$: return V.
- $M\ \mathtt{to}\ \mathtt{x}.\ N$, evaluate M. If this returns V, evaluate $N[V/\mathtt{x}]$.
- $\mathtt{let}\ (\mathtt{x}\ \mathtt{be}\ V,\ \mathtt{y}\ \mathtt{be}\ W).\ M$, evaluate $M[V/\mathtt{x}, W/\mathtt{y}]$.
- $(\lambda \mathtt{x}.\, M)\, V$, evaluate $M[V/\mathtt{x}]$.
- $\mathtt{match}\ \mathtt{inl}\ V\ \mathtt{as}\ \{\mathtt{inl}\ \mathtt{x}.\ N,\ \mathtt{inr}\ \mathtt{x}.\ N'\}$: evaluate $N[V/\mathtt{x}]$.

Equational theory

β-laws

$$\begin{array}{rcl}
\mathtt{match}\ (\mathtt{inl}\ V)\ \mathsf{as}\ \{\mathtt{true}.\, M, \mathtt{false}.\, M'\} & = & M[V/\mathtt{x}] \\
(\lambda\mathtt{x}.\, M)\, V & = & M[V/\mathtt{x}] \\
\mathtt{let}\ (\mathtt{x}\ \mathsf{be}\ V,\ \mathtt{y}\ \mathsf{be}\ W).\ M & = & M[V/\mathtt{x}, W/\mathtt{y}]
\end{array}$$

η-laws

$$\begin{array}{rcl}
M[V/\mathtt{z}] & = & \mathtt{match}\ V\ \mathsf{as}\ \{\mathtt{inl}\ \mathtt{x}.\, M[\mathtt{inl}\ \mathtt{x}/\mathtt{z}],\ \mathtt{inr}\ \mathtt{y}.\, M[\mathtt{inr}\ \mathtt{x}/\mathtt{z}]\} \\
V & = & \lambda\mathtt{x}.\ V\mathtt{x}
\end{array}$$

Sequencing laws

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

CBV to fine-grain call-by-value

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

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

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

$$\begin{array}{rcl}
\mathtt{x} & \longmapsto & \mathtt{x} \\
\lambda \mathtt{x}.\, M & \longmapsto & \lambda \mathtt{x}.\, \hat{M} \\
\mathtt{inl}\ V & \longmapsto & \mathtt{inl}\ \check{V}
\end{array}$$

Nullary functions

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

$$
\begin{array}{rcl@{\qquad}rcl}
T A & \stackrel{\mathrm{def}}{=} & () \to A & [\![T A]\!] & = & [\![A]\!] + E \\
\mathtt{thunk}\ M & \stackrel{\mathrm{def}}{=} & \lambda(). M & [\![\mathtt{thunk}\ M]\!] & = & [\![M]\!] \\
\mathtt{force}\ V & \stackrel{\mathrm{def}}{=} & V\,() & [\![\mathtt{force}\ V]\!] & = & [\![V]\!]
\end{array}
$$

Nullary functions

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

$$\begin{array}{rcl@{\qquad}rcl} TA & \stackrel{\text{def}}{=} & () \to A & [\![TA]\!] & = & [\![A]\!] + E \\ \texttt{thunk}\ M & \stackrel{\text{def}}{=} & \lambda().\ M & [\![\texttt{thunk}\ M]\!] & = & [\![M]\!] \\ \texttt{force}\ V & \stackrel{\text{def}}{=} & V\,() & [\![\texttt{force}\ V]\!] & = & [\![V]\!] \end{array}$$

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

Nullary functions

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

$$\begin{array}{rcl} TA & \stackrel{\text{def}}{=} & () \to A \\ \mathtt{thunk}\ M & \stackrel{\text{def}}{=} & \lambda().\,M \\ \mathtt{force}\ V & \stackrel{\text{def}}{=} & V\,() \end{array} \qquad \begin{array}{rcl} [\![TA]\!] & = & [\![A]\!] + E \\ [\![\mathtt{thunk}\ M]\!] & = & [\![M]\!] \\ [\![\mathtt{force}\ V]\!] & = & [\![V]\!] \end{array}$$

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

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:

$$\begin{aligned}
[\![\texttt{bool} \to (\texttt{bool} \to \texttt{bool})]\!]_* &= (\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E)) \\
[\![\texttt{bool} + \texttt{bool}]\!]_* &= ((\mathbb{B} + E) + (\mathbb{B} + E)) + E \\
[\![\texttt{bool} \mathrel{\Pi} \texttt{bool}]\!]_* &= (\mathbb{B} + E) \times (\mathbb{B} + E)
\end{aligned}$$

Thus we define

$$\begin{aligned}
[\![\texttt{bool}]\!]_* &= \mathbb{B} + E \\
[\![A + B]\!]_* &= ([\![A]\!]_* + [\![B]\!]_*) + E \\
[\![A \to B]\!]_* &= [\![A]\!]_* \to [\![B]\!]_* \\
[\![A \mathrel{\Pi} B]\!]_* &= [\![A]\!]_* \times [\![B]\!]_* \\
[\![\Gamma]\!] &= \prod_{(\mathtt{x}:A) \in \Gamma} [\![A]\!]_*
\end{aligned}$$

Each term $\Gamma \vdash M : B$ should denote a function $[\![M]\!] \ : \ [\![\Gamma]\!] \longrightarrow [\![B]\!]_*$.

Naive semantics: what goes wrong

$$\frac{}{\Gamma \vdash \texttt{error}\ \mathrm{CRASH} : B}$$

denotes $\rho \longmapsto$?

Naive semantics: what goes wrong

$$\frac{}{\Gamma \vdash \texttt{error}\ \mathrm{CRASH} : B} \qquad \text{denotes } \rho \longmapsto ?$$

Example:

- suppose $B = \texttt{bool} \to (\texttt{bool} \to \texttt{bool})$
- then B denotes $(\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E))$
- and $\texttt{error}\ \mathrm{CRASH} \simeq_{\mathbf{CBN}} \lambda\mathrm{x}.\ \lambda\mathrm{y}.\ \texttt{error}\ \mathrm{CRASH}$
- so the answer should be $\lambda x.\ \lambda y.\ \mathsf{inr}\ \mathrm{CRASH}$.

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

Naive semantics: what goes wrong

$$\frac{}{\Gamma \vdash \texttt{error}\ \mathrm{CRASH} : B} \qquad \text{denotes } \rho \longmapsto ?$$

Example:

- suppose $B = \texttt{bool} \to (\texttt{bool} \to \texttt{bool})$
- then B denotes $(\mathbb{B} + E) \to ((\mathbb{B} + E) \to (\mathbb{B} + E))$
- and $\texttt{error}\ \mathrm{CRASH} \simeq_{\mathbf{CBN}} \lambda \mathrm{x}.\ \lambda \mathrm{y}.\ \texttt{error}\ \mathrm{CRASH}$
- so the answer should be $\lambda x.\ \lambda y.$ 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:

$$\begin{array}{rcl}
[\![\texttt{bool} \to (\texttt{bool} \to \texttt{bool})]\!] & = & ((\mathbb{B}+E) \to ((\mathbb{B}+E) \to (\mathbb{B}+E)), \\
 & & \lambda \text{x}.\lambda \text{y}.\text{inr CRASH}, \\
 & & \lambda \text{x}.\lambda \text{y}.\text{inr BANG}) \\
[\![\texttt{bool} + \texttt{bool}]\!] & = & (((\mathbb{B}+E) + (\mathbb{B}+E)) + E, \\
 & & \text{inr CRASH}, \\
 & & \text{inr BANG}) \\
[\![\texttt{bool} \,\Pi\, \texttt{bool}]\!] & = & ((\mathbb{B}+E) \times (\mathbb{B}+E), \\
 & & (\text{inr CRASH}, \text{inr CRASH}), \\
 & & (\text{inr BANG}, \text{inr BANG}))
\end{array}$$

CBN semantics of errors

$$[\![\texttt{bool}]\!] \quad = \quad (\mathbb{B} + E, \mathsf{inr}\ \mathrm{CRASH}, \mathsf{inr}\ \mathrm{BANG})$$

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

then $[\![A + B]\!] = ((X + Y) + E, \mathsf{inr}\ \mathrm{CRASH}, \mathsf{inr}\ \mathrm{BANG})$

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

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

CBN semantics of errors

$$[\![\mathtt{bool}]\!] \quad = \quad (\mathbb{B} + E, \mathsf{inr}\ \mathrm{CRASH}, \mathsf{inr}\ \mathrm{BANG})$$

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

$$\begin{array}{lrcl}
\text{then} & [\![A + B]\!] & = & ((X + Y) + E, \mathsf{inr}\ \mathrm{CRASH}, \mathsf{inr}\ \mathrm{BANG}) \\
\text{and} & [\![A \to B]\!] & = & (X \to Y, \lambda x.\, c', \lambda x.\, b') \\
\text{and} & [\![A \,\Pi\, B]\!] & = & (X \times Y, (c, c'), (b, b'))
\end{array}$$

$$[\![\Gamma]\!] \quad = \quad \prod_{\substack{(\mathtt{x}:A)\in\Gamma \\ [\![A]\!]=(X,c,b)}} X$$

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

Semantics of term constructors

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

$$[\![\texttt{true}]\!] \ : \ \rho \longmapsto \mathsf{inl\ true}$$

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

$$[\![\texttt{match}\ M\ \texttt{as}\ \{\texttt{true}.\ N,\ \texttt{false}.\ N'\}]\!] \ : \ \rho \quad \longmapsto$$

$$\mathsf{match}\ [\![M]\!]\rho\ \mathsf{as} \begin{cases} \mathsf{inl\ true.} & [\![N]\!]\rho \\ \mathsf{inl\ false.} & [\![N']\!]\rho \\ \mathsf{inr}\ \mathrm{CRASH.} & c \\ \mathsf{inr}\ \mathrm{BANG.} & b \end{cases} \qquad \text{where } [\![B]\!] = (Y, c, b)$$

More term constructors

$$\begin{array}{llll}
[\![\lambda \mathtt{x}.M]\!] & : & \rho & \longmapsto & \lambda a.\,[\![M]\!](\rho, \mathtt{x} \mapsto a) \\
[\![M\;N]\!] & : & \rho & \longmapsto & [\![M]\!]\,[\![N]\!] \\
[\![\mathtt{x}]\!] & : & \rho & \longmapsto & \rho_{\mathtt{x}} \\
\mathtt{error}\ \mathrm{CRASH} & : & \rho & \longmapsto & c
\end{array}$$

Soundness/adequacy

- If $M \Downarrow T$ then $[\![M]\!]\varepsilon = [\![T]\!]\varepsilon$.
- If $M \lightning \mathrm{CRASH}$ then $[\![M]\!]\varepsilon = c$.
- If $M \lightning \mathrm{BANG}$ then $[\![M]\!]\varepsilon = b$.

Proved by induction, using the substitution lemma.

Notation for E-pointed sets

- Free E-pointed set on a set X.

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

- Product of two E-pointed sets.

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

- Unit E-pointed set. $\qquad 1_\Pi \quad \stackrel{\text{def}}{=} \quad (1, (\,), (\,))$
- Product of a family of E-pointed sets.

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

- Exponential E-pointed set.

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

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

Summary of call-by-name semantics

A type denotes an E-pointed set.

$$\begin{array}{rcl} [\![\texttt{bool}]\!] & = & F^E(1+1) \\ [\![A+B]\!] & = & F^E(U^E[\![A]\!] + U^E[\![B]\!]) \\ [\![A \to B]\!] & = & U^E[\![A]\!] \to [\![B]\!] \\ [\![A \,\Pi\, B]\!] & = & [\![A]\!] \,\Pi\, [\![B]\!] \end{array}$$

A typing context denotes a set.

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

A term $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a function $[\![\Gamma]\!] \longrightarrow [\![\underline{B}]\!]$.

Summary of call-by-value semantics

A type denotes a set.

$$\begin{array}{rcl} [\![\texttt{bool}]\!] & = & 1+1 \\ [\![A+B]\!] & = & [\![A]\!]+[\![B]\!] \\ [\![A \to B]\!] & = & U^E([\![A]\!] \to F^E[\![B]\!]) \\ [\![TB]\!] & = & U^E F^E [\![B]\!] \end{array}$$

A typing context denotes a set.

$$[\![\Gamma]\!] \quad = \quad \prod_{(\mathtt{x}:A)\in\Gamma} [\![A]\!]$$

A computation $\Gamma \vdash^{\mathsf{c}} M : B$ denotes a function $[\![\Gamma]\!] \longrightarrow F^E[\![B]\!]$.

Call-By-Push-Value Types

Two kinds of type:

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

Call-By-Push-Value Types

Two kinds of type:

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

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

computation type $\underline{B} ::= \;\; FA \mid A \to \underline{B} \mid 1_{\Pi} \mid \underline{B} \mathbin{\Pi} \underline{B} \mid \prod_{i \in \mathbb{N}} \underline{B}_i$

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 $A ::= \; U\underline{B} \mid 1 \mid A \times A \mid 0 \mid A + A \mid \sum_{i \in \mathbb{N}} A_i$

computation type $\underline{B} ::= \; FA \mid A \to \underline{B} \mid 1_{\Pi} \mid \underline{B} \, \Pi \, \underline{B} \mid \prod_{i \in \mathbb{N}} \underline{B}_i$

Strangely function types are computation types, and $\lambda \mathrm{x}.M$ is a computation.

Judgements

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 Γ is a finite set of identifiers with associated value type

$$\mathtt{x}_0 : A_0, \ldots, \mathtt{x}_{m-1} : A_{m-1}$$

Judgements

An identifier gets bound to a value, so it has value type.

A context Γ is a finite set of identifiers with associated value type

$$\mathtt{x}_0 : A_0, \ldots, \mathtt{x}_{m-1} : A_{m-1}$$

Two judgements:

- A value $\Gamma \vdash^{\mathsf{v}} V : A$ denotes a function $[\![V]\!] \ : \ [\![\Gamma]\!] \longrightarrow [\![A]\!]$.
- A computation $\Gamma \vdash^{\mathsf{c}} M : \underline{B}$ denotes a function $[\![M]\!] \ : \ [\![\Gamma]\!] \longrightarrow [\![\underline{B}]\!]$.

The type FA

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

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

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

The type FA

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

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

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

$$\begin{array}{lcl} [\![\mathtt{return}\ V]\!] & : & \rho \longmapsto \mathsf{inl}\ [\![V]\!]\rho \\ [\![M\ \mathtt{to}\ \mathtt{x}.\ N]\!] & : & \rho \longmapsto \end{array}$$

$$\mathsf{match}\ [\![M]\!]\rho\ \mathsf{as} \begin{cases} \mathsf{inl}\ a. & [\![N]\!](\rho, \mathtt{x} \mapsto a) \\ \mathsf{inr}\ \mathrm{CRASH}. & c \\ \mathsf{inr}\ \mathrm{BANG}. & b \end{cases}$$

$$\text{where } [\![\underline{B}]\!] = (Y, c, b)$$

The type $U\underline{B}$

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

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

The type $U\underline{B}$

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

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

$$\begin{array}{rcl} [\![\texttt{thunk}\ M]\!] & = & [\![M]\!] \\ [\![\texttt{force}\ V]\!] & = & [\![V]\!] \end{array}$$

Identifiers

An identifier is a value.

$$\frac{}{\Gamma \vdash^{\mathsf{v}} \mathtt{x} : A} \, (\mathtt{x} : A) \in \Gamma$$

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

Tuples

$$\frac{\Gamma \vdash^{\mathsf{v}} V : A_{\hat{\imath}}}{\Gamma \vdash^{\mathsf{v}} \mathtt{in}_{\hat{\imath}}\, V : \sum_{i \in I} A_i}\ \hat{\imath} \in I \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : \sum_{i \in I} A_i \quad \Gamma, \mathtt{x} : A_i \vdash^{\mathsf{c}} M_i : \underline{B}\ \ (\forall i \in I)}{\Gamma \vdash^{\mathsf{c}} \mathtt{match}\ V\ \mathtt{as}\ \{\mathtt{in}_i\, \mathtt{x}.M_i\}_{i \in I} : \underline{B}}$$

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

The rules for 1 are similar.

Functions

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

$$\frac{\Gamma \vdash^{\mathsf{c}} M_i : \underline{B}_i \ \ (\forall i \in I)}{\Gamma \vdash^{\mathsf{c}} \lambda \{^{i}. M_i\}_{i \in I} : \prod_{i \in I} \underline{B}_i} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : \prod_{i \in I} \underline{B}_i}{\Gamma \vdash^{\mathsf{c}} M^{\hat{\imath}} : \underline{B}_{\hat{\imath}}} \ \hat{\imath} \in I$$

Functions

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

$$\frac{\Gamma \vdash^{\mathsf{c}} M_i : \underline{B}_i \;\; (\forall i \in I)}{\Gamma \vdash^{\mathsf{c}} \lambda\{^{i}. M_i\}_{i\in I} : \prod_{i\in I}\underline{B}_i} \qquad \frac{\Gamma \vdash^{\mathsf{c}} M : \prod_{i\in I}\underline{B}_i}{\Gamma \vdash^{\mathsf{c}} M^{\hat{\imath}} : \underline{B}_{\hat{\imath}}}\ \hat{\imath}\in I$$

It is often convenient to write applications operand-first, as $V\text{`}M$ and ${}^{\hat{\imath}}\text{`}M$.

Definitional interpreter for call-by-push-value

The terminals are computations: $\texttt{return}\ V \qquad \lambda\texttt{x}.M \qquad \lambda\{^{i}.\,M_i\}_{i \in I}$

Definitional interpreter for call-by-push-value

The terminals are computations: $\mathtt{return}\ V \qquad \lambda\mathtt{x}.M \qquad \lambda\{^i.\, M_i\}_{i\in I}$

To evaluate

- $\mathtt{return}\ V$: return $\mathtt{return}\ V$.
- $M\ \mathtt{to}\ \mathtt{x}.\ N$: evaluate M. If this returns $\mathtt{return}\ V$, then evaluate $N[V/\mathtt{x}]$.
- $\lambda\mathtt{x}.N$: return $\lambda\mathtt{x}.N$.
- MV: evaluate M. If this returns $\lambda\mathtt{x}.N$, evaluate $N[V/\mathtt{x}]$.
- $\lambda\{^i.\, N_i\}_{i\in I}$: return $\lambda\{i.N_i\}_{i\in I}$.
- $M^{\hat{\imath}}$: evaluate M. If this returns $\lambda\{^i.\, N_i\}_{i\in I}$, evaluate $N_{\hat{\imath}}$.
- $\mathtt{let}\ (\mathtt{x}\ \mathtt{be}\ V,\ \mathtt{y}\ \mathtt{be}\ W).\ M$: evaluate $M[V/\mathtt{x}, W/\mathtt{y}]$.
- $\mathtt{force}\ \mathtt{thunk}\ M$: evaluate M.
- $\mathtt{match}\ \mathtt{in}_{\hat{\imath}}\, V\ \mathtt{as}\ \{\mathtt{in}_i\, .M_i\}_{i\in I}$: evaluate $M_{\hat{\imath}}[V/\mathtt{x}]$.
- $\mathtt{match}\ \langle V, V'\rangle\ \mathtt{as}\ \langle \mathtt{x}, \mathtt{y}\rangle.M$: evaluate $M[V/\mathtt{x}, V'/\mathtt{y}]$.
- $\mathtt{error}\ e$, print error message e and stop.

Equational theory

β-laws

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

η-laws

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

Sequencing laws

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

Decomposing CBV into CBPV

A CBV type translates into a value type.

$$\begin{array}{rcl} A \to B & \longmapsto & U(A \to FB) \\ TB & \longmapsto & UFB \end{array}$$

Decomposing CBV into CBPV

A CBV type translates into a value type.

$$\begin{array}{rcl} A \to B & \longmapsto & U(A \to FB) \\ TB & \longmapsto & UFB \end{array}$$

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

Decomposing CBV into CBPV

A CBV type translates into a value type.

$$\begin{array}{rcl} A \to B & \longmapsto & U(A \to FB) \\ TB & \longmapsto & UFB \end{array}$$

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

$$\begin{array}{rcl} \lambda\mathtt{x}.\, M & \longmapsto & \mathtt{thunk}\ \lambda\mathtt{x}.\, M \\ V\,W & \longmapsto & (\mathtt{force}\ V)\,W \end{array}$$

Decomposing CBV into CBPV

A CBV type translates into a value type.

$$\begin{array}{rcl} A \to B & \longmapsto & U(A \to FB) \\ TB & \longmapsto & UFB \end{array}$$

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

$$\begin{array}{rcl} \lambda\mathtt{x}.\, M & \longmapsto & \mathtt{thunk}\ \lambda\mathtt{x}.\, M \\ V\, W & \longmapsto & (\mathtt{force}\ V)\, W \end{array}$$

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

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

Decomposing CBN into CBPV

A CBN type translates into a computation type.

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

Decomposing CBN into CBPV

A CBN type translates into a computation type.

$$\begin{array}{rcl} \texttt{bool} & \longmapsto & F(1+1) \\ \underline{A}+\underline{B} & \longmapsto & F(U\underline{A}+U\underline{B}) \\ \underline{A}\to\underline{B} & \longmapsto & U\underline{A}\to\underline{B} \end{array}$$

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

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

Summary

We've seen

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

Summary

We've seen

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

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

Summary

We've seen

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

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

Moggi's TA is UFA.

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

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 still don't understand why a function is a computation.

CK-machine

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

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

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

Initially, K is empty.

Some authors make K into a single context, called an "evaluation context".

Transitions for sequencing

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

$$\boxed{\begin{array}{lrl} M\ \texttt{to x.}\ N & K & \leadsto \\ M & \texttt{to x.}\ N :: K & \end{array}}$$

$$\boxed{\begin{array}{lrl} \texttt{return}\ V & \texttt{to x.}\ N :: K & \leadsto \\ N[V/\texttt{x}] & K & \end{array}}$$

Transitions for application

To evaluate $V\text{`}M$: evaluate M. If this returns $\lambda\mathtt{x}.N$, evaluate $N[V/\mathtt{x}]$.

$$\boxed{\begin{array}{lrl} V\text{`}M & K & \leadsto \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{lrl} \lambda\mathtt{x}.N & V :: K & \leadsto \\ N[V/\mathtt{x}] & K & \end{array}}$$

Those function rules again

$$\boxed{\begin{array}{lrl} V`M & K & \leadsto \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{lrl} \lambda\mathtt{x}.N & V :: K & \leadsto \\ N[V/\mathtt{x}] & K & \end{array}}$$

Those function rules again

$$\boxed{\begin{array}{lrl} V\text{`}M & K & \rightsquigarrow \\ M & V :: K & \end{array}}$$

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

We can read $V\text{`}$ as an instruction "push V".

We can read $\lambda\mathtt{x}$ as an instruction "pop $\mathtt{x}$".

Those function rules again

$$\boxed{\begin{array}{lrr} V`M & K & \leadsto \\ M & V :: K & \end{array}}$$

$$\boxed{\begin{array}{lrr} \lambda\mathtt{x}.N & V :: K & \leadsto \\ N[V/\mathtt{x}] & K & \end{array}}$$

We can read $V`$ as an instruction "push V".

We can read $\lambda\mathtt{x}$ as an instruction "pop $\mathtt{x}$".

Revisiting some equations:

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

Values and Computations

A value is, a computation does.

- A value of type $U\underline{B}$ is a thunk of a computation of type $\underline{B}$.
- A value of type $\sum_{i \in I} A_i$ is a pair $\langle i, V \rangle$.
- A value of type $A \times A'$ is a pair $\langle V, V' \rangle$.

- A computation of type FA aims to return a value of type A.
- A computation of type $A \to \underline{B}$
  aims to pop a value of type A
  then behave in $\underline{B}$.
- A computation of type $\prod_{i \in I} \underline{B}_i$
  aims to pop a tag $i \in I$
  then behave in $\underline{B}_i$.

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 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 $\Gamma \vdash^{\mathsf{c}} P \colon \underline{C}$

Γ	P	$\underline{C}$	`nil`	$\underline{C}$

Transitions

Γ	M `to x.` N	$\underline{B}$	K	$\underline{C}$	$\leadsto$
Γ	M	FA	`to x.` $N :: K$	$\underline{C}$	

Γ	`return` V	FA	`to x.` $N :: K$	$\underline{C}$	$\leadsto$
Γ	$N[V/\mathtt{x}]$	$\underline{B}$	K	$\underline{C}$	

Typically Γ would be empty and $\underline{C} = F\,\mathtt{bool}$.

Typing the CK-machine

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

Γ	P	$\underline{C}$	`nil`	$\underline{C}$	

Transitions

Γ	M `to x.` N	$\underline{B}$	K	$\underline{C}$	$\leadsto$
Γ	M	FA	`to x.` $N :: K$	$\underline{C}$	

Γ	`return` V	FA	`to x.` $N :: K$	$\underline{C}$	$\leadsto$
Γ	$N[V/\mathtt{x}]$	$\underline{B}$	K	$\underline{C}$	

Typically Γ would be empty and $\underline{C} = F\,\mathtt{bool}$.

We write $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}$ 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

$$\frac{}{\Gamma \vdash^{\mathsf{k}} \texttt{nil} : \underline{C} \Longrightarrow \underline{C}} \qquad \frac{\Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} M : \underline{B} \qquad \Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \texttt{to x.}\ M :: K : FA \Longrightarrow \underline{C}}$$

$$\frac{\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \hat{\imath} :: K : \prod_{i \in I} \underline{B}_i \Longrightarrow \underline{C}}\ \hat{\imath} \in I \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} V :: K : A \to \underline{B} \Longrightarrow \underline{C}}$$

Typing rules, read off from the CK-machine

Typing a stack

$$\frac{}{\Gamma \vdash^{\mathsf{k}} \texttt{nil} : \underline{C} \Longrightarrow \underline{C}} \qquad \frac{\Gamma, \mathtt{x} : A \vdash^{\mathsf{c}} M : \underline{B} \qquad \Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \texttt{to x. } M :: K : FA \Longrightarrow \underline{C}}$$

$$\frac{\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} \hat{\imath} :: K : \prod_{i \in I} \underline{B}_i \Longrightarrow \underline{C}}\ \hat{\imath} \in I \qquad \frac{\Gamma \vdash^{\mathsf{v}} V : A \qquad \Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C}}{\Gamma \vdash^{\mathsf{k}} V :: K : A \to \underline{B} \Longrightarrow \underline{C}}$$

Typing a CK-configuration

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

Operations on Stacks

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

Operations on Stacks

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

Operations on Stacks

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

Special Stacks

Continuations

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

Special Stacks

Continuations

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

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

Special Stacks

Continuations

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

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

Top-Level Stack

The top-level stack is $\Gamma \vdash^{\mathsf{k}} \texttt{nil} : \underline{C} \Longrightarrow \underline{C}$.
The top-level type is $\underline{C}$.

Special Stacks

Continuations

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

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

Top-Level Stack

The top-level stack is $\Gamma \vdash^{\mathsf{k}}$ `nil` $: \underline{C} \Longrightarrow \underline{C}$.
The top-level type is $\underline{C}$.

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

Stacks denote homomorphisms

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

where $[\![\underline{B}]\!] = (X, c, b)$ and $[\![\underline{C}]\!] = (Y, c', b')$.

What should K denote?

Stacks denote homomorphisms

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

where $[\![\underline{B}]\!] = (X, c, b)$ and $[\![\underline{C}]\!] = (Y, c', b')$.

What should K denote?

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

So we want $\quad [\![K]\!] \ : \ [\![\Gamma]\!] \times X \longrightarrow Y$.

Stacks denote homomorphisms

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

where $[\![\underline{B}]\!] = (X, c, b)$ and $[\![\underline{C}]\!] = (Y, c', b')$.

What should K denote?

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

So we want $\qquad [\![K]\!] \; : \; [\![\Gamma]\!] \times X \longrightarrow Y$.

This function should be homomorphic in its second argument:

$$\begin{array}{rcl} [\![K]\!](\rho, c) & = & c' \\ [\![K]\!](\rho, b) & = & b' \end{array}$$

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

Stacks denote homomorphisms

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

where $[\![\underline{B}]\!] = (X, c, b)$ and $[\![\underline{C}]\!] = (Y, c', b')$.

What should K denote?

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

So we want $\qquad [\![K]\!] \; : \; [\![\Gamma]\!] \times X \longrightarrow Y$.

This function should be homomorphic in its second argument:

$$\begin{array}{rcl} [\![K]\!](\rho, c) & = & c' \\ [\![K]\!](\rho, b) & = & b' \end{array}$$

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

We assume there's no exception handling.

Operations on stacks

We define $[\![K]\!]$ by induction on K.

Then we prove

- a weakening lemma
- a substitution lemma
- a dismantling lemma
- a concatenation lemma

providing a semantic counterpart for each operation on stacks.

Soundness of CK-machine

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

Soundness of CK-machine

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

$$\begin{array}{rcrcl} [\![(M,K)]\!] & : & [\![\Gamma]\!] & \longrightarrow & [\![\underline{C}]\!] \\ & & \rho & \longmapsto & [\![K]\!](\rho, [\![M]\!]\rho) \end{array}$$

Properties:

1. If $(M, K) \rightsquigarrow (M', K')$ then $[\![(M,K)]\!] = [\![(M',K')]\!]$.
2. $[\![(\mathtt{error}\ \mathrm{CRASH}, K)]\!]\rho = c'$.
3. $[\![(\mathtt{error}\ \mathrm{BANG}, K)]\!]\rho = b'$.

Adjunction between values and stacks

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

$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift left, "F^E"] \arrow[r, phantom, "\perp"] & E/\mathbf{Set} \arrow[l, shift left, "U^E"]\end{tikzcd}$$

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

State

Consider CBPV extended with two storage cells:
`l` stores a natural number, and $\mathtt{l}'$ stores a boolean.

State

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

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{nat} \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{l} := V.\ M : \underline{B}} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{nat} \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{read}\ \mathtt{l}\ \mathtt{as}\ \mathtt{x}.\ M : \underline{B}}$$

State

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

$$\frac{\Gamma \vdash^{\mathsf{v}} V : \mathtt{nat} \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{l} := V.\ M : \underline{B}} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{nat} \vdash^{\mathsf{c}} M : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \mathtt{read\ l\ as\ x.}\ M : \underline{B}}$$

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

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

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

Denotational semantics of state

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

- Algebra semantics.
- Intrinsic semantics.

Algebra semantics for state (briefly)

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

Algebra semantics for state (briefly)

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

A value type A denotes a set $[\![A]\!]$, a semantic domain for values.

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

Algebra semantics for state (briefly)

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

A value type A denotes a set $[\![A]\!]$, a semantic domain for values.

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

We complete the story with an adequacy theorem:

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

This requires an SC-configuration to have a denotation.

Intrinsic semantics of state

A value type A denotes a set $[\![A]\!]$, a semantic domain for values.

A computation type $\underline{B}$ denotes a set $[\![\underline{B}]\!]$,
a semantic domain for SC-configurations.

Intrinsic semantics of state

A value type A denotes a set $[\![A]\!]$, a semantic domain for values.

A computation type $\underline{B}$ denotes a set $[\![\underline{B}]\!]$,
a semantic domain for SC-configurations.

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

$$[\![(s, M)]\!] \;:\; [\![\Gamma]\!] \longrightarrow [\![\underline{B}]\!]$$

Intrinsic semantics of state

A value type A denotes a set $[\![A]\!]$, a semantic domain for values.

A computation type $\underline{B}$ denotes a set $[\![\underline{B}]\!]$,
a semantic domain for SC-configurations.

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

$$[\![(s, M)]\!] \ : \ [\![\Gamma]\!] \longrightarrow [\![\underline{B}]\!]$$

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

$$[\![M]\!] \ : \ S \times [\![\Gamma]\!] \longrightarrow [\![\underline{B}]\!]$$

State: semantics of types

An SC-configuration of type FA will terminate as $s, \texttt{return}\ V$.

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

An SC-configuration of type $A \to \underline{B}$ will pop $\texttt{x} : A$, then behave in $\underline{B}$.

$$[\![A \to \underline{B}]\!] = [\![A]\!] \to [\![\underline{B}]\!]$$

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

$$[\![\prod_{i \in I} \underline{B}_i]\!] = \prod_{i \in I} [\![\underline{B}_i]\!]$$

A value $\Gamma \vdash^{\mathsf{v}} V : U\underline{B}$ can be forced in any state s, giving an SC-configuration $s, \texttt{force}\ V$.

$$[\![U\underline{B}]\!] = S \to [\![\underline{B}]\!]$$

State: the value/stack adjunction

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

What should K denote?

State: the value/stack adjunction

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

What should K denote?

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

So we want $\llbracket K \rrbracket \ : \ \llbracket \Gamma \rrbracket \times \llbracket \underline{B} \rrbracket \longrightarrow \llbracket \underline{C} \rrbracket$.

State: the value/stack adjunction

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

What should K denote?

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

So we want $[\![K]\!] \ : \ [\![\Gamma]\!] \times [\![\underline{B}]\!] \longrightarrow [\![\underline{C}]\!]$.

This gives an adjunction

$$\begin{tikzcd}
\mathbf{Set} \arrow[r, shift left, "S\times-"] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift left, "S\to-"]
\end{tikzcd}$$

between values and stacks.

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

For call-by-value we recover

$$\begin{array}{rcl} [\![\texttt{bool}_{\mathbf{CBV}}]\!] & = & 1+1 \\ [\![A \to_{\mathbf{CBV}} B]\!] & = & [\![U(A \to FB)]\!] \\ & = & S \to ([\![A]\!] \to (S \times [\![B]\!])) \end{array}$$

This is standard.

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

For call-by-value we recover

$$\begin{array}{rcl} [\![\texttt{bool}_{\mathbf{CBV}}]\!] & = & 1+1 \\ [\![A \to_{\mathbf{CBV}} B]\!] & = & [\![U(A \to FB)]\!] \\ & = & S \to ([\![A]\!] \to (S \times [\![B]\!])) \end{array}$$

This is standard.

For call-by-name we recover

$$\begin{array}{rcl} [\![\texttt{bool}_{\mathbf{CBN}}]\!] & = & [\![F(1+1)]\!] \\ & = & S \times (1+1) \\ [\![\underline{A} \to_{\mathbf{CBN}} \underline{B}]\!] & = & [\![U\underline{A} \to \underline{B}]\!] \\ & = & (S \to [\![\underline{A}]\!]) \to [\![\underline{B}]\!] \end{array}$$

This is O'Hearn's semantics of types for a stateful CBN language.

Naming and changing the current stack

Extend the language with two instructions:

- `letstk` α means let α be the current stack.
- `changestk` α means change the current stack to α.

Naming and changing the current stack

Extend the language with two instructions:

- `letstk` α means let α be the current stack.
- `changestk` α means change the current stack to α.

Execution takes places in a bigger language.

$$\boxed{\begin{array}{llllll} \Gamma & \texttt{letstk}\ \alpha.\ M & \underline{B} & K & \underline{C} \mid \Delta & \leadsto \\ \Gamma & M[K/\alpha] & \underline{B} & K & \underline{C} \mid \Delta & \end{array}}$$

$$\boxed{\begin{array}{llllll} \Gamma & \texttt{changestk}\ K.\ M & \underline{B'} & L & \underline{C} \mid \Delta & \leadsto \\ \Gamma & M & \underline{B} & K & \underline{C} \mid \Delta & \end{array}}$$

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

Typing judgements for control

We have typing judgements:

$$\Gamma \vdash^{\mathsf{v}} V : A \mid \Delta \qquad\qquad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta$$

The stack context Δ consists of declarations $\alpha : \underline{B}$,
meaning α is a stack from $\underline{B}$.

Typing judgements for control

We have typing judgements:

$$\Gamma \vdash^{\mathsf{v}} V : A \mid \Delta \qquad\qquad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \mid \Delta$$

The stack context Δ consists of declarations $\alpha : \underline{B}$,
meaning α is a stack from $\underline{B}$.

Example typing rules

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B} \ \mid \Delta, \alpha : \underline{B}}{\Gamma \vdash^{\mathsf{c}} \texttt{letstk}\ \alpha.\ M \ \mid \Delta}$$

$$\frac{\Gamma \vdash^{\mathsf{c}} M : \underline{B} \ \mid \Delta}{\Gamma \vdash^{\mathsf{c}} \texttt{changestk}\ \alpha.\ M : \underline{B}' \ \mid \Delta}\ (\alpha : \underline{B}) \in \Delta$$

Typing judgements for execution language

During execution, the top-level type $\underline{C}$ must be indicated:

$$\Gamma \vdash^{\mathsf{v}} V : A \; [\underline{C}] \; \Delta \qquad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \; [\underline{C}] \; \Delta$$

$$\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta \qquad \Gamma \vdash^{\mathsf{ck}} (M, K) : \underline{C} \mid \Delta$$

Typically Γ and Δ would be empty and $\underline{C} = F\,\texttt{bool}$.

Typing judgements for execution language

During execution, the top-level type $\underline{C}$ must be indicated:

$$\Gamma \vdash^{\mathsf{v}} V : A \; [\underline{C}] \; \Delta \qquad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \; [\underline{C}] \; \Delta$$

$$\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta \qquad \Gamma \vdash^{\mathsf{ck}} (M, K) : \underline{C} \mid \Delta$$

Typically Γ and Δ would be empty and $\underline{C} = F\,\mathtt{bool}$.

Example typing rules

$$\frac{}{\Gamma \vdash^{\mathsf{k}} \alpha : \underline{B} \Longrightarrow \underline{C} \;\mid \Delta} \; (\alpha : \underline{B}) \in \Delta$$

$$\frac{\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \;\mid \Delta \quad \Gamma \vdash^{\mathsf{c}} M : \underline{B} \; [\underline{C}] \; \Delta}{\Gamma \vdash^{\mathsf{c}} \mathtt{changestk} \; K. \; M : \underline{B}' \; [\underline{C}] \; \Delta}$$

Algebra semantics of control

Fix a set R, the semantic domain for CK-configurations.

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

Algebra semantics of control

Fix a set R, the semantic domain for CK-configurations.

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

Moggi's monad for control operators ("continuations") is $(- \to R) \to R$.

Algebra semantics of control

Fix a set R, the semantic domain for CK-configurations.

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

Moggi's monad for control operators ("continuations") is $(- \to R) \to R$.

Maybe we can build a denotational semantics
where a computation type $\underline{B}$ denotes an Eilenberg-Moore algebra $[\![\underline{B}]\!]_{\mathsf{alg}}$,
a semantic domain for computations.

Intrinsic semantics of control

The denotation of $\underline{B}$ is a semantic domain for stacks from $\underline{B}$.

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

Intrinsic semantics of control

The denotation of $\underline{B}$ is a semantic domain for stacks from $\underline{B}$.

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

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

$$[\![M]\!] \;:\; [\![\Gamma]\!] \times [\![\underline{B}]\!] \times [\![\Delta]\!] \longrightarrow R$$

A value $\Gamma \vdash^{\mathsf{v}} V : A \;\mid \Delta$ denotes

$$[\![V]\!] \;:\; [\![\Gamma]\!] \times [\![\Delta]\!] \longrightarrow [\![A]\!]$$

Control: semantics of types

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

$$[\![FA]\!] = [\![A]\!] \to R$$

A stack from $A \to \underline{B}$ is a pair $V :: K$.

$$[\![A \to \underline{B}]\!] = [\![A]\!] \times [\![\underline{B}]\!]$$

A stack from $\prod_{i \in I} \underline{B}_i$ is a pair $\hat{\imath} :: K$.

$$[\![\prod_{i \in I} \underline{B}_i]\!] = \sum_{i \in I} [\![\underline{B}_i]\!]$$

A value of type $U\underline{B}$ can be forced alongside any stack K, giving a configuration.

$$[\![U\underline{B}]\!] = [\![\underline{B}]\!] \to R$$

Semantics of the execution language

The semantics of a term in the execution language
depends not only on the environment and the stack environment
but also on the top-level stack.

Semantics of the execution language

The semantics of a term in the execution language
depends not only on the environment and the stack environment
but also on the top-level stack.

In particular, a stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta$ denotes

$$[\![K]\!] \ : \ [\![\Gamma]\!] \times [\![\underline{C}]\!] \times [\![\Delta]\!] \longrightarrow [\![\underline{B}]\!]$$

Semantics of the execution language

The semantics of a term in the execution language
depends not only on the environment and the stack environment
but also on the top-level stack.

In particular, a stack $\Gamma \vdash^{\mathsf{k}} K : \underline{B} \Longrightarrow \underline{C} \mid \Delta$ denotes

$$[\![K]\!] \ : \ [\![\Gamma]\!] \times [\![\underline{C}]\!] \times [\![\Delta]\!] \longrightarrow [\![\underline{B}]\!]$$

That gives an adjunction

$$\begin{tikzcd} \mathbf{Set} \arrow[r, shift left, "- \to R"] \arrow[r, phantom, "\perp"] & \mathbf{Set}^{\mathsf{op}} \arrow[l, shift left, "- \to R"] \end{tikzcd}$$

between values and stacks.

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

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

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

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

For call-by-value we recover

$$\begin{array}{rcl}
[\![\mathtt{bool}_{\mathbf{CBV}}]\!] & = & 1+1 \\
[\![A \to_{\mathbf{CBV}} B]\!] & = & [\![U(A \to FB)]\!] \\
 & = & \neg([\![A]\!] \times \neg[\![B]\!])
\end{array}$$

This is standard.

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

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

For call-by-value we recover

$$\begin{array}{rcl} [\![\texttt{bool}_{\mathbf{CBV}}]\!] & = & 1+1 \\ [\![A \to_{\mathbf{CBV}} B]\!] & = & [\![U(A \to FB)]\!] \\ & = & \neg([\![A]\!] \times \neg[\![B]\!]) \end{array}$$

This is standard.

For call-by-name we recover

$$\begin{array}{rcl} [\![\texttt{bool}_{\mathbf{CBN}}]\!] & = & [\![F(1+1)]\!] \\ & = & \neg(1+1) \\ [\![\underline{A} \to_{\mathbf{CBN}} \underline{B}]\!] & = & [\![U\underline{A} \to \underline{B}]\!] \\ & = & \neg[\![\underline{A}]\!] \times [\![\underline{B}]\!] \end{array}$$

This is Streicher and Reus' semantics
for a CBN language with control operators.

Summary: adjunctions between values and stacks

For a set E, the adjunction $$\begin{tikzcd} \mathbf{Set} \arrow[r, bend left, "F^E"] \arrow[r, phantom, "\perp"] & E/\mathbf{Set} \arrow[l, bend left, "U^E"] \end{tikzcd}$$

models call-by-push-value with errors.

Summary: adjunctions between values and stacks

For a set E, the adjunction
$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift left, "F^E"] \arrow[r, phantom, "\perp"] & E/\mathbf{Set} \arrow[l, shift left, "U^E"]\end{tikzcd}$$
models call-by-push-value with errors.

For a set S, the adjunction
$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift left, "S\times-"] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift left, "S\to-"]\end{tikzcd}$$
models call-by-push-value with state.

Summary: adjunctions between values and stacks

For a set E, the adjunction
$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift left, "F^E"] \arrow[r, phantom, "\perp"] & E/\mathbf{Set} \arrow[l, shift left, "U^E"]\end{tikzcd}$$
models call-by-push-value with errors.

For a set S, the adjunction
$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift left, "S\times-"] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift left, "S\to-"]\end{tikzcd}$$
models call-by-push-value with state.

For a set R, the adjunction
$$\begin{tikzcd}\mathbf{Set} \arrow[r, shift left, "-\to R"] \arrow[r, phantom, "\perp"] & \mathbf{Set}^{\mathsf{op}} \arrow[l, shift left, "-\to R"]\end{tikzcd}$$
models call-by-push-value with control.

Summary: adjunctions between values and stacks

For a set E, the adjunction
$$\begin{tikzcd} \mathbf{Set} \arrow[r, shift left, "F^E"] \arrow[r, phantom, "\perp"] & E/\mathbf{Set} \arrow[l, shift left, "U^E"] \end{tikzcd}$$
models call-by-push-value with errors.

For a set S, the adjunction
$$\begin{tikzcd} \mathbf{Set} \arrow[r, shift left, "S\times -"] \arrow[r, phantom, "\perp"] & \mathbf{Set} \arrow[l, shift left, "S\to -"] \end{tikzcd}$$
models call-by-push-value with state.

For a set R, the adjunction
$$\begin{tikzcd} \mathbf{Set} \arrow[r, shift left, "-\to R"] \arrow[r, phantom, "\perp"] & \mathbf{Set}^{\mathsf{op}} \arrow[l, shift left, "-\to R"] \end{tikzcd}$$
models call-by-push-value with control.