

# A tutorial on call-by-push-value

Paul Blain Levy

University of Birmingham

February 20, 2012

# Outline

- 1 Typed  $\lambda$ -calculus
- 2 Typed  $\lambda$ -calculus: denotational semantics
- 3 Call-by-push-value
- 4 Stacks
- 5 State
- 6 Control

# Typed $\lambda$ -calculus

We consider typed  $\lambda$ -calculus with boolean, function and sum types.

## Types

$$A ::= \text{bool} \mid A + A \mid A \rightarrow A$$

Typing judgement  $\Gamma \vdash M : A$

## Terms

$$\begin{aligned} M ::= & \quad x \mid \text{let } M \text{ be } x. M \\ & \quad \mid \text{true} \mid \text{false} \mid \text{match } M \text{ as } \{\text{true}. M, \text{false}. M\} \\ & \quad \mid \text{inl } M \mid \text{inr } M \mid \text{match } M \text{ as } \{\text{inl } x. M, \text{inr } x. M\} \\ & \quad \mid \lambda x. M \mid MM \end{aligned}$$

# Equational Laws

We consider the equational theory generated by the  $\beta\eta$ -laws.

## $\eta$ -law for $A \rightarrow B$

Any term  $\Gamma \vdash M : A \rightarrow B$  can be expanded as

$$\lambda x. Mx$$

Anything of function type is a  $\lambda$ -abstraction.

## $\eta$ -law for bool

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

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

Anything of boolean type is a boolean.

The  $\eta$ -law for sum types is similar.

# Denotational semantics in Set

A type denotes a set.

$$\begin{aligned} \llbracket \text{bool} \rrbracket &= \mathbb{B} \stackrel{\text{def}}{=} \{\text{true}, \text{false}\} \\ \llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \end{aligned}$$

A term  $\Gamma \vdash M : B$  denotes a function  $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket B \rrbracket$ .

## Substitution Lemma

Given terms  $\Gamma, x : A \vdash M : B$  and  $\Gamma \vdash N : A$

we can obtain  $\llbracket M[N/x] \rrbracket$  from  $\llbracket M \rrbracket$  and  $\llbracket N \rrbracket$ . It is

$$\rho \mapsto \llbracket M \rrbracket(\rho, x \mapsto \llbracket N \rrbracket \rho)$$

## Corollary

The denotational semantics validates the  $\beta$  and  $\eta$  laws.

# Call-by-name evaluation of a closed term

In CBN the terminals are `true`, `false`, `inl M`, `inr M`,  $\lambda x.M$

To evaluate

- `true`: return `true`.
- $\lambda x.M$ : return  $\lambda x.M$ .
- `inl M`: return `inl M`.
- `let M be x. N`: evaluate  $N[M/x]$ .
- `match M as {true. N, false. N'}`: evaluate  $M$ . If it returns `true`, evaluate  $N$ , but if it returns `false`, evaluate  $N'$ .
- `match M as {inl x. N, inr x. N'}`: evaluate  $M$ . If it returns `inl P`, evaluate  $N[P/x]$ , but if it returns `inr P`, evaluate  $N'[P/x]$ .
- $MN$ : evaluate  $M$ . If it returns  $\lambda x.P$ , evaluate  $P[N/x]$ .

# Call-by-value evaluation of a closed term

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

To evaluate

- **true**: return **true**.
- $\lambda x.M$ : return  $\lambda x.M$ .
- **inl**  $M$ : evaluate  $M$ . If it returns  $T$ , return **inl**  $T$ .
- **let**  $M$  **be**  $x$ .  $N$ : evaluate  $M$ . If it returns  $T$ , evaluate  $N[T/x]$ .
- **match**  $M$  **as**  $\{\text{true}. N, \text{false}. N'\}$ : evaluate  $M$ . If it returns **true**, evaluate  $N$ , but if it returns **false**, evaluate  $N'$ .
- **match**  $M$  **as**  $\{\text{inl } x. N, \text{inr } x. N'\}$ : evaluate  $M$ . If it returns **inl**  $T$ , evaluate  $N[T/x]$ , but if it returns **inr**  $T$ , evaluate  $N'[T/x]$ .
- $MN$ : evaluate  $M$ . If it returns  $\lambda x.P$ , evaluate  $N$ . If that returns  $T$ , evaluate  $P[T/x]$ .

# Adding computational effects

## Errors

Let  $E = \{\text{CRASH}, \text{BANG}, \text{WALLOP}\}$  be a set of “errors”. We add

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

To evaluate **error**  $e$ : halt with error message  $e$ .

## Printing

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

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

To evaluate **print**  $c. M$ : print  $c$  and then evaluate  $M$ .

- ① Evaluate

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

in CBV and CBN

- ② Evaluate

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

in CBV and CBN.

- ③ Evaluate

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

in CBV and CBN.

# Big-Step Operational Semantics

We convert our CBV and CBN interpreters into big-step semantics, defined inductively.

**no effects** We define a relation  $M \Downarrow T$  meaning  $M$  evaluates to  $T$ .

**errors** We define a relation  $M \Downarrow T$  meaning  $M$  evaluates to  $T$ , and a relation  $M \Downarrow e$  meaning  $M$  raises error  $e$ .

**printing** We define a relation  $M \Downarrow m, T$  meaning  $M$  prints  $m \in \mathcal{A}^*$  and finally evaluates to  $T$ .

For example, in the case of printing we have rules such as

$$\frac{}{\text{true} \Downarrow \varepsilon, \text{true}} \quad \frac{M \Downarrow m, \text{true} \quad N \Downarrow m', T}{\text{match } M \text{ as } \{\text{true}. N, \text{false}. N'\} \Downarrow mm', T}$$

These are proved deterministic and total using Tait's method.

# Observational equivalence

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

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

for every ground (i.e. boolean) context  $\mathcal{C}[\cdot]$ .

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

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

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

## $\eta$ -law for bool

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

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

Anything of boolean type is a boolean.

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

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

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

because we can apply the context

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

Similarly the  $\eta$ -law for sum types is valid in CBV but not in CBN.

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

## $\eta$ -law for $A \rightarrow B$

Any term  $\Gamma \vdash M : A \rightarrow B$  can be expanded as

$$\lambda x. Mx$$

Anything of function type is a function.

This fails in CBV, but it holds in CBN.

Similarly

$$\begin{aligned}\lambda x. \text{error } e &\simeq_{\text{CBN}} \text{error } e \\ \lambda x. \text{print } c. M &\simeq_{\text{CBN}} \text{print } c. \lambda x. M\end{aligned}$$

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

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

The pure calculus satisfies all the  $\beta$ - and  $\eta$ -laws.

With computational effects,

- CBV satisfies  $\eta$  for boolean and sum types, but not function types
- CBN satisfies  $\eta$  for function types, but not boolean and sum types.

We want denotational semantics that validate the appropriate  $\eta$ -laws.

We'll do CBV first, as it's easier.

# Denotational Semantics of CBV (Moggi)

Take a (strong) monad  $T$  on **Set**.

- For errors:  $- + E$
- For printing:  $\mathcal{A}^* \times -$

Each type denotes a set (think: the set of terminals)

$$\begin{aligned} \llbracket \text{bool} \rrbracket &= \mathbb{B} \\ \llbracket A + B \rrbracket &= \llbracket A \rrbracket + \llbracket B \rrbracket \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow T\llbracket B \rrbracket \end{aligned}$$

Each term  $\Gamma \vdash M : B$  denotes a **Kleisli morphism**,

i.e. a function  $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} T\llbracket B \rrbracket$ .

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

# CBV Substitution Lemma: What Doesn't Work

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

## CBV Substitution Lemma: What Doesn't Work

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

# CBV Substitution Lemma: What Doesn't Work

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

## Example that rules out a general substitution lemma

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

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

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

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

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

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

# CBV Substitution Lemma: What Doesn't Work

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

## Example that rules out a general substitution lemma

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

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

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

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

$$\llbracket M \rrbracket = \llbracket M' \rrbracket \quad \text{because } M =_{\eta \text{ bool}} M'$$

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

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

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

The terminals are the closed values.

# Substitution Lemma For Values

Each value  $\Gamma \vdash V : B$  denotes a function  $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket V \rrbracket^{\text{val}}} \llbracket B \rrbracket$  such that

$$\begin{array}{ccc} \llbracket \Gamma \rrbracket & \xrightarrow{\llbracket V \rrbracket^{\text{val}}} & \llbracket B \rrbracket \\ & \searrow \llbracket V \rrbracket & \downarrow \eta[\llbracket B \rrbracket] \\ & & T[\llbracket B \rrbracket] \end{array} \quad \text{commutes.}$$

## Substitution Lemma

Given a term  $\Gamma, \mathbf{x} : A \vdash M : B$  and a value  $\Gamma \vdash V : A$  we can obtain  $\llbracket M[V/\mathbf{x}] \rrbracket$  from  $\llbracket M \rrbracket$  and  $\llbracket V \rrbracket^{\text{val}}$ . It is

$$\rho \longmapsto \llbracket M \rrbracket(\rho, \mathbf{x} \mapsto \llbracket V \rrbracket^{\text{val}} \rho)$$

## Errors

- If  $M \Downarrow V$  then  $\llbracket M \rrbracket_{\varepsilon} = \text{inl} (\llbracket V \rrbracket^{\text{val}}_{\varepsilon})$ .
- If  $M \not\Downarrow e$  then  $\llbracket M \rrbracket_{\varepsilon} = \text{inr } e$ .

## Printing

- If  $M \Downarrow m, V$  then  $\llbracket M \rrbracket_{\varepsilon} = \langle m, \llbracket V \rrbracket^{\text{val}}_{\varepsilon} \rangle$ .

These are straightforward inductions, using the substitution lemma.

# Naive Attempt At CBN: “Carrier” Semantics

Each type denotes a set (think: the set of closed terms).

For example  $\text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})$  should denote  $T\mathbb{B} \rightarrow (T\mathbb{B} \rightarrow T\mathbb{B})$ .

We define

$$\begin{aligned} \llbracket \text{bool} \rrbracket &= T\mathbb{B} \\ \llbracket A + B \rrbracket &= T(\llbracket A \rrbracket + \llbracket B \rrbracket) \\ \llbracket A \rightarrow B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket B \rrbracket \end{aligned}$$

Each term  $\Gamma \vdash M : B$  should denote a function  $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket B \rrbracket$ .

# Carrier Semantics: What Goes Wrong

$\overline{\Gamma \vdash \mathbf{error} \ e : B}$

denotes  $\rho \mapsto ?$

# Carrier Semantics: What Goes Wrong

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

denotes  $\rho \mapsto ?$

Example:

- suppose  $B = \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})$
- then  $B$  denotes  $(\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E))$
- and error  $e \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error } e$
- so the answer should be  $\lambda x. \lambda y. \text{inr } e$ .

Intuition: go down through the function types until we hit a boolean or sum type.

# Carrier Semantics: What Goes Wrong

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

denotes  $\rho \mapsto ?$

Example:

- suppose  $B = \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool})$
- then  $B$  denotes  $(\mathbb{B} + E) \rightarrow ((\mathbb{B} + E) \rightarrow (\mathbb{B} + E))$
- and error  $e \simeq_{\text{CBN}} \lambda x. \lambda y. \text{error } e$
- so the answer should be  $\lambda x. \lambda y. \text{inr } e$ .

Intuition: go down through the function types until we hit a boolean or sum type.

A similar problem arises with `match`.

# $E$ -pointed semantics of CBN types

A CBN type should denote a set  $X$  (the carrier) with some designated elements  $E \xrightarrow{\text{error}} X$ .

This is called an  $E$ -pointed set.

Thus `bool` denotes  $\mathbb{B} + E$  with  $e \mapsto \text{inr } e$ .

If  $\llbracket A \rrbracket = (X, \text{error})$  and  $\llbracket B \rrbracket = (Y, \text{error}')$ ,

- then  $A + B$  denotes  $(X + Y) + E$  with  $e \mapsto \text{inr } e$
- and  $A \rightarrow B$  denotes  $X \rightarrow Y$  with  $e \mapsto \lambda x. \text{error}'(e)$ .

## $E$ -pointed semantics of CBN types

A CBN type should denote a set  $X$  (the carrier) with some designated elements  $E \xrightarrow{\text{error}} X$ .

This is called an  $E$ -pointed set.

Thus `bool` denotes  $\mathbb{B} + E$  with  $e \mapsto \text{inr } e$ .

If  $\llbracket A \rrbracket = (X, \text{error})$  and  $\llbracket B \rrbracket = (Y, \text{error}')$ ,

- then  $A + B$  denotes  $(X + Y) + E$  with  $e \mapsto \text{inr } e$
- and  $A \rightarrow B$  denotes  $X \rightarrow Y$  with  $e \mapsto \lambda x. \text{error}'(e)$ .

Can we generalize the notion of  $E$ -pointed set to other monads on `Set`?

# Algebras for a Monad

An *Eilenberg-Moore algebra* for a monad  $T$  on **Set** is

- a set  $X$  (the carrier)
- a function  $TX \xrightarrow{\theta} X$  (the structure)

satisfying

$$\begin{array}{ccccc} X & \xrightarrow{\eta^X} & TX & \xleftarrow{\mu^X} & T^2 X \\ & \searrow \text{id} & \downarrow \theta & & \downarrow T\theta \\ & & X & \xleftarrow{\theta} & TX \end{array}$$

An algebra for the  $- + E$  monad is an  $E$ -pointed set.

# Examples of Algebras

An algebra for the  $- + E$  monad is an  $E$ -pointed set.

An algebra for  $\mathcal{A}^* \times -$  is an  $\mathcal{A}$ -set

i.e. a set  $X$  together with a function  $\mathcal{A} \times X \xrightarrow{*} X$ .

This is what we need to interpret

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

If  $B$  denotes  $(X, *)$  then  $\text{print } c. M$  denotes  $\rho \mapsto c * (\llbracket M \rrbracket \rho)$

# 3 Ways Of Building Algebras

## Free Algebras

Given a set  $X$ , the **free  $T$ -algebra** on  $X$  has carrier  $TX$  and structure  $\mu_X$ .

## Product Algebras

Given a family of  $T$ -algebras  $(X_i, \theta_i)$ , the **product algebra**  $\prod_{i \in I} (X_i, \theta_i)$  has carrier  $\prod_{i \in I} X_i$  and structure given pointwise.

## Exponential Algebras

Given a set  $A$  and a  $T$ -algebra  $(X, \theta)$ , the **exponential algebra**  $A \rightarrow (X, \theta)$  has carrier  $A \rightarrow X$  and structure given pointwise.

Let  $T$  be a monad on  $\mathbf{Set}$ .

A type denotes a  $T$ -algebra.

- $\mathbf{bool}$  denotes the free algebra on  $\mathbb{B}$
- If  $\llbracket A \rrbracket = (X, \theta)$  and  $\llbracket B \rrbracket = (Y, \phi)$ 
  - then  $A + B$  denotes the free algebra on  $X + Y$
  - and  $A \rightarrow B$  denotes the exponential algebra  $X \rightarrow (Y, \phi)$ .

# Algebra semantics for CBN terms

Suppose  $B$  denotes the algebra  $(Y, \theta)$ .

Then a term  $\Gamma \vdash M : B$  denotes a function between the **carrier sets**

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} Y .$$

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

This term denotes

$$\begin{array}{ccccc} & \llbracket \Gamma \rrbracket & & & Y \\ & \downarrow \langle \text{id}, \llbracket M \rrbracket \rangle & & & \uparrow \theta \\ \llbracket \Gamma \rrbracket \times T\mathbb{B} & \xrightarrow{t_{\llbracket \Gamma \rrbracket, \mathbb{B}}} & T(\llbracket \Gamma \rrbracket \times \mathbb{B}) & \xrightarrow{T[\llbracket N \rrbracket, \llbracket N' \rrbracket]} & TY \end{array}$$

## Errors

- If  $M \Downarrow T : B$  then  $\llbracket M \rrbracket_{\varepsilon} = \llbracket T \rrbracket_{\varepsilon}$
- If  $M \Downarrow e : B$  then  $\llbracket M \rrbracket_{\varepsilon} = \text{error } e$  where  $\llbracket B \rrbracket = (X, \text{error})$

## Printing

- If  $M \Downarrow m, T : B$  then  $\llbracket M \rrbracket_{\varepsilon} = m ** (\llbracket T \rrbracket_{\varepsilon})$  where  $\llbracket B \rrbracket = (X, *)$

Straightforward inductive proofs using the substitution lemma.

We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

These are instances of a general recipe using a monad  $T$  on **Set** and its algebras.

We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

These are instances of a general recipe using a monad  $T$  on **Set** and its algebras.

A CBV type denotes a set; a CBN type denotes a  $T$ -algebra.

We have a denotational semantics for errors and printing for CBV and CBN, and shown their correctness.

These are instances of a general recipe using a monad  $T$  on **Set** and its algebras.

A CBV type denotes a set; a CBN type denotes a  $T$ -algebra.

They are fundamentally different things.

# Semantics of Types, Again

We write  $F^T X$  for the free  $T$ -algebra  $(TX, \mu_X)$  on  $X$

## Semantics of Types, Again

We write  $F^T X$  for the free  $T$ -algebra  $(TX, \mu_X)$  on  $X$  and  $U^T(X, \theta)$  for the carrier  $X$  of a  $T$ -algebra  $(X, \theta)$ .

# Semantics of Types, Again

We write  $F^T X$  for the free  $T$ -algebra  $(TX, \mu_X)$  on  $X$  and  $U^T(X, \theta)$  for the carrier  $X$  of a  $T$ -algebra  $(X, \theta)$ .

Our CBN semantics of types can be written

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

## Semantics of Types, Again

We write  $F^T X$  for the free  $T$ -algebra  $(TX, \mu_X)$  on  $X$  and  $U^T(X, \theta)$  for the carrier  $X$  of a  $T$ -algebra  $(X, \theta)$ .

Our CBN semantics of types can be written

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

And our CBV semantics of types can be written

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

# Call-By-Push-Value Types

Call-by-push-value has

- **value types** which (like CBV types) denote sets
- **computation types** which (like CBN types) denote  $T$ -algebras.

# Call-By-Push-Value Types

Call-by-push-value has

- **value types** which (like CBV types) denote sets
- **computation types** which (like CBN types) denote  $T$ -algebras.

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

# Call-By-Push-Value Types

Call-by-push-value has

- **value types** which (like CBV types) denote sets
- **computation types** which (like CBN types) denote  $T$ -algebras.

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

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

**Strangely** function types are computation types, and  $\lambda x.M$  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**  $\Gamma$  is a finite set of identifiers with associated **value type**

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

# Judgements

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

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

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

Judgement for a value:  $\Gamma \vdash^v V : A$

Judgement for a computation:  $\Gamma \vdash^c M : \underline{B}$

# Judgements

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

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

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

Judgement for a value:  $\Gamma \vdash^v V : A$

Judgement for a computation:  $\Gamma \vdash^c M : \underline{B}$

- A value  $\Gamma \vdash^v V : A$  denotes a function  $[[\Gamma]] \xrightarrow{[[V]]} [[A]]$
- If  $\underline{B}$  denotes  $(X, \theta)$ , then a computation  $\Gamma \vdash^c M : \underline{B}$  denotes a function  $[[\Gamma]] \xrightarrow{[[M]]} X$ .

**Note** From the viewpoint of monad/algebra semantics, there is no difference between a computation  $\Gamma \vdash^c M : \underline{B}$  and a value  $\Gamma \vdash^v V : U\underline{B}$ .

The type  $FA$ 

A computation in  $FA$  **returns** a value in  $A$ .

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

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

## The type $FA$

A computation in  $FA$  **returns** a value in  $A$ .

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

The denotation of  $M \text{ to } x. N$  uses the structure of  $\llbracket \underline{B} \rrbracket$ .

## The type $U\underline{B}$

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

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

The constructs **thunk** and **force** are inverse.

They are **invisible** in monad/algebra semantics.

An identifier is a value.

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

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

We write **let** to bind an identifier.

$$\frac{\Gamma \vdash^v V : A_{\hat{i}}}{\Gamma \vdash^v \langle \hat{i}, V \rangle : \sum_{i \in I} A_i} \quad \hat{i} \in I$$

$$\frac{\Gamma \vdash^v V : \sum_{i \in I} A_i \quad \Gamma, \mathbf{x} : A_i \vdash^c M_i : \underline{B} \quad (\forall i \in I)}{\Gamma \vdash^c \text{match } V \text{ as } \{\langle i, \mathbf{x} \rangle . M_i\}_{i \in I} : \underline{B}}$$

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

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

The rules for 1 are similar.

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

$$\frac{\Gamma \vdash^c M : A \rightarrow \underline{B} \quad \Gamma \vdash^v V : A}{\Gamma \vdash^c MV : \underline{B}}$$

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

$$\frac{\Gamma \vdash^c M : \prod_{i \in I} \underline{B}_i \quad \hat{i} \in I}{\Gamma \vdash^c M \hat{i} : \underline{B}_{\hat{i}}}$$

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

$$\frac{\Gamma \vdash^c M : A \rightarrow \underline{B} \quad \Gamma \vdash^v V : A}{\Gamma \vdash^c MV : \underline{B}}$$

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

$$\frac{\Gamma \vdash^c M : \prod_{i \in I} \underline{B}_i \quad \hat{i} \in I}{\Gamma \vdash^c M \hat{i} : \underline{B}_{\hat{i}}}$$

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

# Interpreter

The terminals are **computations**:

`return V`      $\lambda x.M$       $\lambda\{i.M_i\}_{i \in I}$

# Interpreter

The terminals are **computations**:

`return V`     `$\lambda x.M$`      `$\lambda\{i.M_i\}_{i \in I}$`

To evaluate

- `return V`: return `return V`.
- `M to x. N`: evaluate `M`. If it returns `return V`, then evaluate `N[V/x]`.
- `$\lambda x.N$` : return  `$\lambda x.N$` .
- `MV`: evaluate `M`. If it returns  `$\lambda x.N$` , evaluate `N[V/x]`.
- `$\lambda\{i.N_i\}_{i \in I}$` : return  `$\lambda\{i.N_i\}_{i \in I}$` .
- `M $\hat{i}$` : evaluate `M`. If it returns  `$\lambda\{i.N_i\}_{i \in I}$` , evaluate `N $\hat{i}$` .
- `let V be x. M`: evaluate `M[V/x]`.
- `force thunk M`: evaluate `M`.
- `match  $\langle \hat{i}, V \rangle$  as  $\{\langle i, x \rangle.M_i\}_{i \in I}$` : evaluate `M $\hat{i}$ [V/x]`.
- `match  $\langle V, V' \rangle$  as  $\langle x, y \rangle.M$` : evaluate `M[V/x, V'/y]`.

# Decomposing CBV into CBPV

A CBV type translates into a value type.

$$A \rightarrow B \mapsto U(A \rightarrow FB)$$

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

$$\begin{array}{lll} x & \mapsto & \text{return } x \\ \lambda x. M & \mapsto & \text{return thunk } \lambda x. M \\ M N & \mapsto & M \text{ to } f. N \text{ to } y. ((\text{force } f) y) \\ \text{let } M \text{ be } x. N & \mapsto & M \text{ to } y. \text{let } y \text{ be } x. N \end{array}$$

# Decomposing CBV into CBPV

A CBV type translates into a value type.

$$A \rightarrow B \mapsto U(A \rightarrow FB)$$

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

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

# Decomposing CBN into CBPV

A CBN type translates into a computation type.

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

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

$$\begin{aligned}x &\mapsto \text{force } x \\ \text{let } M \text{ be } x. N &\mapsto \text{let } (\text{think } M) \text{ be } x. N \\ \lambda x. M &\mapsto \lambda x. M \\ M N &\mapsto M (\text{think } N) \\ \text{inl } M &\mapsto \text{return inl think } M\end{aligned}$$

# Summary

We've seen the call-by-push-value calculus, its operational and monad/algebra semantics.

# Summary

We've seen the call-by-push-value calculus, its operational and monad/algebra semantics.

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

# Summary

We've seen the call-by-push-value calculus, its operational and monad/algebra semantics.

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

Moggi's  $TA$  is  $UFA$ .

We've seen the call-by-push-value calculus, its operational and monad/algebra semantics.

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

Moggi's  $TA$  is  $UFA$ .

But

We've seen the call-by-push-value calculus, its operational and monad/algebra semantics.

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

Moggi's  $TA$  is  $UFA$ .

But

- the monad/algebra semantics makes `thunk` and `force` invisible

We've seen the call-by-push-value calculus, its operational and monad/algebra semantics.

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

Moggi's  $TA$  is  $UFA$ .

But

- the monad/algebra semantics makes `thunk` and `force` invisible
- we still don't understand why a function is a “computation”.

- Landin's ISWIM: CBV  $\lambda$ -calculus with effects. Influenced ML and other languages.
- Plotkin:  $\lambda_v$ -calculus provided equations for CBV with divergence.
- Numerous researchers: CPS transforms for CBV.
- Felleisen *et al*: CBV semantics for various effects.
- Moggi:  $\lambda_c$ -calculus, monads for CBV and monadic metalanguage.
- Power and Robinson: Freyd categories for CBV.
- Benton and Kennedy: MIL-lite.
- Thielecke: **thunk** and **force** constructs in CBV with `callcc`.

## Antecedents: CBN without $\eta$ -law for functions

- Plotkin: CPS transform for CBN without  $\eta$ -law.
- Abramsky and Ong: untyped lazy  $\lambda$ -calculus.
- Ong: typed lazy  $\lambda$ -calculus.
- Moggi: monads for CBN without  $\eta$ -law.

## Antecedents: CBN with $\eta$ -law for functions

- Plotkin's PCF, a CBN calculus for recursion.
- Hennessy and Ashcroft: recursion and nondeterminism.
- O'Hearn: semantics of conditional in Reynolds' Idealized Algol.
- Streicher and Reus: semantics of control effects in CBN.
- $\neg\neg$  translations of classical logic. Also CBV.
- Game semantics. Also CBV.

# Calculi combining CBV and CBN

- Various calculi based on CPS.
- Filinski's Effect-PCF provided the CBPV **to** construct. *U is implicit.*
- Howard's  $\lambda^{\mu\nu\perp}$  calculus for recursion. *U is implicit.*
- Egger, Møgelberg and Simpson's Effect Calculus emphasizes connection to Benton's Linear/Nonlinear Logic. *U is implicit.*
- Marz' SFPL for recursion and sequentiality. *F is implicit.*
- Nygaard and Winskel's HOPLA for recursion and nondeterminism. *F is implicit.*
- Laurent's LLP has extra type constructors not included in CBPV.
- Harper, Licata and Zeilberger's Polarized Intuitionistic Logic.

An operational semantics due to Felleisen and Friedman (1986).

And Landin, Krivine, Streicher and Reus, Bierman, Pitts, ...

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

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

Initially, K is empty.

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

# Transitions for sequencing

To evaluate  $M$  to  $x$ .  $N$ : evaluate  $M$ . If it returns **return**  $V$ , then evaluate  $N[V/x]$ .

$$\begin{array}{l} M \text{ to } x. N \\ M \end{array} \quad \begin{array}{l} K \\ \text{to } x. N :: K \end{array} \quad \rightsquigarrow$$

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

# Transitions for application

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

$V'M$	$K$	$\rightsquigarrow$
$M$	$V :: K$	

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

# Those function rules again

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

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

# Those function rules again

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

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

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

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

# Those function rules again

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

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

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

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

Revisiting some equations:

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

# Values and Computations

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

- A value of type  $UB$  **is** a thunk of a computation of type  $\underline{B}$ .
- A value of type  $\sum_{i \in I} A_i$  **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  $FA$  **returns** a value of type  $A$ .
- A computation of type  $A \rightarrow \underline{B}$   
    **pops** a value of type  $A$   
    then **behaves** in  $\underline{B}$ .
- A computation of type  $\prod_{i \in I} \underline{B}_i$   
    **pops** a tag  $i \in I$   
    then **behaves** 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

```
print "hello0".
let 3 be x.
let think (
    print "hello1".
    λz.
    print "we just popped "z.
    return x + z
) be y.
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

# Typing the CK-machine

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

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

Transitions

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

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

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

# Typing the CK-machine

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

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

Transitions

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

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

Typically  $\Gamma$  would be empty and  $\underline{C} = F \text{ bool}$ . We write  $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$  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

$$\frac{}{\Gamma \mid \underline{C} \vdash^k \text{nil} : \underline{C}}$$

$$\frac{\Gamma \mid \underline{B}_{\hat{i}} \vdash^k K : \underline{C}}{\Gamma \mid \prod_{i \in I} \underline{B}_i \vdash^k \hat{i} :: K : \underline{C}} \quad \hat{i} \in I$$

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

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

# Typing rules, read off from the CK-machine

## Typing a stack

$$\frac{}{\Gamma \mid \underline{C} \vdash^k \text{nil} : \underline{C}}$$

$$\frac{\Gamma \mid \underline{B}_{\hat{i}} \vdash^k K : \underline{C}}{\Gamma \mid \prod_{i \in I} \underline{B}_i \vdash^k \hat{i} :: K : \underline{C}} \quad \hat{i} \in I$$

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

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

## Typing a CK-configuration

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

## Continuations

A **continuation** is a stack from an  $F$  type. For example:

$$\Gamma \mid FA \vdash^k \text{to } x. M :: K : \underline{C}$$

It describes what happens next once it receives a value.

## Continuations

A **continuation** is a stack from an  $F$  type. For example:

$$\Gamma \mid FA \vdash^k \text{to } x. M :: K : \underline{C}$$

It describes what happens next once it receives a value.

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

## Continuations

A **continuation** is a stack from an  $F$  type. For example:

$$\Gamma \mid FA \vdash^k \text{to } x. M :: K : \underline{C}$$

It describes what happens next once it receives a value.

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

## Top-Level Stack

The **top-level stack** is

$$\Gamma \mid \underline{C} \vdash^k \text{nil} : \underline{C}$$

and  $\underline{C}$  is the **top-level type**.

## Continuations

A **continuation** is a stack from an  $F$  type. For example:

$$\Gamma \mid FA \vdash^k \text{to } x. M :: K : \underline{C}$$

It describes what happens next once it receives a value.

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

## Top-Level Stack

The **top-level stack** is

$$\Gamma \mid \underline{C} \vdash^k \text{nil} : \underline{C}$$

and  $\underline{C}$  is the **top-level type**.

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

# Denotational semantics of stacks

Suppose  $\llbracket \underline{C} \rrbracket = (Y, \phi)$ . The behaviour of  $\Gamma \vdash^{\text{ck}} (M, K) : \underline{C}$  depends on the environment:

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (M, K) \rrbracket} Y$$

to be preserved by each transition.

# Denotational semantics of stacks

Suppose  $\llbracket \underline{C} \rrbracket = (Y, \phi)$ . The behaviour of  $\Gamma \vdash^{\text{ck}} (M, K) : \underline{C}$  depends on the environment:

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (M, K) \rrbracket} Y$$

to be preserved by each transition.

Suppose also  $\llbracket \underline{B} \rrbracket = (X, \theta)$ . A stack  $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$  transforms computations to CK-configurations. So we get a function

$$\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y$$

homomorphic in its second argument.

# Denotational semantics of stacks

Suppose  $\llbracket \underline{C} \rrbracket = (Y, \phi)$ . The behaviour of  $\Gamma \vdash^{\text{ck}} (M, K) : \underline{C}$  depends on the environment:

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (M, K) \rrbracket} Y$$

to be preserved by each transition.

Suppose also  $\llbracket \underline{B} \rrbracket = (X, \theta)$ . A stack  $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$  transforms computations to CK-configurations. So we get a function

$$\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y$$

homomorphic in its second argument.

because if  $M$  raises an error or prints, then so does  $M, K$ .

# Denotational semantics of stacks

Suppose  $\llbracket \underline{C} \rrbracket = (Y, \phi)$ . The behaviour of  $\Gamma \vdash^{\text{ck}} (M, K) : \underline{C}$  depends on the environment:

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (M, K) \rrbracket} Y$$

to be preserved by each transition.

Suppose also  $\llbracket \underline{B} \rrbracket = (X, \theta)$ . A stack  $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$  transforms computations to CK-configurations. So we get a function

$$\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y$$

homomorphic in its second argument.

because if  $M$  raises an error or prints, then so does  $M, K$ .

We assume there's no exception handling.

# Adjunction between values and stacks

We have an adjunction between the category of values ([sets and functions](#)) and the category of stacks ([T-algebras and homomorphisms](#)).

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

This resolves the monad  $T$  on  $\mathbf{Set}$ .

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

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

$$\frac{\Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^c \mathbf{l} := n. M : \underline{B}} \quad n \in \mathbb{N}$$

$$\frac{\Gamma \vdash^c M_n : \underline{B} \quad (\forall n \in \mathbb{N})}{\Gamma \vdash^c \text{read } \mathbf{l} \text{ as } \{n. M_n\}_{n \in \mathbb{N}} : \underline{B}}$$

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

$$\frac{\Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^c \mathbf{l} := n. M : \underline{B}} \quad n \in \mathbb{N}$$

$$\frac{\Gamma \vdash^c M_n : \underline{B} \quad (\forall n \in \mathbb{N})}{\Gamma \vdash^c \text{read } \mathbf{l} \text{ as } \{n. M_n\}_{n \in \mathbb{N}} : \underline{B}}$$

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

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

# Big-step semantics for state

The big-step semantics takes the form  $s, M \Downarrow s', T$ .

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

We can type these using

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

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

We can take algebras for this and obtain a denotational semantics of CBPV with state.

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

We can take algebras for this and obtain a denotational semantics of CBPV with state.

But it doesn't fit well with SC-configurations.

We'd like a soundness result of the following form:

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

This requires an SC-configuration to have a denotation.

# Semantics of SC-configurations

Value type  $A$  denotes the set of denotations of values of type  $A$ . Like in monad semantics.

# Semantics of SC-configurations

Value type  $A$  denotes the set of **denotations of** values of type  $A$ . **Like in monad semantics.**

Computation type  $\llbracket \underline{B} \rrbracket$  denotes the set of behaviours of configurations of type  $\underline{B}$ .

# Semantics of SC-configurations

Value type  $A$  denotes the set of **denotations of values** of type  $A$ . Like in **monad semantics**.

Computation type  $\llbracket \underline{B} \rrbracket$  denotes the set of behaviours of configurations of type  $\underline{B}$ .

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

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (s, M) \rrbracket} \llbracket \underline{B} \rrbracket$$

# Semantics of SC-configurations

Value type  $A$  denotes the set of **denotations of values** of type  $A$ . **Like in monad semantics.**

Computation type  $\llbracket \underline{B} \rrbracket$  denotes the set of behaviours of configurations of type  $\underline{B}$ .

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

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (s, M) \rrbracket} \llbracket \underline{B} \rrbracket$$

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

$$S \times \llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket \underline{B} \rrbracket$$

## State: semantics of types

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

$$\llbracket FA \rrbracket = S \times \llbracket A \rrbracket$$

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

$$\llbracket A \rightarrow \underline{B} \rrbracket = \llbracket A \rrbracket \rightarrow \llbracket \underline{B} \rrbracket$$

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

$$\llbracket \prod_{i \in I} \underline{B}_i \rrbracket = \prod_{i \in I} \llbracket \underline{B}_i \rrbracket$$

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

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

## State: semantics of types

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

$$\llbracket FA \rrbracket = S \times \llbracket A \rrbracket$$

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

$$\llbracket A \rightarrow \underline{B} \rrbracket = \llbracket A \rrbracket \rightarrow \llbracket \underline{B} \rrbracket$$

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

$$\llbracket \prod_{i \in I} \underline{B}_i \rrbracket = \prod_{i \in I} \llbracket \underline{B}_i \rrbracket$$

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

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

We recover standard semantics for CBV,  
and O'Hearn's semantics for CBN.

# The SCK-machine

We replace  $\vdash^{\text{ck}}$  with  $\vdash^{\text{sck}}$ .

$$\frac{\Gamma \vdash^{\text{c}} M : \underline{B} \quad \Gamma \mid \underline{B} \vdash^{\text{k}} K : \underline{C}}{\Gamma \vdash^{\text{sck}} (s, M, K) : \underline{C}} \quad s \in \mathcal{S}$$

# The SCK-machine

We replace  $\vdash^{\text{ck}}$  with  $\vdash^{\text{sck}}$ .

$$\frac{\Gamma \vdash^{\text{c}} M : \underline{B} \quad \Gamma \mid \underline{B} \vdash^{\text{k}} K : \underline{C}}{\Gamma \vdash^{\text{sck}} (s, M, K) : \underline{C}} \quad s \in \mathcal{S}$$

The behaviour of an SCK-configuration  $\Gamma \vdash^{\text{sck}} (s, M, K) : \underline{C}$  depends on the environment:

$$\llbracket \Gamma \rrbracket \xrightarrow{\llbracket (s, M, K) \rrbracket} \llbracket \underline{C} \rrbracket$$

to be preserved by each transition.

# The SCK-machine

We replace  $\vdash^{\text{ck}}$  with  $\vdash^{\text{sck}}$ .

$$\frac{\Gamma \vdash^{\text{c}} M : \underline{B} \quad \Gamma \mid \underline{B} \vdash^{\text{k}} K : \underline{C}}{\Gamma \vdash^{\text{sck}} (s, M, K) : \underline{C}} \quad s \in S$$

The behaviour of an SCK-configuration  $\Gamma \vdash^{\text{sck}} (s, M, K) : \underline{C}$  depends on the environment:

$$[[\Gamma]] \xrightarrow{[(s, M, K)]} [[\underline{C}]]$$

to be preserved by each transition.

A stack  $\Gamma \mid \underline{B} \vdash^{\text{k}} K : \underline{C}$  transforms SC-configuration behaviours to SCK-configuration behaviours:

$$[[\Gamma]] \times [[\underline{B}]] \xrightarrow{[[K]]} [[\underline{C}]]$$

## State: the value/stack adjunction

We've seen that a stack  $\mid \underline{B} \vdash^k K : \underline{C}$  denotes a function  $\llbracket \underline{B} \rrbracket \xrightarrow{\llbracket K \rrbracket} \llbracket \underline{C} \rrbracket$ .

# State: the value/stack adjunction

We've seen that a stack  $\mid \underline{B} \vdash^k K : \underline{C}$  denotes a function  $\llbracket \underline{B} \rrbracket \xrightarrow{\llbracket K \rrbracket} \llbracket \underline{C} \rrbracket$ .

We have an adjunction

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

between values and stacks.

# Catching and throwing a stack

We extend CBPV with Crolard's instructions for changing the stack.

- `catch  $\alpha$`  means “let  $\alpha$  be the current stack”.
- `throw  $K$`  means “change the current stack to  $K$ ”.

$\Gamma$	<code>catch <math>\alpha</math>. <math>M</math></code>	$\underline{B}$	$K$	$\Delta$	$\rightsquigarrow$
$\Gamma$	<code><math>M[K/\alpha]</math></code>	$\underline{B}$	$K$	$\Delta$	

$\Gamma$	<code>throw <math>K</math>. <math>M</math></code>	$\underline{B}'$	$K'$	$\Delta$	$\rightsquigarrow$
$\Gamma$	<code><math>M</math></code>	$\underline{B}$	$K$	$\Delta$	

# Typing judgements for control

The **stack context**  $\Delta$  consists of stack names  $\alpha : \underline{B}$ .

$\vdash^n$  indicates “no top-level type”.

value	$\Gamma \vdash^v V : A \mid \Delta$	
computation	$\Gamma \vdash^c M : \underline{B} \mid \Delta$	
stack	$\Gamma \mid \underline{B} \vdash^{nk} K \mid \Delta$	$\Gamma \mid \underline{B} \vdash^k \alpha. K : \underline{C} \mid \Delta$
CK-configuration	$\Gamma \vdash^{nck} (M, K) \mid \Delta$	$\Gamma \vdash^{ck} \alpha. (M, K) : \underline{C} \mid \Delta$

## Creating nil at start of evaluation

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

$\Gamma$	$P$	$\underline{C}$	nil	$\Delta, \text{nil} : \underline{C}$
----------	-----	-----------------	-----	--------------------------------------

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

Fix a set  $R$ , the set of behaviours of CK-configurations.

Fix a set  $R$ , the set of behaviours of CK-configurations.

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

Fix a set  $R$ , the set of behaviours of CK-configurations.

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

**Maybe** we can use algebras for this to build a denotational semantics of control.

# Semantics of control using stacks

**Informally** a computation type  $\llbracket \underline{B} \rrbracket$  denotes the set of stacks from  $\underline{B}$ .

# Semantics of control using stacks

**Informally** a computation type  $\llbracket B \rrbracket$  denotes the set of stacks from  $\underline{B}$ .

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

$$\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \times \llbracket \underline{B} \rrbracket \xrightarrow{\llbracket M \rrbracket} R$$

# Semantics of control using stacks

**Informally** a computation type  $\llbracket \underline{B} \rrbracket$  denotes the set of stacks from  $\underline{B}$ .

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

$$\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \times \llbracket \underline{B} \rrbracket \xrightarrow{\llbracket M \rrbracket} R$$

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

$$\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \xrightarrow{\llbracket V \rrbracket} \llbracket A \rrbracket$$

A stack  $\Gamma \mid \underline{B} \vdash^{\text{nk}} K \mid \Delta$  denotes

$$\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \xrightarrow{\llbracket K \rrbracket} \llbracket \underline{B} \rrbracket$$

A CK-configuration  $\Gamma \vdash^{\text{nck}} \text{nil}.(M, K) : \Delta$  denotes

$$\llbracket \Gamma \rrbracket \times \llbracket \Delta \rrbracket \xrightarrow{\llbracket (M, K) \rrbracket} R$$

to be preserved by each transition.

## Control: semantics of types

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

$$\llbracket FA \rrbracket = \llbracket A \rrbracket \rightarrow R$$

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

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

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

$$\llbracket \prod_{i \in I} \underline{B}_i \rrbracket = \sum_{i \in I} \llbracket \underline{B}_i \rrbracket$$

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

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

## Control: semantics of types

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

$$\llbracket FA \rrbracket = \llbracket A \rrbracket \rightarrow R$$

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

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

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

$$\llbracket \prod_{i \in I} \underline{B}_i \rrbracket = \sum_{i \in I} \llbracket \underline{B}_i \rrbracket$$

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

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

We recover standard continuation semantics for CBV, and Streicher and Reus' semantics for CBN.

# Control: the value/stack adjunction

A stack with top-level type

$$\Gamma \mid \underline{B} \vdash^k \alpha. K : \underline{C} \mid \Delta$$

denotes a function

$$[[\Gamma]] \times [[\Delta]] \times [[\underline{C}]] \xrightarrow{[[\alpha. K]]} [[\underline{B}]]$$

# Control: the value/stack adjunction

A stack with top-level type

$$\Gamma \mid \underline{B} \vdash^k \alpha. K : \underline{C} \mid \Delta$$

denotes a function

$$[[\Gamma]] \times [[\Delta]] \times [[\underline{C}]] \xrightarrow{[[\alpha. K]]} [[\underline{B}]]$$

So we have an adjunction

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

between values and stacks with top-level type.

# Summary of models

For every monad  $T$  on **Set** we have an adjunction

$$\mathbf{Set} \begin{array}{c} \xrightarrow{F^T} \\ \xleftarrow[U^T]{\perp} \\ \end{array} \mathbf{Set}^T$$

This is useful for modelling CBPV with errors and printing.

# Summary of models

For every monad  $T$  on  $\mathbf{Set}$  we have an adjunction

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

This is useful for modelling CBPV with errors and printing.

For a set  $S$  we have an adjunction

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

This is useful for modelling CBPV with state.

# Summary of models

For every monad  $T$  on  $\mathbf{Set}$  we have an adjunction

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

This is useful for modelling CBPV with errors and printing.

For a set  $S$  we have an adjunction

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

This is useful for modelling CBPV with state.

For a set  $R$  we have an adjunction

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

This is useful for modelling CBPV with control.

# Summary of models

For every monad  $T$  on  $\mathbf{Set}$  we have an adjunction

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

This is useful for modelling CBPV with errors and printing.

For a set  $S$  we have an adjunction

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

This is useful for modelling CBPV with state.

For a set  $R$  we have an adjunction

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

This is useful for modelling CBPV with control.