Compiling with Call-by-push-value

Max S. New

University of Michigan

MFPS 2023

Special Session on Semantics and Compilers

Overview of CBPV

Paul Blain Levy introduced **Call-by-push-value** as a *subsuming paradigm* for effectful computation

$$
\begin{tikzcd}
\mathrm{CBV} \arrow[dr] & & \mathrm{CBN} \arrow[dl] \\
& \mathrm{CBPV} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Domain Theory} & \text{Presheaves} & \text{Op. Semantics}
\end{tikzcd}
$$

- Preserves equational theories

Overview of CBPV

Paul Blain Levy introduced **Call-by-push-value** as a *subsuming paradigm* for effectful computation

$$
\begin{tikzcd}
\text{CBV} \arrow[dr] & & \text{CBN} \arrow[dl] \\
& \text{CBPV} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Domain Theory} & \text{Presheaves} & \text{Op. Semantics}
\end{tikzcd}
$$

- Preserves equational theories
- Observation: *Denotational models* of CBV/CBN *naturally* decompose into CBPV structure. Semantics of CBPV is *easier* even though it's *more general*

Intermediate Representations

$$
\begin{tikzcd}
\text{Haskell} \arrow[dr] & \text{C++} \arrow[d] & \text{Rust} \arrow[dl] \\
& \text{LLVM} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{x86} & \text{ARM} & \text{WASM}
\end{tikzcd}
$$

Language Platforms

$$
\begin{tikzcd}
\text{Java} \arrow[dr] & \text{Scala} \arrow[d] & \text{Clojure} \arrow[dl] \\
& \text{JVM Bytecode} \arrow[dl] \arrow[dr] & \\
\text{JVM} & & \text{Web Browsers}
\end{tikzcd}
$$

Compare: Racket, .NET,

CBPV as an IR or Language Platform?

1. As an IR: CBPV structure arises in compilation

CBPV as an IR or Language Platform?

1. As an IR: CBPV structure arises in compilation
2. CBV, CBN embeddings in CBPV preserve and reflect equational theories:
   Foundation for a language platform for *verified* language implementations that preserve *reasoning* (equality, logics) not just *whole-program behavior*?

Outline

1. Call-by-push-value Overview
2. CBPV subsumes Functional IRs
   - CBPV subsumes ANF, MNF
   - Stack-Passing Style subsumes CPS
3. Equality-Preserving Compiler Passes in CBPV/SPS
   - Polymorphic Closure Conversion
   - Polymorphic CPS Conversion
4. Computation/Stack Types in Compilation
   - Calling Conventions as Types
   - Relative Monads
5. Future Work

Call-by-push-value Overview

Basics of CBPV

Refine Moggi's analysis of effects using monads in terms of *adjunctions*
Effectful computation naturally involves two *kinds* of types:

1. Value types: the types of pure data, first-class values
2. Computation types: the types of effectful computations

Basics of CBPV

Refine Moggi's analysis of effects using monads in terms of *adjunctions*
Effectful computation naturally involves two *kinds* of types:

1. Value types: the types of pure data, first-class values
2. Computation types: the types of effectful computations

Three notions of term

1. Pure functions $\Gamma \vdash V : A$
2. Effectful functions $\Gamma \vdash M : B$
3. Linear functions aka "Stacks" $\Gamma \mid z : B \vdash L : B'$

Basics of CBPV

Value Types, Values

$$\begin{array}{rl} \mathrm{A,A'} ::= & UB \mid \mathrm{Bool} \\ \mathrm{V,\ V'} ::= & x \mid \mathrm{thunk} M \\ & \mathrm{true} \mid \mathrm{false} \\ & (V, V') \mid V.\pi_i \end{array}$$

A value *is*

- A UB is a "thUnked" B
- A Bool is either true or false

Computation Types, Computations

$$\begin{array}{rl} \mathrm{B,B'} ::= & FA \mid A \to B \\ \mathrm{M,M'} ::= & z \mid \mathrm{force}\ V \\ & \mathrm{if}\ V\ M\ M' \\ & \mathrm{ret}\ V \\ & \mathrm{let}\ x \leftarrow M; M' \\ & \lambda x.M \mid M\ V \\ & \mathrm{print} s; M \\ & \mathrm{read} x.M \end{array}$$

A computation *does*

- An FA "Feturns" A values
- An $A \to B$ pops an A, continues as B

Equations in CBPV

Every type has associated $\beta\eta$ equality rules

$$\mathrm{force}\,\mathrm{thunk}\,M = M \qquad (V : UB) = \mathrm{thunk}\,\mathrm{force}\,V$$

$$(\lambda x.M)V = M[V/x] \qquad (M : A \to B) = \lambda x.Mx$$

$$\mathrm{let}\,x \leftarrow \mathrm{ret}\,V; N = M[V/x] \qquad N[M : FA/z] = \mathrm{let}\,z \leftarrow M; N$$

And linear terms are *homomorphisms* of effect operations:

$$M[\mathrm{print}\,s; N/z] = \mathrm{print}\,s; M[N/z]$$

$$M[\mathrm{read}\,x.N/z] = \mathrm{read}\,x.M[N/z]$$

CBPV Reconstructs CBV and CBN

CBV term $\Gamma \vdash M : A$ becomes

$$\Gamma^{cbv} \vdash M^{cbv} : FA^{cbv}$$

"CBV terms are always returning"

CBPV Reconstructs CBV and CBN

CBV term $\Gamma \vdash M : A$ becomes

$$\Gamma^{cbv} \vdash M^{cbv} : FA^{cbv}$$

"CBV terms are always returning"

$$(\mathrm{Bool})^{cbv} = \mathrm{Bool}$$

$$(A \rightharpoonup A')^{cbv} = U(A^{cbv} \to FA'^{cbv})$$

CBPV Reconstructs CBV and CBN

CBV term $\Gamma \vdash M : A$ becomes

$$\Gamma^{cbv} \vdash M^{cbv} : FA^{cbv}$$

"CBV terms are always returning"

$$(\mathrm{Bool})^{cbv} = \mathrm{Bool}$$

$$(A \rightharpoonup A')^{cbv} = U(A^{cbv} \to FA'^{cbv})$$

CBN terms $x_1 : B_1, \ldots \vdash M : B$ become

$$x_1 : UB_1^{cbn}, \ldots \vdash M^{cbn} : B^{cbn}$$

"CBN variables are always thunks"

CBPV Reconstructs CBV and CBN

CBV term $\Gamma \vdash M : A$ becomes

$$\Gamma^{cbv} \vdash M^{cbv} : FA^{cbv}$$

"CBV terms are always returning"

$$(\mathrm{Bool})^{cbv} = \mathrm{Bool}$$

$$(A \rightharpoonup A')^{cbv} = U(A^{cbv} \to FA'^{cbv})$$

CBN terms $x_1 : B_1, \ldots \vdash M : B$ become

$$x_1 : UB_1^{cbn}, \ldots \vdash M^{cbn} : B^{cbn}$$

"CBN variables are always thunks"

$$(\mathrm{Bool})^{cbn} = F\mathrm{Bool}$$

$$(B \to B')^{cbn} = UB^{cbn} \to B'^{cbn}$$

CBPV subsumes Functional IRs

A-Normal Form, Monadic Normal Form

A-Normal Form:

$$
\begin{aligned}
\text{Values} &::= \quad x \mid \lambda x.M \mid \text{true} \mid \text{false} \\
\text{Operations} O &::= \quad \text{ret}\ V \mid \text{if}\ V\ M\ M' \mid V\ V' \mid \text{print}\ s \mid \text{read} \\
\text{Terms} M &::= \quad O \mid \text{let}\ x \leftarrow O; M'
\end{aligned}
$$

A-Normal Form, Monadic Normal Form

A-Normal Form:

$$\begin{aligned} \text{Values} &::= \quad x \mid \lambda x.M \mid \text{true} \mid \text{false} \\ \text{Operations}\, O &::= \quad \text{ret } V \mid \text{if } V\ M\ M' \mid V\ V' \mid \text{print } s \mid \text{read} \\ \text{Terms}\, M &::= \quad O \mid \text{let } x \leftarrow O; M' \end{aligned}$$

Monadic Normal Form:

$$\begin{aligned} \text{Values} &::= \quad x \mid \lambda x.M \mid \text{true} \mid \text{false} \\ \text{Terms}\, M &::= \quad \text{let } x \leftarrow M; M' \mid \text{ret } V \mid \text{if } V\ M\ M' \mid V\ V' \mid \text{print } s \mid \text{read} \end{aligned}$$

With equational theories as well. Every MNF term is equal in the theory to an ANF term.

A-Normal Form, Monadic Normal Form

A-Normal Form:

$$\begin{aligned} \text{Values} &::= \; x \,|\, \lambda x.M \,|\, \text{true} \,|\, \text{false} \\ \text{Operations} O &::= \; \text{ret } V \,|\, \text{if } V\; M\; M' \,|\, V\; V' \,|\, \text{print } s \,|\, \text{read} \\ \text{Terms} M &::= \; O \,|\, \text{let } x \leftarrow O; M' \end{aligned}$$

Monadic Normal Form:

$$\begin{aligned} \text{Values} &::= \; x \,|\, \lambda x.M \,|\, \text{true} \,|\, \text{false} \\ \text{Terms} M &::= \; \text{let } x \leftarrow M; M' \,|\, \text{ret } V \,|\, \text{if } V\; M\; M' \,|\, V\; V' \,|\, \text{print } s \,|\, \text{read} \end{aligned}$$

With equational theories as well. Every MNF term is equal in the theory to an ANF term.

Observe: this is isomorphic a “full” subset of CBPV where the only computation type is FA and $A \rightharpoonup A'$ is given $\beta\eta$ rules corresponding to $U(A \to FA')$.

“Fine-grained CBV”, see Levy, Power and Thielecke, Information and Computation 2003.

CBPV Subsumes ANF, MNF

$$
\begin{tikzcd}
 & \mathrm{CBV} \arrow[dl, hook'] \arrow[dr, hook] & \\
\mathrm{MNF} \arrow[rr, two heads, bend left] \arrow[dr, hook] & & \mathrm{ANF} \arrow[ll, bend left] \arrow[dl, hook'] \\
 & \mathrm{CBPV} &
\end{tikzcd}
$$

ANF is Equivalent to Continuation Passing Style

A-normal form was introduced in Sabry and Felleisen *Reasoning about Programs in Continuation-Passing Style* Lisp & F.P. 1992.
Conversion to A-normal form is equivalent to CPS conversion followed by "unCPS".

$$\begin{tikzcd} & \mathrm{CBV} \arrow[dl, hook] \arrow[dr, hook] & \\ \mathrm{ANF} \arrow[rr, bend left=10] & & \mathrm{CPS} \arrow[ll, bend left=10] \end{tikzcd}$$

ANF : CPS as CBPV : ?

$$\begin{tikzcd} \mathrm{ANF} \arrow[r, bend left] \arrow[d, hook] & \mathrm{CPS} \arrow[l, bend left] \\ \mathrm{CBPV} & \end{tikzcd}$$

ANF : CPS as CBPV : ?

$$\begin{tikzcd}
\mathrm{ANF} \arrow[r, bend left] \arrow[d, hook] & \mathrm{CPS} \arrow[l, bend left] \arrow[d, hook] \\
\mathrm{CBPV} \arrow[r, bend left] & ? \arrow[l, bend left]
\end{tikzcd}$$

Stack-Passing Style: The Opposite of CBPV

Two *kinds* of types:

1. Value types: similar to CBPV
2. *Stack* types: the type of the stack a computation runs against

Stack-Passing Style: The Opposite of CBPV

Two *kinds* of types:

1. Value types: similar to CBPV
2. *Stack* types: the type of the stack a computation runs against

Three notions of term

1. Values $\Gamma \vdash V : A$
2. Stacks, i.e., linear values $\Gamma \mid z : B \vdash S : B'$
3. Computations, $\Gamma \mid z : B \vdash M$

With "obvious" substitution principles.

Stack-Passing Style: The Opposite of CBPV

Value Types, Values

$$\begin{array}{rl} \text{A,A'} ::= & \overset{p}{\neg} B \mid \mathrm{Bool} \\ \text{V, V'} ::= & x \mid \lambda z.M \mid \mathrm{true} \mid \mathrm{false} \end{array}$$

A value *is*

- A $\overset{p}{\neg} B$ is a first class procedure that requires a B stack to run.
- A Bool is true or false.

Stack Types, Stacks

$$\begin{array}{rl} \text{B,B'} ::= & \overset{k}{\neg} A \mid A \oslash B \\ \text{S, S'} ::= & z \mid \lambda x.S \mid (V, S) \end{array}$$

A stack *is, linearly,*

- A $\overset{k}{\neg} A$ is a linear *kontinuation* for A values
- An $A \oslash B$ is an A pushed onto a B stack.

Stack-Passing Style: The Opposite of CBPV

Value Types, Values

$$\begin{array}{rl} \text{A,A' ::=} & \overset{p}{\neg} B \mid \mathrm{Bool} \\ \text{V, V' ::=} & x \mid \lambda z.M \mid \mathrm{true} \mid \mathrm{false} \end{array}$$

A value *is*

- A $\overset{p}{\neg} B$ is a first class procedure that requires a B stack to run.
- A Bool is true or false.

Stack Types, Stacks

$$\begin{array}{rl} \text{B,B' ::=} & \overset{k}{\neg} A \mid A \oslash B \\ \text{S, S' ::=} & z \mid \lambda x.S \mid (V, S) \end{array}$$

A stack *is, linearly,*

- A $\overset{k}{\neg} A$ is a linear *kontinuation* for A values
- An $A \oslash B$ is an A pushed onto a B stack.

Computations

$$\begin{array}{rl} M, M' ::= & V(S) \mid S(V) \mid \mathrm{if}\ V\ M\ M' \\ & \mathrm{let}(x, z) = S\ \mathrm{in}\ M \\ & \mathrm{print} s; M \mid \mathrm{read} x.M \end{array}$$

A computation *isn't* (no output)

CBPV to SPS and Back

$$
\begin{aligned}
\mathrm{Bool}^{sps} &= \mathrm{Bool} \\
(UB)^{sps} &= \overset{p}{\neg} B^{sps} \\
(A \to B)^{sps} &= A^{sps} \oslash B^{sps} \\
(FA)^{sps} &= \overset{k}{\neg} A^{sps}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Bool}^{cbpv} &= \mathrm{Bool} \\
(\overset{p}{\neg} B)^{cbpv} &= UB^{cbpv} \\
(A \oslash B)^{cbpv} &= A^{cbpv} \to B^{cbpv} \\
(\overset{k}{\neg} A)^{cbpv} &= FA^{cbpv}
\end{aligned}
$$

CBPV to SPS and Back

$$
\begin{aligned}
\mathrm{Bool}^{sps} &= \mathrm{Bool} \\
(UB)^{sps} &= \overset{p}{\neg} B^{sps} \\
(A \to B)^{sps} &= A^{sps} \oslash B^{sps} \\
(FA)^{sps} &= \overset{k}{\neg} A^{sps}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Bool}^{cbpv} &= \mathrm{Bool} \\
(\overset{p}{\neg} B)^{cbpv} &= UB^{cbpv} \\
(A \oslash B)^{cbpv} &= A^{cbpv} \to B^{cbpv} \\
(\overset{k}{\neg} A)^{cbpv} &= FA^{cbpv}
\end{aligned}
$$

Linear duality!

CBPV and SPS as Flavors of Linear Logic

Different "flavors" of linear logic based on the allowed sequents

$$\Gamma \mid \Delta \vdash M : \Delta'$$

CBPV and SPS as Flavors of Linear Logic

Different "flavors" of linear logic based on the allowed sequents

$$\Gamma \mid \Delta \vdash M : \Delta'$$

Calculus	Allowed $\lvert\Delta\rvert$	Allowed $\lvert\Delta'\rvert$
Enriched-Effect Calculus	$= 1$	$= 1$
Call-by-push-value	≤ 1	$= 1$
Stack-passing Style	$= 1$	≤ 1
Intuitionistic	$< \omega$	$= 1$
Co-Intuitionistic	$= 1$	$< \omega$
Classical	$< \omega$	$< \omega$

ANF-CPS Correspondence as Linear Duality

$$
\begin{tikzcd}
\mathrm{ANF} \arrow[rr, bend left] \arrow[d, hook] & & \mathrm{CPS} \arrow[ll, bend left] \arrow[d, hook] \\
\mathrm{CBPV} \arrow[rr, bend left] \arrow[dr, hook] & & \mathrm{SPS} \arrow[ll, bend left] \arrow[dl, hook'] \\
& \mathrm{EEC} \arrow[loop right] \arrow[d] & \\
& \mathrm{CLL} \arrow[loop right] &
\end{tikzcd}
$$

Equality-Preserving Compiler Passes in CBPV/SPS

Two “Polymorphic” Compiler Passes

- *Typed Closure conversion*, Minamide, Morrisett and Harper, POPL '96

$$(A \rightharpoonup A')^{cc} = \exists X.X \times (X, A \rightharpoonup_{code} A')$$

- *Polymorphic Continuation Passing style*

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

From control effects to typed continuation passing, Thielecke, POPL '03

Both passes are type preserving, equivalence preserving*.

Polymorphic Closure Conversion

Target architectures don't have built in support for closures, need to implement them as a pair of an environment and a *code pointer*.

$$(A \rightharpoonup A')^{cc} = \exists X.X \times (X, A \rightharpoonup_{code} A')$$

(For equality preservation: need quotient/parametricity)

Polymorphic Closure Conversion

Target architectures don't have built in support for closures, need to implement them as a pair of an environment and a *code pointer*.

$$(A \rightharpoonup A')^{cc} = \exists X.X \times (X, A \rightharpoonup_{code} A')$$

(For equality preservation: need quotient/parametricity)
In CBPV the closures are the *thunks*:

$$(UB)^{cc} = \exists X : \mathrm{VTy}.X \times \mathrm{CODE}(X \to B^{cc})$$

Polymorphic Closure Conversion

Target architectures don't have built in support for closures, need to implement them as a pair of an environment and a *code pointer*.

$$(A \rightharpoonup A')^{cc} = \exists X.X \times (X, A \rightharpoonup_{code} A')$$

(For equality preservation: need quotient/parametricity)
In CBPV the closures are the *thunks*:

$$(UB)^{cc} = \exists X : \mathrm{VTy}.X \times \mathrm{CODE}(X \to B^{cc})$$

In SPS, the closures are the *procedures*:

$$(\overset{p}{\neg} B)^{cc} = \exists X : \mathrm{ValTy}.X \times \overset{\texttt{code}}{\neg}(X \oslash B^{cc})$$

Polymorphic CPS Conversion

Target architectures only support jumps, not calls with return, need to pass continuations as arguments.

To support arbitrary calls, functions must pass return continuations as arguments.

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

(To preserve equality: require naturality/parametricity)

Polymorphic CPS Conversion

Target architectures only support jumps, not calls with return, need to pass continuations as arguments.
To support arbitrary calls, functions must pass return continuations as arguments.

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

(To preserve equality: require naturality/parametricity)
In CBPV only FA computations return:

$$(FA)^{cps} = \forall R : \mathrm{CompTy}.U(A^{cps} \to R) \to R$$

Polymorphic CPS Conversion

Target architectures only support jumps, not calls with return, need to pass continuations as arguments.
To support arbitrary calls, functions must pass return continuations as arguments.

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

(To preserve equality: require naturality/parametricity)
In CBPV only FA computations return:

$$(FA)^{cps} = \forall R : \mathrm{CompTy}.U(A^{cps} \to R) \to R$$

Isn't SPS already in CPS form?

Polymorphic CPS Conversion

Target architectures only support jumps, not calls with return, need to pass continuations as arguments.
To support arbitrary calls, functions must pass return continuations as arguments.

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

(To preserve equality: require naturality/parametricity)
In CBPV only FA computations return:

$$(FA)^{cps} = \forall R : \mathrm{CompTy}.U(A^{cps} \to R) \to R$$

Isn't SPS already in CPS form? But we dualize:

Polymorphic CPS Conversion

Target architectures only support jumps, not calls with return, need to pass continuations as arguments.
To support arbitrary calls, functions must pass return continuations as arguments.

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

(To preserve equality: require naturality/parametricity)
In CBPV only FA computations return:

$$(FA)^{cps} = \forall R : \mathrm{CompTy}.U(A^{cps} \to R) \to R$$

Isn't SPS already in CPS form? But we dualize:
In SPS, FA becomes $\overset{k}{\neg}A$ the *linear continuations*:

$$(\overset{k}{\neg}A)^{cps} = \exists S : \mathrm{StkTy}.\overset{p}{\neg}(A^{cps} \oslash S) \oslash S$$

Polymorphic CPS Conversion

Target architectures only support jumps, not calls with return, need to pass continuations as arguments.
To support arbitrary calls, functions must pass return continuations as arguments.

$$(A \rightharpoonup A')^{cps} = \forall X.A, (A' \to X) \to X$$

(To preserve equality: require naturality/parametricity)
In CBPV only FA computations return:

$$(FA)^{cps} = \forall R : \mathrm{CompTy}.U(A^{cps} \to R) \to R$$

Isn't SPS already in CPS form? But we dualize:
In SPS, FA becomes $\overset{k}{\neg}A$ the *linear continuations*:

$$(\overset{k}{\neg}A)^{cps} = \exists S : \mathrm{StkTy}.\overset{p}{\neg}(A^{cps} \oslash S) \oslash S$$

In the dual "polymorphic CPS" is "polymorphic closure conversion" of kontinuations!

Does Polymorphic CPS Conversion Preserve Equivalence?

Ahmed and Blume, ICFP '11: polymorphic CPS does not preserve equivalence in CBV evaluation order:

$$\Lambda X.\lambda x : 1, k : (\mathrm{Bool} \to X).y \leftarrow k(\mathrm{true}); k(\mathrm{false})$$

Polymorphic but still "abuses" the kontinuation.

Does Polymorphic CPS Conversion Preserve Equivalence?

Ahmed and Blume, ICFP '11: polymorphic CPS does not preserve equivalence in CBV evaluation order:

$$\Lambda X.\lambda x : 1, k : (\mathrm{Bool} \to X).y \leftarrow k(\mathrm{true}); k(\mathrm{false})$$

Polymorphic but still "abuses" the kontinuation.
But in CBPV parametricity is enough to rule out this behavior. Why?

Does Polymorphic CPS Conversion Preserve Equivalence?

Ahmed and Blume, ICFP '11: polymorphic CPS does not preserve equivalence in CBV evaluation order:

$$\Lambda X.\lambda x : 1, k : (\mathrm{Bool} \to X).y \leftarrow k(\mathrm{true}); k(\mathrm{false})$$

Polymorphic but still "abuses" the kontinuation.
But in CBPV parametricity is enough to rule out this behavior. Why?

$$\begin{aligned}((A \rightharpoonup A')^{cps})^{cbv} &= (\forall X.A^{cps}, (A'^{cps} \to X) \to X)^{cbv} \\ &= \forall X : \mathrm{ValTy}.A^{cps,cbv} \to U(A'^{cps,cbpv} \to FX) \to FX \\ &\ncong \forall R : \mathrm{CompTy}.A^{cps,cbv} \to U(A'^{cps,cbpv} \to R) \to R\end{aligned}$$

Computation/Stack Types in Compilation

(Stack-based) Calling Conventions as Computation Types

$$A_1, \ldots, A_n \rightharpoonup A'$$

(Stack-based) Calling Conventions as Computation Types

1. Left-to-right $$A_1, \dots, A_n \rightharpoonup A'$$

$$A_0 \to A_1 \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

(Stack-based) Calling Conventions as Computation Types

1. Left-to-right

$$A_1, \ldots, A_n \rightharpoonup A'$$

$$A_0 \to A_1 \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

2. Right-to-left

$$A_n \to A_{n-1} \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

(Stack-based) Calling Conventions as Computation Types

$$A_1, \ldots, A_n \rightharpoonup A'$$

1. Left-to-right

$$A_0 \to A_1 \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

2. Right-to-left

$$A_n \to A_{n-1} \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

3. return address before arguments

$$\forall R.\mathrm{CODE}(A' \to R) \to A_0 \to A_1 \to \cdots \to R$$

(Stack-based) Calling Conventions as Computation Types

1. Left-to-right

$$A_1, \ldots, A_n \rightharpoonup A'$$

$$A_0 \to A_1 \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

2. Right-to-left

$$A_n \to A_{n-1} \to \cdots \to \forall R.\mathrm{CODE}(A' \to R) \to R$$

3. return address before arguments

$$\forall R.\mathrm{CODE}(A' \to R) \to A_0 \to A_1 \to \cdots \to R$$

4. Caller-cleanup (cdecl)

$$\forall R.\mathrm{CODE}(A' \to A_0 \to A_1 \to \cdots R) \to A_0 \to A_1 \to \cdots \to R$$

(Stack-based) Calling Conventions as Stack Types

Can dualize the same translations to SPS:

$$A_1, \ldots, A_n \rightharpoonup A'$$

e.g.,

$$A_0 \oslash A_1 \oslash \cdots \exists S. \overset{\texttt{code}}{\neg}(A' \oslash S) \oslash S$$

Compare: Stack-based calling conventions in *Stack-Based Typed Assembly Language* Morrissett, Krary, Glew and Walker JFP 2002

Monads

A monad T in λ calculus is an operation on types T with

$$\eta : B \to TB' \qquad\qquad -^* : (B \to TB') \to (TB \to TB')$$

satisfying 3 equations.

Monads

A monad T in λ calculus is an operation on types T with

$$\eta : B \to TB' \qquad\qquad -^* : (B \to TB') \to (TB \to TB')$$

satisfying 3 equations.
Example: "error monad"

$$TA = E + A$$

Monads

A monad T in λ calculus is an operation on types T with

$$\eta : B \to TB' \qquad\qquad -^* : (B \to TB') \to (TB \to TB')$$

satisfying 3 equations.
Example: "error monad"

$$TA = E + A$$

Good for equational reasoning, but not a good model of how exceptions are *implemented*.

Monads

A monad T in λ calculus is an operation on types T with

$$\eta : B \to TB' \qquad\qquad -^* : (B \to TB') \to (TB \to TB')$$

satisfying 3 equations.
Example: "error monad"

$$TA = E + A$$

Good for equational reasoning, but not a good model of how exceptions are *implemented*. Monads for effects fundamentally *conflate* two aspects: *TA* is a *first class value* representing a *computation that can run*.

Relative Monads

A *relative* monad[1][2] in CBPV consists of a type constructor

$$\mathrm{Eff} : \mathrm{ValTy} \to \mathrm{CompTy}$$

with operations

$$\eta : A \to \mathrm{Eff}\, A \qquad \frac{x : A \vdash N : \mathrm{Eff}\, A'}{z : \mathrm{Eff}\, A \vdash x \leftarrow^{\mathit{Eff}} z; N : \mathrm{Eff}\, A'}$$

satisfying 3 equations.

[1] Altenkirch, Chapman and Uustalu, LMCS 2015

[2] Relative to F, or to the profunctor of computations

Relative Exception Monads

Naïve implementation:

$$F(A + E)$$

Double barreled continuations:

$$\forall R. U(A \to R) \to U(E \to R) \to R$$

Double barreled code pointers:

$$\forall R. \mathrm{CODE}(A \to R) \to \mathrm{CODE}(E \to R) \to R$$

Relative Exception Monads

Stack-walking exception[3]:

[1] Caveat: Need to restrict to well-behaved elements to get a monad

[2] Caveat: need to restrict to a well-behaved subset to get a monad

Relative Exception Monads

Stack-walking exception[3]:

$$\begin{aligned}\mathrm{Exn}\,E\,A \cong F(A+E)\\ &\&(\forall X : \mathrm{ValTy}.U(A \to \mathrm{Exn}\,E\,X) \to \mathrm{Exn}\,E\,X)\\ &\&\forall X : \mathrm{ValTy}.U(E \to \mathrm{Exn}\,X\,A) \to \mathrm{Exn}\,X\,A\end{aligned}$$

[1]Caveat: Need to restrict to well-behaved elements to get a monad

[2]Caveat: need to restrict to a well-behaved subset to get a monad

Relative Exception Monads

Stack-walking exception[3]:

$$\begin{aligned}\mathrm{Exn}\,E\,A \cong\ & F(A+E)\\ &\&(\forall X : \mathrm{ValTy}.U(A \to \mathrm{Exn}\,E\,X) \to \mathrm{Exn}\,E\,X)\\ &\&\forall X : \mathrm{ValTy}.U(E \to \mathrm{Exn}\,X\,A) \to \mathrm{Exn}\,X\,A\end{aligned}$$

Easier to see as the dual in SPS:

$$\begin{aligned}\mathrm{Exn}\,E\,A \cong\ & \overset{k}{\neg}(A+E)\\ &\oplus (\exists X : \mathrm{ValTy}.U(A \oslash \mathrm{Exn}\,E\,X) \oslash \mathrm{Exn}\,E\,X)\\ &\oplus (\exists X : \mathrm{ValTy}.U(E \oslash \mathrm{Exn}\,X\,A) \oslash \mathrm{Exn}\,X\,A\end{aligned}$$

(Caveat: need to quotient to get a monad)

---

[1] Caveat: Need to restrict to well-behaved elements to get a monad

[2] Caveat: need to restrict to a well-behaved subset to get a monad

Future Work

Future: Beyond The Stack, Beyond Sequentiality

1. Only have *stack*-based calling conventions in CBPV proper. Can registers be incorporated in a similarly well-behaved type theory?

Future: Beyond The Stack, Beyond Sequentiality

1. Only have *stack*-based calling conventions in CBPV proper. Can registers be incorporated in a similarly well-behaved type theory?
2. CBPV gives a foundation for sequential composition, can we combine CBPV with Intuitionistic/Classical LL to similarly analyze IRs for concurrent/parallel code?

WIP: Implementation

1. Zydeco, a CBPV Surface Language + Polymorphism
   https://github.com/zydeco-lang/zydeco
2. Surface language where we can experiment with writing code using new abstractions like relative monads.
3. Ongoing work on a backend using a CBPV IR
4. Extend to Dependent CBPV, compile Dependent CBPV...

CBPV as an IR

- CBPV structure arises naturally in compilation
- Foundation for verified equality preserving compilation
- Computation/Stack types useful for typing low-level programming idioms
- An implementation called Zydeco in progress:
  https://github.com/zydeco-lang/zydeco

BONUS: Relative Monads in SPS

A *relative* monad in SPS consists of a type constructor

$$\mathrm{Not} : \mathrm{ValTy} \to \mathrm{StkTy}$$

with operations

$$x : A \,|\, z : \mathrm{Not}\, A \vdash \mathrm{call}(z, x) \qquad \frac{x : A \,|\, z : \mathrm{Not}\, A' \vdash M}{z : \mathrm{Not}\, A' \vdash \lambda^{\mathrm{Not}} x.M : \mathrm{Not}\, A}$$

satisfying 3 equations.