

A tutorial on call-by-push-value

Paul Blain Levy

University of Birmingham

December 19, 2007

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

Typed λ -calculus

We consider typed λ -calculus with boolean, function and sum types.

Types

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

Typing judgement $\Gamma \vdash M : B$

Terms

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

Equational Laws

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

η -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 λ -abstraction.

η -law for `bool`

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

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

Anything of boolean type is a boolean.

The η -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 $\Gamma \vdash N : B$

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 β and η 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 $N[M/x]$.
- `pm M as {true.N, false.N'}`, evaluate M . If it returns `true`, evaluate N , but if it returns `false`, evaluate N' .
- `pm 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 $\lambda x.M$.
- $\text{inl } M$, evaluate M . If it returns T , return $\text{inl } T$.
- $\text{let } M \text{ be } x. N$, evaluate M . If it returns T , evaluate $N[T/x]$.
- $\text{pm } M \text{ as } \{\text{true}.N, \text{false}.N'\}$, evaluate M . If it returns true , evaluate N , but if it returns false , evaluate N' .
- $\text{pm } M \text{ as } \{\text{inl } x.N, \text{inr } x.N'\}$, evaluate M . If it returns $\text{inl } T$, evaluate $N[T/x]$, but if it returns $\text{inr } T$, 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} be a set of **characters**. We add

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

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

- 1 Evaluate

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

in CBV and CBN

- 2 Evaluate

```
(λx.(x + x))(print "hello". 4)
```

in CBV and CBN.

- 3 Evaluate

```
pm (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

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

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

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 η -law for boolean type: has it survived?

η -law for bool

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

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

Anything of boolean type is a boolean.

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

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

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

because we can apply the context

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

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 $A \rightarrow B$

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 β - and η -laws.

With computational effects,

- CBV satisfies η for boolean and sum types, but not function types
- CBN satisfies η for function types, but not boolean and sum types.

We want denotational semantics that validate the appropriate η -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.**

For example, define $z : \text{bool} \vdash M, M' : \text{bool}$ and $\vdash N : \text{bool}$

$$\begin{aligned} M &\stackrel{\text{def}}{=} \text{true} \\ M' &\stackrel{\text{def}}{=} \text{pm } z \text{ as } \{\text{true. true, false. false}\} \\ N &\stackrel{\text{def}}{=} \text{error CRASH} \end{aligned}$$

Then we want

$$\begin{aligned} \llbracket M \rrbracket &= \llbracket M' \rrbracket \\ \llbracket M[N/x] \rrbracket &\neq \llbracket M'[N/x] \rrbracket \end{aligned}$$

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

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

Errors

- If $M \Downarrow V$ then $\llbracket M \rrbracket_{\varepsilon} = \text{inl} (\llbracket V \rrbracket^{\text{val}}_{\varepsilon})$.
- If $M \Downarrow e$ 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 $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket B \rrbracket$.

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

denotes $\rho \mapsto ?$

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

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

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

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

E -set 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 -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 -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

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

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

Examples of Algebras

An algebra for the $- + E$ monad is an E -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

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

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 μ_X .

Product Algebras

Given a family of T -algebras (X_i, θ_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, θ) , the **exponential algebra** $A \rightarrow (X, \theta)$ has carrier $A \rightarrow X$ and structure given pointwise.

Let T be a monad on **Set**.

A type denotes a T -algebra.

- `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

A term $x : A, x' : A' \vdash M : B$ denotes a function between the **carrier sets**

$$X \times X' \xrightarrow{\llbracket M \rrbracket} Y.$$

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

If B denotes (Y, θ) then this term denotes

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

Errors

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

Printing

- If $M \Downarrow m, T : B$ then $\llbracket M \rrbracket_{\mathcal{E}} = m ** (\llbracket T \rrbracket_{\mathcal{E}})$ 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, μ_X) on X

Semantics of Types, Again

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

Semantics of Types, Again

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

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, μ_X) on X and $U^T(X, \theta)$ for the carrier X of a T -algebra (X, θ) .

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

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

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.

We underline computation types.

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.

We underline computation types.

value types $A ::= U\underline{B} \mid \sum_{i \in I} A_i \mid 1 \mid A \times A$

computation types $\underline{B} ::= FA \mid \prod_{i \in I} \underline{B}_i \mid A \rightarrow \underline{B}$

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.

We underline computation types.

value types $A ::= \underline{UB} \mid \sum_{i \in I} A_i \mid 1 \mid A \times A$

computation types $B ::= FA \mid \prod_{i \in I} \underline{B}_i \mid A \rightarrow \underline{B}$

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** Γ is a finite set of identifiers with associated **value type**

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

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**

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

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** Γ is a finite set of identifiers with associated **value type**

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

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 $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket V \rrbracket} \llbracket A \rrbracket$
- If \underline{B} denotes (X, θ) , then a computation $\Gamma \vdash^c M : \underline{B}$ denotes a function $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} X$.

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

The type FA

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

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

This follows Moggi and Filinski. `to` 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.

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

We write `let` to bind an identifier.

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

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

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

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

The rules for 1 are similar.

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

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

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

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

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

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

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

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

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

The terminals are **computations**:

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

To evaluate

- $\text{return } V$, return $\text{return } V$.
- M to x . N , evaluate M . If it returns $\text{return } V$, 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 $N_{\hat{i}}$.
- $\text{let } V \text{ be } x. M$, evaluate $M[V/x]$.
- $\text{force thunk } M$, evaluate M .
- $\text{pm } \langle \hat{i}, V \rangle$ as $\{\langle i, x \rangle.M_i\}_{i \in I}$, evaluate $M_{\hat{i}}[V/x]$.
- $\text{pm } \langle V, V' \rangle$ 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 $x : A, y : B \vdash^c M : FC$.

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

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 $x : A, y : B \vdash^c M : FC$.

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

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 $x : U\underline{A}, y : U\underline{B} \vdash^c M : \underline{B}$.

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

We've seen the CBPV calculus, its operational and monad/algebra semantics.

We've seen the CBPV calculus, its operational and monad/algebra semantics.

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

We've seen the CBPV 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 CBPV calculus, its operational and monad/algebra semantics.

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

Moggi's *TA* is *UFA*.

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

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

It can be used for CBV, CBN and CBPV.

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

Initially and finally, K is the empty stack **nil**.

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

Transitions for sequencing

To evaluate M to x . N , first evaluate M . If this returns the terminal return V , then evaluate $N[V/x]$.

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

Transitions for application

To evaluate $V'M$, first evaluate M . If this returns the terminal $\lambda x.N$, then evaluate $N[V/x]$.

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

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

Those function rules again

$$\begin{array}{l} V'M \\ M \end{array} \qquad \begin{array}{l} K \\ V :: K \end{array} \qquad \rightsquigarrow$$

$$\begin{array}{l} \lambda x.N \\ N[V/x] \end{array} \qquad \begin{array}{l} V :: K \\ K \end{array} \qquad \rightsquigarrow$$

Those function rules again

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

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

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

We can read λx as an instruction “pop x ”.

Those function rules again

$$\frac{V' M}{M} \quad K \quad \rightsquigarrow \quad V :: K$$

$$\frac{\lambda x. N}{N[V/x]} \quad V :: K \quad \rightsquigarrow \quad K$$

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

We can read λx as an instruction “pop x ”.

Revisiting some equations:

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

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 in 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 .

Example program of type F_{nat}

```
print "hello0".
let 3 be x.
let thunk (
    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

Initial Configuration

M C nil C

Transitions

M to x. N B K C \rightsquigarrow
 M FA to x. $N :: K$ C

return V FA to x. $N :: K$ C \rightsquigarrow
 $N[V/x]$ B K C

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

Typing the CK-machine

Initial Configuration

M C nil C

Transitions

M to x. N B K C \rightsquigarrow
 M FA to x. $N :: K$ C

return V FA to x. $N :: K$ C \rightsquigarrow
 $N[V/x]$ B K C

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

More generally $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ when there are free identifiers.

The Stack Judgement

The typing rules can be read off from the CK-machine transitions.

Typing Rules For Stacks

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

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

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

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

The Stack Judgement

The typing rules can be read off from the CK-machine transitions.

Typing Rules For Stacks

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

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

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

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

A stack from an F type is often called a **continuation**.

Denotational semantics of stacks

If $\llbracket \underline{B} \rrbracket = (X, \theta)$ and $\llbracket \underline{C} \rrbracket = (Y, \phi)$

then a stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ denotes a function $\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y$
homomorphic in its second argument.

Concatenation of stacks corresponds to composition of homomorphisms.

Denotational semantics of stacks

If $\llbracket \underline{B} \rrbracket = (X, \theta)$ and $\llbracket \underline{C} \rrbracket = (Y, \phi)$

then a stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ denotes a function $\llbracket \Gamma \rrbracket \times X \xrightarrow{\llbracket K \rrbracket} Y$
homomorphic in its second argument.

Concatenation of stacks corresponds to composition of homomorphisms.

We have an adjunction between the category of values (semantically: sets and functions) and the category of stacks (semantically: 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 2 storage cells: `fred` stores a natural number and `mary` stores a boolean.

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

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

A state is `fred` \mapsto n , `mary` \mapsto b .

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

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

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

Formally, we define a judgement $\Gamma \vdash^{\text{sc}} P : \underline{B}$ with formation rule

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

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.

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} .

Thus an SC-configuration $\Gamma \vdash^{\text{sc}} P : \underline{B}$ denotes a function $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket P \rrbracket} \llbracket \underline{B} \rrbracket$.

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} .

Thus an SC-configuration $\Gamma \vdash^{\text{sc}} P : \underline{B}$ denotes a function $\llbracket \Gamma \rrbracket \xrightarrow{\llbracket P \rrbracket} \llbracket \underline{B} \rrbracket$.

The behaviour of a computation $\Gamma \vdash^c M : \underline{B}$ depends on state and environment. So $\Gamma \vdash^c M : \underline{B}$ denotes a function $S \times \llbracket \Gamma \rrbracket \xrightarrow{\llbracket M \rrbracket} \llbracket \underline{B} \rrbracket$.

In particular, the configuration s, M denotes $\rho \mapsto \llbracket M \rrbracket(s, \rho)$.

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.

State: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ can be applied to an SC-configuration giving another SC-configuration.

State: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ can be applied to an SC-configuration giving another SC-configuration.

Accordingly it denotes a function $[[\Gamma]] \times [[\underline{B}]] \xrightarrow{[[K]]} [[\underline{C}]]$.

State: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ can be applied to an SC-configuration giving another SC-configuration.

Accordingly it denotes a function $[[\Gamma]] \times [[\underline{B}]] \xrightarrow{[[K]]} [[\underline{C}]]$.

Concatenation of stacks corresponds to composition of functions.

State: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ can be applied to an SC-configuration giving another SC-configuration.

Accordingly it denotes a function $[[\Gamma]] \times [[\underline{B}]] \xrightarrow{[[K]]} [[\underline{C}]]$.

Concatenation of stacks corresponds to composition of functions.

So we have an adjunction

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

Control Operators

Extend CBPV with two instructions for changing the stack:

- `letstk x` means “let x be the current stack”
- `changestk V` means “change the current stack to V ”.

A stack K can now be turned into a value `sv K` .

<code>letstk x. M</code>	<u>B</u>	K	<u>C</u>	\rightsquigarrow
<code>M[<code>sv K/x</code>]</code>	<u>B</u>	K	<u>C</u>	

<code>changestk sv K. M</code>	<u>B'</u>	K	<u>C</u>	\rightsquigarrow
<code>M</code>	<u>B</u>	K	<u>C</u>	

Typing rules

We need a new kind of value type:

$$\begin{aligned} A &::= \underline{UB} \mid \sum_{i \in I} A_i \mid 1 \mid A \times A \mid \text{stk } \underline{B} \\ \underline{B} &::= \underline{FA} \mid \prod_{i \in I} \underline{B}_i \mid A \rightarrow \underline{B} \end{aligned}$$

A value of type $\text{stk } \underline{B}$ is a stack from \underline{B} . (The target type is fixed within a given term.)

Typing rules for control operators

$$\frac{\Gamma, x : \text{stk } \underline{B} \vdash^c M : \underline{B}}{\Gamma \vdash^c \text{letstk } x. M : \underline{B}} \qquad \frac{\Gamma \vdash^v V : \text{stk } \underline{B} \quad \Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^c \text{changestk } V. M : \underline{B}'}$$

We have to treat `nil` as a free identifier:

$$\frac{\Gamma \mid \underline{B} \vdash^k K : \underline{C}}{\Gamma, \text{nil} : \text{stk } \underline{C} \vdash^k \text{sv } K : \text{stk } \underline{B}}$$

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.

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

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 stacks from \underline{B} .

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 stacks from \underline{B} .

Thus we will have $\llbracket \text{stk } \underline{B} \rrbracket = \llbracket \underline{B} \rrbracket$.

Semantics of control using stacks

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 stacks from \underline{B} .

Thus we will have $\llbracket \text{stk } \underline{B} \rrbracket = \llbracket \underline{B} \rrbracket$.

The behaviour of a computation $\Gamma \vdash^c M : \underline{B}$ depends on environment and stack, so it denotes $\llbracket \Gamma \rrbracket \times \llbracket \underline{B} \rrbracket \xrightarrow{\llbracket M \rrbracket} R$.

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.

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

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 UB can be forced alongside any stack K , giving a configuration.

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

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

Control: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk } \underline{C} \vdash^v V : \text{stk } \underline{B}$$

Control: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk } \underline{C} \vdash^v V : \text{stk } \underline{B}$$

Accordingly it denotes a function $[[\Gamma]] \times [[\underline{C}]] \xrightarrow{[[K]]} [[\underline{B}]]$.

Control: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk } \underline{C} \vdash^v V : \text{stk } \underline{B}$$

Accordingly it denotes a function $[[\Gamma]] \times [[\underline{C}]] \xrightarrow{[[K]]} [[\underline{B}]]$.

Concatenation of stacks corresponds to composition of “op-functions”.

Control: the value/stack adjunction

A stack $\Gamma \mid \underline{B} \vdash^k K : \underline{C}$ corresponds to a value

$$\Gamma, \text{nil} : \text{stk } \underline{C} \vdash^v V : \text{stk } \underline{B}$$

Accordingly it denotes a function $[[\Gamma]] \times [[\underline{C}]] \xrightarrow{[[K]]} [[\underline{B}]]$.

Concatenation of stacks corresponds to composition of “op-functions”.

So we have an adjunction

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

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 **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 **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 **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.