Deriving Practical Implementations of First-Class Functions

Zachary J. Sullivan

February 19, 2021, University of Oregon

Gap of theory and practice

Gap of theory and practice

The λ-calculus is the foundation of functional programming languages like Ocaml and Haskell.

Gap of theory and practice

The λ-calculus is the foundation of functional programming languages like Ocaml and Haskell.

λ-calculus

$$(\lambda f.\, f\ (f\ x))\ (\lambda x.\, x)$$

Gap of theory and practice

The λ-calculus is the foundation of functional programming languages like Ocaml and Haskell.

λ-calculus

$$(\lambda f.\, f\ (f\ x))\ (\lambda x.\, x)$$

Vax Machine code

```
AP_Unwind :
    movl    Head(r0), r0
    movl    r0, -(%EP)
    movl    (r0), r1
    jmp     *0_Unwind(r1)
```

Gap of theory and practice

The λ-calculus is the foundation of functional programming languages like Ocaml and Haskell.

λ-calculus

$$(\lambda f.\, f\ (f\ x))\ (\lambda x.\, x)$$

Vax Machine code

```
AP_Unwind :
  movl    Head(r0), r0
  movl    r0, -(%EP)
  movl    (r0), r1
  jmp     *0_Unwind(r1)
```

What are the intermediate theories between these?

Intermediate theories

Reduction Theory

Intermediate theories

Reduction Theory

Operational Semantics

Intermediate theories

Reduction Theory

Operational Semantics

Combinators and their Machines

Intermediate theories

Reduction Theory

Operational Semantics

Combinators and their Machines

Abstract Machines

Intermediate theories

Reduction Theory

Operational Semantics

Combinators and their Machines

Abstract Machines

Compilation through Intermediate Languages

Schism of Evaluation Strategy

Schism of Evaluation Strategy

Implementations can be mostly divided based on *evaluation strategy*:

Schism of Evaluation Strategy

Implementations can be mostly divided based on *evaluation strategy*:

Call-by-value

- Scheme
- SML
- Ocaml

Schism of Evaluation Strategy

Implementations can be mostly divided based on *evaluation strategy*:

Call-by-value

- Scheme
- SML
- Ocaml

Call-by-need

- LazyML
- Miranda
- Haskell

Schism of Evaluation Strategy

Implementations can be mostly divided based on *evaluation strategy*:

Call-by-value

- Scheme
- SML
- Ocaml

Call-by-need

- LazyML
- Miranda
- Haskell

Have we been duplicating work?

Goals

Goals

What are the intermediate theories between λ-calculus and implementations on modern machines?

Goals

What are the intermediate theories between λ-calculus and implementations on modern machines?

What differences in techniques truly depend on evaluation strategy?

Goals

What are the intermediate theories between λ-calculus and implementations on modern machines?

What differences in techniques truly depend on evaluation strategy?

What do we know about our implementations? Are they correct?

Reduction Theory

What is a reduction theory?

What is a reduction theory?

It is composed of some expression syntax S and rewriting rules $S \to S$.

What is a reduction theory?

It is composed of some expression syntax S and rewriting rules $S \to S$.

$$E \in \mathit{Arith\ Expr} ::= n \mid E + E$$

What is a reduction theory?

It is composed of some expression syntax S and rewriting rules $S \to S$.

$$E \in \mathit{Arith\ Expr} ::= n \mid E + E$$

$$n + m \longrightarrow_{\oplus} c \quad \text{where } n \oplus m = c$$

What is a reduction theory?

It is composed of some expression syntax S and rewriting rules $S \to S$.

$$E \in Arith\ Expr ::= n \mid E + E$$

$$n + m \longrightarrow_{\oplus} c \quad \text{where } n \oplus m = c$$

e.g.

$$(0 + 1) + (2 + 3) \longrightarrow_{\oplus}$$

What is a reduction theory?

It is composed of some expression syntax S and rewriting rules $S \to S$.

$$E \in \mathit{Arith\ Expr} ::= n \mid E + E$$

$$n + m \longrightarrow_{\oplus} c \quad \text{where } n \oplus m = c$$

e.g.

$$(0+1)+(2+3) \longrightarrow_{\oplus} (0+1)+5 \longrightarrow_{\oplus} 1+5 \longrightarrow_{\oplus} 6$$

What is a reduction theory?

It is composed of some expression syntax S and rewriting rules $S \to S$.

$$E \in \mathit{Arith\ Expr} ::= n \mid E + E$$

$$n + m \longrightarrow_{\oplus} c \quad \text{where } n \oplus m = c$$

e.g.

$$(0+1) + (2+3) \longrightarrow_{\oplus} (0+1) + 5 \longrightarrow_{\oplus} 1 + 5 \longrightarrow_{\oplus} 6$$

Reductions are often **compatible**, thereby rewrites can be applied in any order.

The λ-calculus

The λ-calculus

The terms of λ-calculus the following grammar:

$$L, M, N \in \mathit{Expression} ::= x \mid \lambda x.\, M \mid M\ N$$

The λ-calculus

The terms of λ-calculus the following grammar:

$$L, M, N \in \mathit{Expression} ::= x \mid \lambda x.\, M \mid M\ N$$

Evaluation strategies are defined by different sets of reduction rules:

The λ-calculus

The terms of λ-calculus the following grammar:

$$L, M, N \in \mathit{Expression} ::= x \mid \lambda x.\, M \mid M\ N$$

Evaluation strategies are defined by different sets of reduction rules:

- Call-by-name (Church)

The λ-calculus

The terms of λ-calculus the following grammar:

$$L, M, N \in \mathit{Expression} ::= x \mid \lambda x.\, M \mid M\ N$$

Evaluation strategies are defined by different sets of reduction rules:

- Call-by-name (Church)
- Call-by-value (Plotkin)

The λ-calculus

The terms of λ-calculus the following grammar:

$$L, M, N \in \mathit{Expression} ::= x \mid \lambda x.\, M \mid M\ N$$

Evaluation strategies are defined by different sets of reduction rules:

- Call-by-name (Church)
- Call-by-value (Plotkin)
- Call-by-need (Ariola *et al.*)

The λ-calculus

The terms of λ-calculus the following grammar:

$$L, M, N \in \mathit{Expression} ::= x \mid \lambda x.\, M \mid M\ N$$

Evaluation strategies are defined by different sets of reduction rules:

- Call-by-name (Church)
- Call-by-value (Plotkin)
- Call-by-need (Ariola *et al.*)

Variables and Substitution

These calculi depend on two meta language operations:

Variables and Substitution

These calculi depend on two meta language operations:

A variable is **free** if there is no binder for it.

$$\begin{array}{rcl}
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(\lambda x.\, M) & = & \mathrm{FV}(M) - \{x\} \\
\mathrm{FV}(M\ N) & = & \mathrm{FV}(M) \cup \mathrm{FV}(N)
\end{array}$$

Variables and Substitution

These calculi depend on two meta language operations:

A variable is **free** if there is no binder for it.

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(\lambda x.\, M) & = & \mathrm{FV}(M) - \{x\} \\ \mathrm{FV}(M\ N) & = & \mathrm{FV}(M) \cup \mathrm{FV}(N) \end{array}$$

Substitution, $M[N/x]$, replaces N for a variable x in M.

Call-by-name

Church's calculus has two equalities and a single reduction:

Call-by-name

Church's calculus has two equalities and a single reduction:

$$\lambda x.\, M \quad =_\alpha \quad \lambda y.\, M[y/x] \quad \text{where } y \notin \mathrm{FV}(M)$$

Call-by-name

Church's calculus has two equalities and a single reduction:

$$\begin{array}{rcll} \lambda x.\, M & =_\alpha & \lambda y.\, M[y/x] & \text{where } y \notin \mathrm{FV}(M) \\ \lambda x.\, M\ x & =_\eta & M & \text{where } x \notin \mathrm{FV}(M) \end{array}$$

Call-by-name

Church's calculus has two equalities and a single reduction:

$$\begin{array}{rcll} \lambda x.\, M & =_\alpha & \lambda y.\, M[y/x] & \text{where } y \notin \mathrm{FV}(M) \\ \lambda x.\, M\ x & =_\eta & M & \text{where } x \notin \mathrm{FV}(M) \\[1ex] (\lambda x.\, M)\ N & \longrightarrow_\beta & M[N/x] & \end{array}$$

Call-by-value

Call-by-value

Was defined after the SECD machine.

Call-by-value

Was defined after the SECD machine.

Call-by-value restricts β-reduction to **values**:

Call-by-value

Was defined after the SECD machine.

Call-by-value restricts β-reduction to **values**:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

Call-by-value

Was defined after the SECD machine.

Call-by-value restricts β-reduction to **values**:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

$$V, W \in \mathit{Value} ::= x \mid \lambda x.\, M$$

Call-by-value

Was defined after the SECD machine.

Call-by-value restricts β-reduction to **values**:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

$$V, W \in \mathit{Value} ::= x \mid \lambda x.\, M$$

Can reduce fewer terms than call-by-name, *e.g.*

$$(\lambda x.\, 42)\ \Omega$$

Call-by-need

Call-by-name reduction can duplicate work

Call-by-need

Call-by-name reduction can duplicate work *e.g.*

$$(\lambda x.\, x + x)\ (1 + 3) \quad \longrightarrow_{\beta}$$

Call-by-need

Call-by-name reduction can duplicate work *e.g.*

$$\begin{array}{lll} (\lambda x.\, x + x)\ (1+3) & \longrightarrow_\beta & (1+3) + (1+3) \\ & \longrightarrow_+ & 4 + (3+1) \\ & \longrightarrow_+ & 4 + 4 \\ & \longrightarrow_+ & 8 \end{array}$$

Call-by-need

Call-by-name reduction can duplicate work *e.g.*

$$
\begin{aligned}
(\lambda x.\, x + x)\ (1 + 3) \quad & \longrightarrow_\beta \quad (1 + 3) + (1 + 3) \\
& \longrightarrow_+ \quad 4 + (3 + 1) \\
& \longrightarrow_+ \quad 4 + 4 \\
& \longrightarrow_+ \quad 8
\end{aligned}
$$

Call-by-need will share the computation of $(1 + 3)$.

Call-by-need λ-calculus

Applied λ-expressions are like let-expressions:

$$(\lambda x.\, x + x)\ (1 + 3) = \texttt{let}\ x := 1 + 3\ \texttt{in}\ x + x$$

Call-by-need λ-calculus

Applied λ-expressions are like let-expressions:

$$(\lambda x.\, x + x)\ (1 + 3) = \texttt{let}\ x := 1 + 3\ \texttt{in}\ x + x$$

$$\texttt{let}\ x := V\ \texttt{in}\ E[x] \quad \longrightarrow_V \quad \texttt{let}\ x := V\ \texttt{in}\ E[V]$$

Call-by-need λ-calculus

Applied λ-expressions are like let-expressions:

$$(\lambda x.\, x + x)\ (1 + 3) = \texttt{let}\ x := 1 + 3\ \texttt{in}\ x + x$$

$$\begin{array}{rcl} \texttt{let}\ x := V\ \texttt{in}\ E[x] & \longrightarrow_V & \texttt{let}\ x := V\ \texttt{in}\ E[V] \\ (\texttt{let}\ x := M\ \texttt{in}\ L)\ N & \longrightarrow_C & \texttt{let}\ x := M\ \texttt{in}\ L\ N \end{array}$$

Call-by-need λ-calculus

Applied λ-expressions are like let-expressions:

$$(\lambda x.\, x + x)\ (1 + 3) = \texttt{let}\ x := 1 + 3\ \texttt{in}\ x + x$$

$$\begin{array}{rcl}
\texttt{let}\ x := V\ \texttt{in}\ E[x] & \longrightarrow_V & \texttt{let}\ x := V\ \texttt{in}\ E[V] \\
(\texttt{let}\ x := M\ \texttt{in}\ L)\ N & \longrightarrow_C & \texttt{let}\ x := M\ \texttt{in}\ L\ N \\
\texttt{let}\ x := (\texttt{let}\ y := N\ \texttt{in}\ M)\ \texttt{in}\ L & \longrightarrow_A & \texttt{let}\ y := N\ \texttt{in} \\
 & & \quad \texttt{let}\ x := M\ \texttt{in}\ L
\end{array}$$

Call-by-need λ-calculus

Applied λ-expressions are like let-expressions:

$$(\lambda x.\, x + x)\ (1 + 3) = \texttt{let}\ x := 1 + 3\ \texttt{in}\ x + x$$

$$\begin{array}{rcl}
\texttt{let}\ x := V\ \texttt{in}\ E[x] & \longrightarrow_V & \texttt{let}\ x := V\ \texttt{in}\ E[V] \\
(\texttt{let}\ x := M\ \texttt{in}\ L)\ N & \longrightarrow_C & \texttt{let}\ x := M\ \texttt{in}\ L\ N \\
\texttt{let}\ x := (\texttt{let}\ y := N\ \texttt{in}\ M)\ \texttt{in}\ L & \longrightarrow_A & \texttt{let}\ y := N\ \texttt{in} \\
 & & \quad \texttt{let}\ x := M\ \texttt{in}\ L
\end{array}$$

$$V, W \in \quad \mathit{Value} \quad ::= \lambda x.\, M$$

Call-by-need λ-calculus

Applied λ-expressions are like let-expressions:

$$(\lambda x.\, x + x)\ (1 + 3) = \texttt{let}\ x := 1 + 3\ \texttt{in}\ x + x$$

$$\begin{array}{rcl}
\texttt{let}\ x := V\ \texttt{in}\ E[x] & \longrightarrow_V & \texttt{let}\ x := V\ \texttt{in}\ E[V] \\
(\texttt{let}\ x := M\ \texttt{in}\ L)\ N & \longrightarrow_C & \texttt{let}\ x := M\ \texttt{in}\ L\ N \\
\texttt{let}\ x := (\texttt{let}\ y := N\ \texttt{in}\ M)\ \texttt{in}\ L & \longrightarrow_A & \texttt{let}\ y := N\ \texttt{in} \\
 & & \quad \texttt{let}\ x := M\ \texttt{in}\ L
\end{array}$$

$$V, W \in \quad \mathit{Value} \quad ::= \lambda x.\, M$$

Instead of β, the call-by-need calculus operates on graphs represented by let-expressions.

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

These can be slow in practice.

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

These can be slow in practice.

As a remedy:

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

These can be slow in practice.

As a remedy:

- De Bruijn notation allows us to reason about programs free of variables

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

These can be slow in practice.

As a remedy:

- De Bruijn notation allows us to reason about programs free of variables *e.g.*

$$(\lambda x.\lambda y.\, x) \Longrightarrow \lambda\lambda\underline{1}$$

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

These can be slow in practice.

As a remedy:

- De Bruijn notation allows us to reason about programs free of variables *e.g.*
$$(\lambda x.\lambda y.\, x) \Longrightarrow \lambda\lambda\underline{1}$$

- Explicit substitutions completely remove substitutions from the meta language

Reducing the Meta Language

So far, I have used $\mathrm{FV}(M)$, $M[N/x]$, and $E[M]$.

These can be slow in practice.

As a remedy:

- De Bruijn notation allows us to reason about programs free of variables *e.g.*

$$(\lambda x.\lambda y.\, x) \Longrightarrow \lambda\lambda\underline{1}$$

- Explicit substitutions completely remove substitutions from the meta language
- Abstract machines avoid the evaluation context's meta operation

Operational Semantics

What is an operational semantics?

Reduction theories simply specify a set of rules to apply without guidance.

What is an operational semantics?

Reduction theories simply specify a set of rules to apply without guidance.

In practice, we often want to know exactly how our program computes a result *e.g.* consider the order of effects.

What is an operational semantics?

Reduction theories simply specify a set of rules to apply without guidance.

In practice, we often want to know exactly how our program computes a result *e.g.* consider the order of effects.

An operational semantics describes a method for computing a result.

e.g. Arithmetic

An operational semantics for numbers will pick an evaluation order:

$$\frac{m \oplus n = c}{m + n \longmapsto c}$$

e.g. Arithmetic

An operational semantics for numbers will pick an evaluation order:

$$\frac{m \oplus n = c}{m + n \longmapsto c} \qquad \frac{M \longmapsto M'}{M + N \longmapsto M' + N}$$

e.g. Arithmetic

An operational semantics for numbers will pick an evaluation order:

$$\frac{m \oplus n = c}{m + n \longmapsto c} \qquad \frac{M \longmapsto M'}{M + N \longmapsto M' + N} \qquad \frac{N \longmapsto N'}{m + N \longmapsto m + N'}$$

e.g. Arithmetic

An operational semantics for numbers will pick an evaluation order:

$$\frac{m \oplus n = c}{m + n \longmapsto c} \qquad \frac{M \longmapsto M'}{M + N \longmapsto M' + N} \qquad \frac{N \longmapsto N'}{m + N \longmapsto m + N'}$$

The last two rules are called **structural rules**.

e.g. Arithmetic

An operational semantics for numbers will pick an evaluation order:

$$\frac{m \oplus n = c}{m + n \longmapsto c} \qquad \frac{M \longmapsto M'}{M + N \longmapsto M' + N} \qquad \frac{N \longmapsto N'}{m + N \longmapsto m + N'}$$

The last two rules are called **structural rules**. They find the next reducible expression.

e.g. Arithmetic

An operational semantics for numbers will pick an evaluation order:

$$\frac{m \oplus n = c}{m + n \longmapsto c} \qquad \frac{M \longmapsto M'}{M + N \longmapsto M' + N} \qquad \frac{N \longmapsto N'}{m + N \longmapsto m + N'}$$

The last two rules are called **structural rules**. They find the next reducible expression.

Evaluation chains multiple steps till a result is found:

$$\mathrm{eval}(M) = M' \quad \text{where } M \longmapsto^* M' \not\longmapsto M''$$

Operational Semantics for Λ-terms

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]}$$

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]} \qquad \frac{M \longmapsto M'}{M\ N \longmapsto M'\ N}$$

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]} \qquad \frac{M \longmapsto M'}{M\ N \longmapsto M'\ N}$$

For call-by-value, we must evaluate the argument to a value:

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]} \qquad \frac{M \longmapsto M'}{M\ N \longmapsto M'\ N}$$

For call-by-value, we must evaluate the argument to a value:

$$\frac{}{(\lambda x.\, M)\ V \longmapsto M[V/x]}$$

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]} \qquad \frac{M \longmapsto M'}{M\ N \longmapsto M'\ N}$$

For call-by-value, we must evaluate the argument to a value:

$$\frac{}{(\lambda x.\, M)\ V \longmapsto M[V/x]}$$

$$\frac{M \longmapsto M'}{M\ N \longmapsto M'\ N}$$

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\; N \longmapsto M[N/x]} \qquad \frac{M \longmapsto M'}{M\; N \longmapsto M'\; N}$$

For call-by-value, we must evaluate the argument to a value:

$$\frac{}{(\lambda x.\, M)\; V \longmapsto M[V/x]}$$

$$\frac{M \longmapsto M'}{M\; N \longmapsto M'\; N} \qquad \frac{N \longmapsto N'}{(\lambda x.\, M)\; N \longmapsto (\lambda x.\, M)\; N'}$$

Operational Semantics for Λ-terms

For call-by-name, arguments are not evaluated:

$$\frac{}{(\lambda x.\, M)\ N \longmapsto M[N/x]} \qquad \frac{M \longmapsto M'}{M\ N \longmapsto M'\ N}$$

For call-by-value, we must evaluate the argument to a value:

$$\frac{}{(\lambda x.\, M)\ V \longmapsto M[V/x]}$$

$$\frac{M \longmapsto M'}{M\ N \longmapsto M'\ N} \qquad \frac{N \longmapsto N'}{(\lambda x.\, M)\ N \longmapsto (\lambda x.\, M)\ N'}$$

The only difference in these two strategies is forcing the argument.

Evaluation Contexts

Evaluation Contexts

Evaluation contexts allow us to specify only a single structural rule:

Evaluation Contexts

Evaluation contexts allow us to specify only a single structural rule:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Evaluation Contexts

Evaluation contexts allow us to specify only a single structural rule:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Given a set of evaluation contexts, we can uniquely decide the next redex:

$$\mathit{Eval\ Context}_{+} \quad ::= \Box \mid E + N \mid n + E$$

Evaluation Contexts

Evaluation contexts allow us to specify only a single structural rule:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Given a set of evaluation contexts, we can uniquely decide the next redex:

$$\begin{array}{ll} \mathit{Eval\ Context}_{+} & ::= \Box \mid E + N \mid n + E \\ \mathit{Eval\ Context}_{\mathcal{N}} & ::= \Box \mid E\ N \end{array}$$

Evaluation Contexts

Evaluation contexts allow us to specify only a single structural rule:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Given a set of evaluation contexts, we can uniquely decide the next redex:

$$\begin{array}{ll}
\mathit{Eval\ Context}_{+} & ::= \Box \mid E + N \mid n + E \\
\mathit{Eval\ Context}_{\mathcal{N}} & ::= \Box \mid E\ N \\
\mathit{Eval\ Context}_{\mathcal{V}} & ::= \Box \mid E\ N \mid (\lambda x.\, M)\ E
\end{array}$$

Evaluation Contexts

Evaluation contexts allow us to specify only a single structural rule:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Given a set of evaluation contexts, we can uniquely decide the next redex:

$$\begin{array}{ll}
\mathit{Eval\ Context}_{+} & ::= \Box \mid E + N \mid n + E \\
\mathit{Eval\ Context}_{\mathcal{N}} & ::= \Box \mid E\ N \\
\mathit{Eval\ Context}_{\mathcal{V}} & ::= \Box \mid E\ N \mid (\lambda x.\, M)\ E \\
\mathit{Eval\ Context}_{\mathcal{L}} & ::= \Box \mid E\ N \mid \texttt{let}\ x := N\ \texttt{in}\ E \mid \texttt{let}\ x := E\ \texttt{in}\ E[x]
\end{array}$$

Big-Step Semantics

Big-Step Semantics

Big-step semantics relate an expression to its *final* output.

Big-Step Semantics

Big-step semantics relate an expression to its *final* output.

e.g.

$$\frac{M \Downarrow m \quad N \Downarrow n}{M + N \Downarrow m \oplus n} \qquad \frac{}{n \Downarrow n}$$

Big-Step Semantics

Big-step semantics relate an expression to its *final* output.

e.g.

$$\frac{M \Downarrow m \quad N \Downarrow n}{M + N \Downarrow m \oplus n} \qquad \frac{}{n \Downarrow n}$$

These are defined inductively over the syntax of an expression.

Big-Step Semantics

Big-step semantics relate an expression to its *final* output.

e.g.

$$\frac{M \Downarrow m \quad N \Downarrow n}{M + N \Downarrow m \oplus n} \qquad \frac{}{n \Downarrow n}$$

These are defined inductively over the syntax of an expression.

This style of semantics can be easily implemented as a recursive traversal of a term.

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x}$$

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \qquad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M}$$

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \qquad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M} \qquad \frac{M \Downarrow_{\mathcal{N}} \lambda x.\, L \quad L[N/x] \Downarrow_{\mathcal{N}} R}{M\ N \Downarrow_{\mathcal{N}} R}$$

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \qquad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M} \qquad \frac{M \Downarrow_{\mathcal{N}} \lambda x.\, L \quad L[N/x] \Downarrow_{\mathcal{N}} R}{M\ N \Downarrow_{\mathcal{N}} R}$$

Call-by-value changes only in the application case:

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \qquad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M} \qquad \frac{M \Downarrow_{\mathcal{N}} \lambda x.\, L \quad L[N/x] \Downarrow_{\mathcal{N}} R}{M\ N \Downarrow_{\mathcal{N}} R}$$

Call-by-value changes only in the application case:

$$\frac{M \Downarrow_{\mathcal{V}} \lambda x.\, L \quad N \Downarrow_{\mathcal{V}} V \quad L[V/x] \Downarrow_{\mathcal{V}} R}{M\ N \Downarrow_{\mathcal{V}} R}$$

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \quad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M} \quad \frac{M \Downarrow_{\mathcal{N}} \lambda x.\, L \quad L[N/x] \Downarrow_{\mathcal{N}} R}{M\ N \Downarrow_{\mathcal{N}} R}$$

Call-by-value changes only in the application case:

$$\frac{M \Downarrow_{\mathcal{V}} \lambda x.\, L \quad N \Downarrow_{\mathcal{V}} V \quad L[V/x] \Downarrow_{\mathcal{V}} R}{M\ N \Downarrow_{\mathcal{V}} R}$$

Call-by-need must thread a heap throughout:

Big-Step Semantics for Λ-terms

Call-by-name rules:

$$\frac{}{x \Downarrow_{\mathcal{N}} x} \quad \frac{}{\lambda x.\, M \Downarrow_{\mathcal{N}} \lambda x.\, M} \quad \frac{M \Downarrow_{\mathcal{N}} \lambda x.\, L \quad L[N/x] \Downarrow_{\mathcal{N}} R}{M\ N \Downarrow_{\mathcal{N}} R}$$

Call-by-value changes only in the application case:

$$\frac{M \Downarrow_{\mathcal{V}} \lambda x.\, L \quad N \Downarrow_{\mathcal{V}} V \quad L[V/x] \Downarrow_{\mathcal{V}} R}{M\ N \Downarrow_{\mathcal{V}} R}$$

Call-by-need must thread a heap throughout:

$$\frac{\langle \Phi \parallel M \rangle \Downarrow_{\mathcal{L}} (\Phi', \lambda x.\, L) \quad \langle \Phi', x' \mapsto N \parallel L[x'/x] \rangle \Downarrow_{\mathcal{L}} R}{\langle \Phi \parallel M\ N \rangle \Downarrow_{\mathcal{L}} R}$$

Big-step Environment Semantics

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Considering call-by-name:

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Considering call-by-name:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}$$

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Considering call-by-name:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}$$

Closures are added as results to capture the values of free variables.

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Considering call-by-name:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}$$

Closures are added as results to capture the values of free variables.

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{N}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma', x \mapsto (\Sigma, N) \parallel L \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{N}} R}$$

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Considering call-by-name:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}$$

Closures are added as results to capture the values of free variables.

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{N}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma', x \mapsto (\Sigma, N) \parallel L \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{N}} R}$$

Thunk closures are added as values since parameters may have free variables.

Big-step Environment Semantics

Instead of substitutions, the semantics are defined with an environment.

Considering call-by-name:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{N}} (\Sigma, \lambda x.\, M)}$$

Closures are added as results to capture the values of free variables.

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{N}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma', x \mapsto (\Sigma, N) \parallel L \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{N}} R}$$

Thunk closures are added as values since parameters may have free variables.

$$\frac{\Sigma(x) = (\Sigma', M) \quad \langle \Sigma' \parallel M \rangle \Downarrow_{\mathcal{N}} R}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{N}} R}$$

Evaluation of variable is delayed until lookup.

Big-step Environment Semantics

Considering call-by-value:

Big-step Environment Semantics

Considering call-by-value:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}} (\Sigma, \lambda x.\, M)}$$

Big-step Environment Semantics

Considering call-by-value:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}} (\Sigma, \lambda x.\, M)}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}} V \quad \langle \Sigma', x \mapsto V \parallel L \rangle \Downarrow_{\mathcal{V}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{V}} R}$$

Big-step Environment Semantics

Considering call-by-value:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}} (\Sigma, \lambda x.\, M)}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}} V \quad \langle \Sigma', x \mapsto V \parallel L \rangle \Downarrow_{\mathcal{V}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{V}} R}$$

$$\frac{\Sigma(x) = R}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{V}} R}$$

Big-step Environment Semantics

Considering call-by-value:

$$\frac{}{\langle \Sigma \parallel \lambda x.\, M \rangle \Downarrow_{\mathcal{V}} (\Sigma, \lambda x.\, M)}$$

$$\frac{\langle \Sigma \parallel M \rangle \Downarrow_{\mathcal{V}} (\Sigma', \lambda x.\, L) \quad \langle \Sigma \parallel N \rangle \Downarrow_{\mathcal{V}} V \quad \langle \Sigma', x \mapsto V \parallel L \rangle \Downarrow_{\mathcal{V}} R}{\langle \Sigma \parallel M\ N \rangle \Downarrow_{\mathcal{V}} R}$$

$$\frac{\Sigma(x) = R}{\langle \Sigma \parallel x \rangle \Downarrow_{\mathcal{V}} R}$$

Call-by-value does **not** require thunk closures.

Operational Semantics Recap

Operational Semantics Recap

Describe how to get from a program to a final result.

Operational Semantics Recap

Describe how to get from a program to a final result.

Small-step semantics adds structural rules to decide which reduction to apply next.

Operational Semantics Recap

Describe how to get from a program to a final result.

Small-step semantics adds structural rules to decide which reduction to apply next.

Big-step semantics describes evaluation inductively over the syntax.

Operational Semantics Recap

Describe how to get from a program to a final result.

Small-step semantics adds structural rules to decide which reduction to apply next.

Big-step semantics describes evaluation inductively over the syntax.

Concerning evaluation strategies:

Operational Semantics Recap

Describe how to get from a program to a final result.

Small-step semantics adds structural rules to decide which reduction to apply next.

Big-step semantics describes evaluation inductively over the syntax.

Concerning evaluation strategies:

- All call-by-value semantics must force arguments.

Operational Semantics Recap

Describe how to get from a program to a final result.

Small-step semantics adds structural rules to decide which reduction to apply next.

Big-step semantics describes evaluation inductively over the syntax.

Concerning evaluation strategies:

- All call-by-value semantics must force arguments.
- Call-by-name environment semantics require more closures.

Operational Semantics Recap

Describe how to get from a program to a final result.

Small-step semantics adds structural rules to decide which reduction to apply next.

Big-step semantics describes evaluation inductively over the syntax.

Concerning evaluation strategies:

- All call-by-value semantics must force arguments.
- Call-by-name environment semantics require more closures.
- Call-by-need big-step semantics requires heaps.

Combinators and their Machines

Combinators

Combinators

Combinators are expressions without free variables:

Combinators

Combinators are expressions without free variables:

$$\mathtt{S}\ f\ g\ x \quad = \quad f\ x\ (g\ x)$$

Combinators

Combinators are expressions without free variables:

$$
\begin{aligned}
\mathtt{S}\ f\ g\ x &= f\ x\ (g\ x) \\
\mathtt{K}\ x\ y &= x
\end{aligned}
$$

Combinators

Combinators are expressions without free variables:

$$\begin{aligned} \mathtt{S}\ f\ g\ x &= f\ x\ (g\ x) \\ \mathtt{K}\ x\ y &= x \\ \mathtt{I}\ x &= x \end{aligned}$$

Combinators

Combinators are expressions without free variables:

$$
\begin{aligned}
\mathtt{S}\ f\ g\ x &= f\ x\ (g\ x) \\
\mathtt{K}\ x\ y &= x \\
\mathtt{I}\ x &= x \\
\mathtt{plus}\ x\ y &= x + y
\end{aligned}
$$

Combinators

Combinators are expressions without free variables:

$$\begin{aligned}
\mathtt{S}\ f\ g\ x &= f\ x\ (g\ x) \\
\mathtt{K}\ x\ y &= x \\
\mathtt{I}\ x &= x \\
\mathtt{plus}\ x\ y &= x + y
\end{aligned}$$

To represent computable functions, we need only S and K.

Combinators

Combinators are expressions without free variables:

$$\begin{aligned}
\mathtt{S}\ f\ g\ x &= f\ x\ (g\ x) \\
\mathtt{K}\ x\ y &= x \\
\mathtt{I}\ x &= x \\
\mathtt{plus}\ x\ y &= x + y
\end{aligned}$$

To represent computable functions, we need only S and K.

Combinator machines (Turner) take combinators to be the only function primitives.

Combinator machines

We must first convert our functions into combinators:

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

We can build a machine that manipulates a term with only combinator application:

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S (S (K plus) I) I}$$

We can build a machine that manipulates a term with only combinator application:

$$\texttt{(S (S (K plus) I) I) 21}$$
$$\longrightarrow_{\texttt{S}}$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

We can build a machine that manipulates a term with only combinator application:

$$\begin{array}{l} (\texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I})\ 21 \\ \longrightarrow_{\texttt{S}} (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ 21\ (\texttt{I}\ 21) \end{array}$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

We can build a machine that manipulates a term with only combinator application:

$$\begin{aligned}
&(\texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I})\ 21 \\
&\longrightarrow_{\texttt{S}} (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ 21\ (\texttt{I}\ 21) \\
&\longrightarrow_{\texttt{S}} (\texttt{K}\ \texttt{plus})\ 21\ (\texttt{I}\ 21)\ (\texttt{I}\ 21)
\end{aligned}$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

We can build a machine that manipulates a term with only combinator application:

$$\begin{array}{l} (\texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I})\ 21 \\ \longrightarrow_{\texttt{S}} (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ 21\ (\texttt{I}\ 21) \\ \longrightarrow_{\texttt{S}} (\texttt{K}\ \texttt{plus})\ 21\ (\texttt{I}\ 21)\ (\texttt{I}\ 21) \\ \longrightarrow_{\texttt{K}} \texttt{plus}\ (\texttt{I}\ 21)\ (\texttt{I}\ 21) \end{array}$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S (S (K plus) I) I}$$

We can build a machine that manipulates a term with only combinator application:

$$\begin{array}{l}
\texttt{(S (S (K plus) I) I) 21} \\
\longrightarrow_{\texttt{S}} \texttt{(S (K plus) I) 21 (I 21)} \\
\longrightarrow_{\texttt{S}} \texttt{(K plus) 21 (I 21) (I 21)} \\
\longrightarrow_{\texttt{K}} \texttt{plus (I 21) (I 21)} \\
\longrightarrow_{\texttt{I}}^{2} \texttt{plus 21 21}
\end{array}$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

We can build a machine that manipulates a term with only combinator application:

$$
\begin{aligned}
&(\texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I})\ 21 \\
&\longrightarrow_{\texttt{S}} (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ 21\ (\texttt{I}\ 21) \\
&\longrightarrow_{\texttt{S}} (\texttt{K}\ \texttt{plus})\ 21\ (\texttt{I}\ 21)\ (\texttt{I}\ 21) \\
&\longrightarrow_{\texttt{K}} \texttt{plus}\ (\texttt{I}\ 21)\ (\texttt{I}\ 21) \\
&\longrightarrow_{\texttt{I}}^{2} \texttt{plus}\ 21\ 21 \\
&\longrightarrow_{+} 42
\end{aligned}
$$

Combinator machines

We must first convert our functions into combinators:

$$\texttt{def}\ \mathit{double}\ x = x + x$$

is converted to:

$$\texttt{def}\ \mathit{double} \quad = \quad \texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I}$$

We can build a machine that manipulates a term with only combinator application:

$$\begin{array}{l}
(\texttt{S}\ (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ \texttt{I})\ 21 \\
\longrightarrow_{\texttt{S}} (\texttt{S}\ (\texttt{K}\ \texttt{plus})\ \texttt{I})\ 21\ (\texttt{I}\ 21) \\
\longrightarrow_{\texttt{S}} (\texttt{K}\ \texttt{plus})\ 21\ (\texttt{I}\ 21)\ (\texttt{I}\ 21) \\
\longrightarrow_{\texttt{K}} \texttt{plus}\ (\texttt{I}\ 21)\ (\texttt{I}\ 21) \\
\longrightarrow_{\texttt{I}}^{2} \texttt{plus}\ 21\ 21 \\
\longrightarrow_{+} 42
\end{array}$$

Notice the lack of β-reduction and variables.

Super-combinators and Lambda-lifting

Super-combinators and Lambda-lifting

Super-combinators can be derived from source code by lambda-lifting.

Super-combinators and Lambda-lifting

Super-combinators can be derived from source code by lambda-lifting.

e.g.

$$\ldots(\lambda x.x + 6 * y)\ldots$$

Super-combinators and Lambda-lifting

Super-combinators can be derived from source code by lambda-lifting.

e.g.

$$\ldots(\lambda x.x + 6 * y)\ldots$$

generates the super-combinator:

$$\mathsf{F}\ y\ x = x + 6 * y$$

Super-combinators and Lambda-lifting

Super-combinators can be derived from source code by lambda-lifting.

e.g.

$$\ldots(\lambda x.x + 6 * y)\ldots$$

generates the super-combinator:

$$\mathsf{F}\ y\ x = x + 6 * y$$

And a partial applications is left in place of the original function:

$$\ldots\mathsf{F}\ y\ldots$$

Combinator Machines

Combinator Machines

Fixed set of combinators:

Combinator Machines

Fixed set of combinators:

- Turner's machine

Combinator Machines

Fixed set of combinators:

- Turner's machine
- Categorical Abstract Machine (call-by-value)

Combinator Machines

Fixed set of combinators:

- Turner's machine
- Categorical Abstract Machine (call-by-value)

Super-combinator machines:

Combinator Machines

Fixed set of combinators:

- Turner's machine
- Categorical Abstract Machine (call-by-value)

Super-combinator machines:

- The G-machine

Combinator Machines

Fixed set of combinators:

- Turner's machine
- Categorical Abstract Machine (call-by-value)

Super-combinator machines:

- The G-machine
- Categorical Multi-combinator Machine (call-by-value)

Combinator Machines

Fixed set of combinators:

- Turner's machine
- Categorical Abstract Machine (call-by-value)

Super-combinator machines:

- The G-machine
- Categorical Multi-combinator Machine (call-by-value)
- The Spineless G-machine

Combinator Machines

Fixed set of combinators:

- Turner's machine
- Categorical Abstract Machine (call-by-value)

Super-combinator machines:

- The G-machine
- Categorical Multi-combinator Machine (call-by-value)
- The Spineless G-machine

Combinator machines approach has been largely abandoned for environment machines.

Abstract Machines

What are abstract machines?

Recall the evaluation rule for small-step semantics:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

What are abstract machines?

Recall the evaluation rule for small-step semantics:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Every evaluation step requires a recursive search for an evaluation context.

What are abstract machines?

Recall the evaluation rule for small-step semantics:

$$\frac{M \longrightarrow M'}{E[M] \longmapsto E[M']}$$

Every evaluation step requires a recursive search for an evaluation context.

Instead of searching for the next redex, abstract machines keep track of the context as state.

SECD Machine

$$\textit{Machine Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

SECD Machine

$$\mathit{Machine\ Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

SECD Machine

$$\mathit{Machine\ Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

SECD Machine

$$\mathit{Machine\ Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

SECD Machine

$$\textit{Machine Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

D holds a machine state

SECD Machine

$$\textit{Machine Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

D holds a machine state

$$\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle \quad \longmapsto \quad \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle$$

SECD Machine

$$\mathit{Machine\ Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

D holds a machine state

$$\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle
\end{array}$$

SECD Machine

$$\mathit{Machine\ Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

D holds a machine state

$$\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle
\end{array}$$

SECD Machine

$$Machine\ Configuration ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

D holds a machine state

$$\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle
\end{array}$$

SECD Machine

$$\mathit{Machine\ Configuration} ::= \langle S \parallel E \parallel C \parallel D \rangle$$

S is a stack of values

E is an environment mapping variables to values

C holds a control stack

D holds a machine state

$$\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}$$

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}
$$

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}
$$

Note:

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \texttt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \texttt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}
$$

Note:

- Strictly evaluates function arguments

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}
$$

Note:

- Strictly evaluates function arguments
- Right-to-left evaluation order

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D\rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D\rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D\rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D\rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D\rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D)\rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D\rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D\rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D')\rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D'\rangle
\end{array}
$$

Note:

- Strictly evaluates function arguments
- Right-to-left evaluation order

Can be made call-by-name by adding thunk rules:

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \mathtt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \mathtt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}
$$

Note:

- Strictly evaluates function arguments
- Right-to-left evaluation order

Can be made call-by-name by adding thunk rules:

$$
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle \quad \longmapsto \quad \langle (E, N) \cdot S \parallel E \parallel M \cdot \mathtt{ap} \cdot C \parallel D \rangle
$$

SECD Machine

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle S \parallel E \parallel N \cdot M \cdot \texttt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel \lambda x.\, M \cdot C \parallel D \rangle & \longmapsto & \langle (E, \lambda x.\, M) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle (E', \lambda x.\, M) \cdot V \cdot S \parallel E \parallel \texttt{ap} \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E', x \mapsto V \parallel M \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle E(x) \cdot S \parallel E \parallel C \parallel D \rangle \\
\langle V \cdot \varepsilon \parallel E \parallel \varepsilon \parallel (S', E', C', D') \rangle & \longmapsto & \langle V \cdot S' \parallel E' \parallel C' \parallel D' \rangle
\end{array}
$$

Note:

- Strictly evaluates function arguments
- Right-to-left evaluation order

Can be made call-by-name by adding thunk rules:

$$
\begin{array}{rcl}
\langle S \parallel E \parallel M\ N \cdot C \parallel D \rangle & \longmapsto & \langle (E, N) \cdot S \parallel E \parallel M \cdot \texttt{ap} \cdot C \parallel D \rangle \\
\langle S \parallel E \parallel x \cdot C \parallel D \rangle & \longmapsto & \langle \varepsilon \parallel E' \parallel N \cdot \varepsilon \parallel (S, E, C, D) \rangle \\
& & \text{where } E(x) = (E', N)
\end{array}
$$

Curried Functions in Machines

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

λ-calculus encourages the use of curried functions *e.g.*
$(\lambda x.\lambda y.\, x + y)\ 1\ 2$

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

λ-calculus encourages the use of curried functions *e.g.* $(\lambda x.\lambda y.\, x + y)\ 1\ 2$

$$\langle 1 \cdot 2 \parallel E \parallel (\lambda x.\lambda y.\, x + y) \cdot \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle$$

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

λ-calculus encourages the use of curried functions *e.g.* $(\lambda x.\lambda y.\, x + y)\ 1\ 2$

$$\begin{aligned}&\langle 1 \cdot 2 \parallel E \parallel (\lambda x.\lambda y.\, x + y) \cdot \mathtt{ap} \cdot \mathtt{ap} \parallel D\rangle \\ &\longrightarrow \langle (E, \lambda x.\lambda y.\, x + y) \cdot 1 \cdot 2 \parallel E \parallel \mathtt{ap} \cdot \mathtt{ap} \parallel D\rangle\end{aligned}$$

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

λ-calculus encourages the use of curried functions *e.g.* $(\lambda x.\lambda y.\, x + y)\ 1\ 2$

$$\begin{array}{l}
\langle 1 \cdot 2 \parallel E \parallel (\lambda x.\lambda y.\, x + y) \cdot \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle \\
\longrightarrow \langle (E, \lambda x.\lambda y.\, x + y) \cdot 1 \cdot 2 \parallel E \parallel \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle \\
\longrightarrow \langle \parallel E, x \mapsto 1 \parallel \lambda y.\, x + y \parallel (2, E, \mathtt{ap}, D) \rangle
\end{array}$$

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

λ-calculus encourages the use of curried functions *e.g.* $(\lambda x. \lambda y.\, x + y)\ 1\ 2$

$$
\begin{aligned}
&\langle 1 \cdot 2 \parallel E \parallel (\lambda x. \lambda y.\, x + y) \cdot \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle \\
&\longrightarrow \langle (E, \lambda x. \lambda y.\, x + y) \cdot 1 \cdot 2 \parallel E \parallel \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle \\
&\longrightarrow \langle \parallel E, x \mapsto 1 \parallel \lambda y.\, x + y \parallel (2, E, \mathtt{ap}, D) \rangle \\
&\longrightarrow \langle ((E, x \mapsto 1), \lambda y.\, x + y) \parallel E, x \mapsto 1 \parallel \parallel (2, E, \mathtt{ap}, D) \rangle
\end{aligned}
$$

Curried Functions in Machines

The most important operation in the λ-calculus is function application.

λ-calculus encourages the use of curried functions *e.g.* $(\lambda x.\lambda y.\, x + y)\ 1\ 2$

$$
\begin{aligned}
&\langle 1 \cdot 2 \parallel E \parallel (\lambda x.\lambda y.\, x + y) \cdot \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle \\
&\longrightarrow \langle (E, \lambda x.\lambda y.\, x + y) \cdot 1 \cdot 2 \parallel E \parallel \mathtt{ap} \cdot \mathtt{ap} \parallel D \rangle \\
&\longrightarrow \langle \parallel E, x \mapsto 1 \parallel \lambda y.\, x + y \parallel (2, E, \mathtt{ap}, D) \rangle \\
&\longrightarrow \langle ((E, x \mapsto 1), \lambda y.\, x + y) \parallel E, x \mapsto 1 \parallel \parallel (2, E, \mathtt{ap}, D) \rangle
\end{aligned}
$$

The closure $((E, x \mapsto 1), \lambda y.\, x + y)$ is unnecessary!

Multi-arity Function Calls

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multiple arguments are pushed on the stack.

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multiple arguments are pushed on the stack.

$$\langle \texttt{Eval}\ (f\ x_0 \cdots x_n)\ E \parallel A \parallel U \parallel H \rangle \quad \longmapsto \quad \langle \texttt{Enter}\ E(f) \parallel E(x_0) \cdots E(x_n) \cdot A \parallel U \parallel H \rangle$$

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multiple arguments are pushed on the stack.

$$\langle \texttt{Eval}\ (f\ x_0 \cdots x_n)\ E \parallel A \parallel U \parallel H \rangle \quad \longmapsto \quad \langle \texttt{Enter}\ E(f) \parallel E(x_0) \cdots E(x_n) \cdot A \parallel U \parallel H \rangle$$

Entering a closure:

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multiple arguments are pushed on the stack.

$$\langle \texttt{Eval}\ (f\ x_0 \cdots x_n)\ E \parallel A \parallel U \parallel H \rangle \quad \longmapsto \quad \langle \texttt{Enter}\ E(f) \parallel E(x_0) \cdots E(x_n) \cdot A \parallel U \parallel H \rangle$$

Entering a closure:

$$\langle \texttt{Enter}\ l \parallel A \parallel U \parallel H \rangle \quad \longmapsto \quad \langle \texttt{Eval}\ M\ E \parallel \varepsilon \parallel (A, l) \cdot U \parallel H \rangle$$
$$\text{where } H(l) = (E, M)$$

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multiple arguments are pushed on the stack.

$$\langle \texttt{Eval}\ (f\ x_0 \cdots x_n)\ E \parallel A \parallel U \parallel H \rangle \quad \longmapsto \quad \langle \texttt{Enter}\ E(f) \parallel E(x_0) \cdots E(x_n) \cdot A \parallel U \parallel H \rangle$$

Entering a closure:

$$\begin{array}{rcl} \langle \texttt{Enter}\ l \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ M\ E \parallel \varepsilon \parallel (A, l) \cdot U \parallel H \rangle \\ & & \quad \text{where } H(l) = (E, M) \\ \langle \texttt{Enter}\ l \parallel \overline{V} \cdot A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ M\ (E, \overline{x \mapsto V}) \parallel A \parallel U \parallel H \rangle \\ & & \quad \text{where } H(l) = (E, \lambda \overline{x}.\ M) \\ & & \qquad\qquad |\overline{V}| = |\overline{x}| \end{array}$$

Multi-arity Function Calls

The STG (GHC) and ZINC (Ocaml) machines were designed with fast curried calls in mind.

Multiple arguments are pushed on the stack.

$$\langle \texttt{Eval}\ (f\ x_0 \cdots x_n)\ E \parallel A \parallel U \parallel H \rangle \quad \longmapsto \quad \langle \texttt{Enter}\ E(f) \parallel E(x_0) \cdots E(x_n) \cdot A \parallel U \parallel H \rangle$$

Entering a closure:

$$\begin{array}{rcl}
\langle \texttt{Enter}\ l \parallel A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ M\ E \parallel \varepsilon \parallel (A, l) \cdot U \parallel H \rangle \\
& & \quad \text{where } H(l) = (E, M) \\[1em]
\langle \texttt{Enter}\ l \parallel \overline{V} \cdot A \parallel U \parallel H \rangle & \longmapsto & \langle \texttt{Eval}\ M\ (E, \overline{x \mapsto V}) \parallel A \parallel U \parallel H \rangle \\
& & \quad \text{where } H(l) = (E, \lambda\overline{x}.\, M) \\
& & \qquad\qquad |\overline{V}| = |\overline{x}| \\[1em]
\langle \texttt{Enter}\ l \parallel \overline{V} \parallel (A', l') \cdot U \parallel H \rangle & \longmapsto & \langle \texttt{Enter}\ l \parallel \overline{V} \cdot A' \parallel U \parallel H \rangle \\
& & \quad \text{where } H(l) = (E, \lambda\overline{x}.\, M) \\
& & \qquad\qquad |\overline{V}| < |\overline{x}| \\
& & \qquad\qquad H' = H[l' \mapsto ((E, \overline{x \mapsto V}, f \mapsto l), f\ \overline{x})]
\end{array}$$

Abstract Machine Recap

Abstract Machine Recap

Abstract machines keep track of where they are in the computation.

Abstract Machine Recap

Abstract machines keep track of where they are in the computation.

Multi-arity function calls are important for functional languages.

Abstract Machine Recap

Abstract machines keep track of where they are in the computation.

Multi-arity function calls are important for functional languages. They can be accomplished with runtime argument checking.

Abstract Machine Recap

Abstract machines keep track of where they are in the computation.

Multi-arity function calls are important for functional languages. They can be accomplished with runtime argument checking.

The differences in evaluation strategy:

Abstract Machine Recap

Abstract machines keep track of where they are in the computation.

Multi-arity function calls are important for functional languages. They can be accomplished with runtime argument checking.

The differences in evaluation strategy:

- The number of closures

Abstract Machine Recap

Abstract machines keep track of where they are in the computation.

Multi-arity function calls are important for functional languages. They can be accomplished with runtime argument checking.

The differences in evaluation strategy:

- The number of closures
- Update stacks for call-by-need

Compilation through Intermediate Languages

The structure of modern functional compilers involves several intermediate languages:

The structure of modern functional compilers involves several intermediate languages:

$$\begin{array}{c}
\texttt{SML} \\
\Big\downarrow \scriptstyle{\textit{TC / Desugar}} \\
\text{System F} \\
\Big\downarrow \scriptstyle{\textit{CPS}} \\
\Lambda_{\text{CPS}} \\
\Big\downarrow \scriptstyle{\textit{Clos. Conv.}} \\
\Lambda_{\text{clos}} \\
\Big\downarrow \scriptstyle{\textit{Explicit Alloc.}} \\
\Lambda_{\text{alloc}} \\
\Big\downarrow \scriptstyle{\textit{Code Gen.}} \\
\texttt{ASM}
\end{array}$$

The structure of modern functional compilers involves several intermediate languages:

$$
\begin{tikzcd}
\texttt{SML} \arrow[d, "\mathit{TC}\,/\,\mathit{Desugar}"] \\
\text{System F} \arrow[d, "\mathit{CPS}"] \\
\Lambda_{\text{CPS}} \arrow[d, "\mathit{Clos.\ Conv.}"] \\
\Lambda_{\text{clos}} \arrow[d, "\mathit{Explicit\ Alloc.}"] \\
\Lambda_{\text{alloc}} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{ASM}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\texttt{Haskell} \arrow[d, "\mathit{TC}\,/\,\mathit{Desugar}"] \\
\text{System F}_{\text{CJ}} \arrow[d, "\mathit{Norm.}\,/\,\mathit{Erasure}"] \\
\texttt{STG} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{Cmm} \arrow[d, "\mathit{Instr.\ Select.}"] \\
\texttt{ASM}
\end{tikzcd}
$$

The structure of modern functional compilers involves several intermediate languages:

$$
\begin{tikzcd}
\texttt{SML} \arrow[d, "\mathit{TC}\,/\,\mathit{Desugar}"] \\
\text{System F} \arrow[d, "\mathit{CPS}"] \\
\Lambda_{\mathrm{CPS}} \arrow[d, "\mathit{Clos.\ Conv.}"] \\
\Lambda_{\mathrm{clos}} \arrow[d, "\mathit{Explicit\ Alloc.}"] \\
\Lambda_{\mathrm{alloc}} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{ASM}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\texttt{Haskell} \arrow[d, "\mathit{TC}\,/\,\mathit{Desugar}"] \\
\text{System } \mathrm{F}_{\mathrm{CJ}} \arrow[d, "\mathit{Norm.}\,/\,\mathit{Erasure}"] \\
\texttt{STG} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{Cmm} \arrow[d, "\mathit{Instr.\ Select.}"] \\
\texttt{ASM}
\end{tikzcd}
$$

Each language has its use:

The structure of modern functional compilers involves several intermediate languages:

$$
\begin{array}{c}
\texttt{SML} \\
\Big\downarrow \mathit{TC / Desugar} \\
\text{System F} \\
\Big\downarrow \mathit{CPS} \\
\Lambda_{\text{CPS}} \\
\Big\downarrow \mathit{Clos.\ Conv.} \\
\Lambda_{\text{clos}} \\
\Big\downarrow \mathit{Explicit\ Alloc.} \\
\Lambda_{\text{alloc}} \\
\Big\downarrow \mathit{Code\ Gen.} \\
\texttt{ASM}
\end{array}
\qquad\qquad
\begin{array}{c}
\texttt{Haskell} \\
\Big\downarrow \mathit{TC / Desugar} \\
\text{System } \mathrm{F}_{\text{CJ}} \\
\Big\downarrow \mathit{Norm. / Erasure} \\
\texttt{STG} \\
\Big\downarrow \mathit{Code\ Gen.} \\
\texttt{Cmm} \\
\Big\downarrow \mathit{Instr.\ Select.} \\
\texttt{ASM}
\end{array}
$$

Each language has its use:

- **Necessity**, required by target machine/language (closure-conversion)

The structure of modern functional compilers involves several intermediate languages:

$$
\begin{tikzcd}
\texttt{SML} \arrow[d, "\mathit{TC} / \mathit{Desugar}"] \\
\text{System F} \arrow[d, "\mathit{CPS}"] \\
\Lambda_{\mathrm{CPS}} \arrow[d, "\mathit{Clos.\ Conv.}"] \\
\Lambda_{\mathrm{clos}} \arrow[d, "\mathit{Explicit\ Alloc.}"] \\
\Lambda_{\mathrm{alloc}} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{ASM}
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{Haskell} \arrow[d, "\mathit{TC} / \mathit{Desugar}"] \\
\text{System } \mathrm{F}_{\mathrm{CJ}} \arrow[d, "\mathit{Norm.} / \mathit{Erasure}"] \\
\texttt{STG} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{Cmm} \arrow[d, "\mathit{Instr.\ Select.}"] \\
\texttt{ASM}
\end{tikzcd}
$$

Each language has its use:

- **Necessity**, required by target machine/language (closure-conversion)
- **Flexibility**, easier to perform optimizations (CPS)

The structure of modern functional compilers involves several intermediate languages:

$$
\begin{tikzcd}
\texttt{SML} \arrow[d, "\mathit{TC}/\mathit{Desugar}"] \\
\text{System F} \arrow[d, "\mathit{CPS}"] \\
\Lambda_{\mathrm{CPS}} \arrow[d, "\mathit{Clos.\ Conv.}"] \\
\Lambda_{\mathrm{clos}} \arrow[d, "\mathit{Explicit\ Alloc.}"] \\
\Lambda_{\mathrm{alloc}} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{ASM}
\end{tikzcd}
\qquad
\begin{tikzcd}
\texttt{Haskell} \arrow[d, "\mathit{TC}/\mathit{Desugar}"] \\
\text{System F}_{\mathrm{CJ}} \arrow[d, "\mathit{Norm.}/\mathit{Erasure}"] \\
\texttt{STG} \arrow[d, "\mathit{Code\ Gen.}"] \\
\texttt{Cmm} \arrow[d, "\mathit{Instr.\ Select.}"] \\
\texttt{ASM}
\end{tikzcd}
$$

Each language has its use:

- **Necessity**, required by target machine/language (closure-conversion)
- **Flexibility**, easier to perform optimizations (CPS)
- **Generality**, can be used to unify evaluation strategies (thunking,call-by-push-value)

Closure-conversion

Closure-conversion

λ-calculus functions are higher-order and can contain free variables.

Closure-conversion

λ-calculus functions are higher-order and can contain free variables.

C functions are “global” and only know of their formal parameters.

Closure-conversion

λ-calculus functions are higher-order and can contain free variables.

C functions are "global" and only know of their formal parameters.

Closure-conversion turns the former into the latter.

Closure-conversion

The transformation works by turning functions into a data structure containing a product of free variables and a combinator.

Closure-conversion

The transformation works by turning functions into a data structure containing a product of free variables and a combinator.

$$\begin{array}{rcl} \mathrm{CC}[\![\lambda x.\, M]\!] & = & ((y_0, \ldots, y_n), \lambda((y_0, \ldots, y_n), x).\, \mathrm{CC}[\![M]\!]) \\ & & \text{where } y_0, \ldots, y_n = \mathrm{FV}(\lambda x.\, M) \end{array}$$

Closure-conversion

The transformation works by turning functions into a data structure containing a product of free variables and a combinator.

$$
\begin{array}{rcl}
\mathrm{CC}[\![\lambda x.\, M]\!] & = & ((y_0, \ldots, y_n), \lambda((y_0, \ldots, y_n), x).\, \mathrm{CC}[\![M]\!]) \\
 & & \text{where } y_0, \ldots, y_n = \mathrm{FV}(\lambda x.\, M) \\
\mathrm{CC}[\![M\ N]\!] & = & \texttt{case}\ \mathrm{CC}[\![M]\!]\ \texttt{of}\ (e, f) \rightarrow f\ (e, \mathrm{CC}[\![N]\!])
\end{array}
$$

Closure-conversion

The transformation works by turning functions into a data structure containing a product of free variables and a combinator.

$$
\begin{array}{rcl}
\mathrm{CC}[\![\lambda x.\, M]\!] & = & ((y_0, \ldots, y_n), \lambda((y_0, \ldots, y_n), x).\, \mathrm{CC}[\![M]\!]) \\
 & & \text{where } y_0, \ldots, y_n = \mathrm{FV}(\lambda x.\, M) \\
\mathrm{CC}[\![M\ N]\!] & = & \texttt{case}\ \mathrm{CC}[\![M]\!]\ \texttt{of}\ (e, f) \rightarrow f\ (e, \mathrm{CC}[\![N]\!])
\end{array}
$$

In our recent work:

Closure-conversion

The transformation works by turning functions into a data structure containing a product of free variables and a combinator.

$$
\begin{array}{rcl}
\mathrm{CC}[\![\lambda x.\, M]\!] & = & ((y_0, \ldots, y_n), \lambda((y_0, \ldots, y_n), x).\, \mathrm{CC}[\![M]\!]) \\
 & & \text{where } y_0, \ldots, y_n = \mathrm{FV}(\lambda x.\, M) \\
\mathrm{CC}[\![M\ N]\!] & = & \texttt{case}\ \mathrm{CC}[\![M]\!]\ \texttt{of}\ (e, f) \to f\ (e, \mathrm{CC}[\![N]\!])
\end{array}
$$

In our recent work:

- Extend to non-strict languages by adding thunk closure-conversion

Closure-conversion

The transformation works by turning functions into a data structure containing a product of free variables and a combinator.

$$
\begin{array}{rcl}
\mathrm{CC}[\![\lambda x.\, M]\!] & = & ((y_0, \ldots, y_n), \lambda((y_0, \ldots, y_n), x).\, \mathrm{CC}[\![M]\!]) \\
 & & \text{where } y_0, \ldots, y_n = \mathrm{FV}(\lambda x.\, M) \\
\mathrm{CC}[\![M\ N]\!] & = & \texttt{case}\ \mathrm{CC}[\![M]\!]\ \texttt{of}\ (e, f) \to f\ (e, \mathrm{CC}[\![N]\!])
\end{array}
$$

In our recent work:

- Extend to non-strict languages by adding thunk closure-conversion
- Show that closure-conversion is only correct and useful if the target language is strict with closed functions

Continuation-Passing Style

Continuation-Passing Style

The call-by-value β-rule restricts inlining:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

Continuation-Passing Style

The call-by-value β-rule restricts inlining:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

After the CPS transformation every function is applied to a value:

$$\begin{array}{rcl} \mathrm{K}_{\mathcal{V}}[\![x]\!] & = & \lambda k.\, k\ x \\ \mathrm{K}_{\mathcal{V}}[\![\lambda x.\, M]\!] & = & \lambda k.\, k\ (\lambda x.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]) \\ \mathrm{K}_{\mathcal{V}}[\![M\ N]\!] & = & \lambda k.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]\ (\lambda m.\, \mathrm{K}_{\mathcal{V}}[\![N]\!]\ (\lambda n.\, m\ n\ k)) \end{array}$$

Continuation-Passing Style

The call-by-value β-rule restricts inlining:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

After the CPS transformation every function is applied to a value:

$$\begin{array}{rcl}
\mathrm{K}_{\mathcal{V}}[\![x]\!] & = & \lambda k.\, k\ x \\
\mathrm{K}_{\mathcal{V}}[\![\lambda x.\, M]\!] & = & \lambda k.\, k\ (\lambda x.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]) \\
\mathrm{K}_{\mathcal{V}}[\![M\ N]\!] & = & \lambda k.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]\ (\lambda m.\, \mathrm{K}_{\mathcal{V}}[\![N]\!]\ (\lambda n.\, m\ n\ k))
\end{array}$$

In call-by-name, β always applies.

Continuation-Passing Style

The call-by-value β-rule restricts inlining:

$$(\lambda x.\, M)\ V \longrightarrow_{\beta_{\mathcal{V}}} M[V/x]$$

After the CPS transformation every function is applied to a value:

$$\begin{array}{rcl}
\mathrm{K}_{\mathcal{V}}[\![x]\!] & = & \lambda k.\, k\ x \\
\mathrm{K}_{\mathcal{V}}[\![\lambda x.\, M]\!] & = & \lambda k.\, k\ (\lambda x.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]) \\
\mathrm{K}_{\mathcal{V}}[\![M\ N]\!] & = & \lambda k.\, \mathrm{K}_{\mathcal{V}}[\![M]\!]\ (\lambda m.\, \mathrm{K}_{\mathcal{V}}[\![N]\!]\ (\lambda n.\, m\ n\ k))
\end{array}$$

In call-by-name, β always applies.

In call-by-need, inline with β can result in a loss of sharing.

Evaluation Strategy Unifying Languages

Call-by-name and call-by-value CPSed programs can run in the same runtime.

Evaluation Strategy Unifying Languages

Call-by-name and call-by-value CPSed programs can run in the same runtime.

Approaches to unifying evaluation strategies:

- CPS

Evaluation Strategy Unifying Languages

Call-by-name and call-by-value CPSed programs can run in the same runtime.

Approaches to unifying evaluation strategies:

- CPS
- Thunking

Evaluation Strategy Unifying Languages

Call-by-name and call-by-value CPSed programs can run in the same runtime.

Approaches to unifying evaluation strategies:

- CPS
- Thunking
- Call-by-push-value

Thunking

Thunking embeds a call-by-name or call-by-need language in call-by-value language:

Thunking

Thunking embeds a call-by-name or call-by-need language in call-by-value language:

$$
\begin{array}{rcl}
\mathrm{T}[\![x]\!] & = & \texttt{force}\ x \\
\mathrm{T}[\![\lambda x.\, M]\!] & = & \lambda x.\, \mathrm{T}[\![M]\!] \\
\mathrm{T}[\![M\ N]\!] & = & \mathrm{T}[\![M]\!]\ (\texttt{delay}\ \mathrm{T}[\![N]\!])
\end{array}
$$

Thunking

Thunking embeds a call-by-name or call-by-need language in call-by-value language:

$$\begin{array}{rcl} \mathrm{T}[\![x]\!] & = & \texttt{force}\ x \\ \mathrm{T}[\![\lambda x.\, M]\!] & = & \lambda x.\, \mathrm{T}[\![M]\!] \\ \mathrm{T}[\![M\ N]\!] & = & \mathrm{T}[\![M]\!]\ (\texttt{delay}\ \mathrm{T}[\![N]\!]) \end{array}$$

Whether we have preserved a call-by-name or -need source depends on the semantics of `delay` and `force`:

Thunking

Thunking embeds a call-by-name or call-by-need language in call-by-value language:

$$\begin{array}{rcl} \mathrm{T}[\![x]\!] & = & \texttt{force}\ x \\ \mathrm{T}[\![\lambda x.\, M]\!] & = & \lambda x.\, \mathrm{T}[\![M]\!] \\ \mathrm{T}[\![M\ N]\!] & = & \mathrm{T}[\![M]\!]\ (\texttt{delay}\ \mathrm{T}[\![N]\!]) \end{array}$$

Whether we have preserved a call-by-name or -need source depends on the semantics of `delay` and `force`:

- Memoizing force for call-by-need

Thunking

Thunking embeds a call-by-name or call-by-need language in call-by-value language:

$$
\begin{array}{rcl}
\mathrm{T}[\![x]\!] & = & \texttt{force}\ x \\
\mathrm{T}[\![\lambda x.\, M]\!] & = & \lambda x.\, \mathrm{T}[\![M]\!] \\
\mathrm{T}[\![M\ N]\!] & = & \mathrm{T}[\![M]\!]\ (\texttt{delay}\ \mathrm{T}[\![N]\!])
\end{array}
$$

Whether we have preserved a call-by-name or -need source depends on the semantics of `delay` and `force`:

- Memoizing force for call-by-need
- Non-memoizing for call-by-name

Call-by-push-value

Call-by-push-value

Call-by-push-value subsumes call-by-value and call-by-name

Call-by-push-value

Call-by-push-value subsumes call-by-value and call-by-name; both compile into it.

Call-by-push-value

Call-by-push-value subsumes call-by-value and call-by-name; both compile into it.

For optimization, it has an always applicable β law for inlining applications.

Call-by-push-value

Call-by-push-value subsumes call-by-value and call-by-name; both compile into it.

For optimization, it has an always applicable β law for inlining applications.

It separates values (being) from computations (doing):

Call-by-push-value

Call-by-push-value subsumes call-by-value and call-by-name; both compile into it.

For optimization, it has an always applicable β law for inlining applications.

It separates values (being) from computations (doing):

$$V, W \in \quad \mathit{Value} \quad ::= x \mid \texttt{thunk}\ M \mid c$$

Call-by-push-value

Call-by-push-value subsumes call-by-value and call-by-name; both compile into it.

For optimization, it has an always applicable β law for inlining applications.

It separates values (being) from computations (doing):

$$
\begin{array}{rcl}
V, W \in & \mathit{Value} & ::= x \mid \mathtt{thunk}\ M \mid c \\
M, N \in & \mathit{Computation} & ::= \mathtt{return}\ V \mid \lambda x.\, M \mid M\ V \mid \mathtt{force}\ V \\
 & & \quad \mid \mathtt{let}\ x = V\ \mathtt{in}\ M \\
 & & \quad \mid M\ \mathtt{to}\ x\ \mathtt{in}\ N
\end{array}
$$

Summary

In deriving more efficient implementations of functions:

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts
- Intermediate languages are added because they are sometimes necessary,

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts
- Intermediate languages are added because they are sometimes necessary, they are for optimizations,

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts
- Intermediate languages are added because they are sometimes necessary, they are for optimizations, and/or they compile more programs.

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts
- Intermediate languages are added because they are sometimes necessary, they are for optimizations, and/or they compile more programs.

Regarding the difference in evaluation strategies:

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts
- Intermediate languages are added because they are sometimes necessary, they are for optimizations, and/or they compile more programs.

Regarding the difference in evaluation strategies:

- Non-strict strategies require thunk closures in addition to function closures.

Summary

In deriving more efficient implementations of functions:

- Environments replace substitutions
- Machines with continuations replace evaluation contexts
- Intermediate languages are added because they are sometimes necessary, they are for optimizations, and/or they compile more programs.

Regarding the difference in evaluation strategies:

- Non-strict strategies require thunk closures in addition to function closures.
- Call-by-need implementations require "heaps" and update frames.

In the Paper

More details on the following:

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics
- Closure-conversion of thunks

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics
- Closure-conversion of thunks

A section on correctness:

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics
- Closure-conversion of thunks

A section on correctness:

- Machine Reflection

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics
- Closure-conversion of thunks

A section on correctness:

- Machine Reflection
- Type preservation of transformations

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics
- Closure-conversion of thunks

A section on correctness:

- Machine Reflection
- Type preservation of transformations
- Logical relations for closure-conversion

In the Paper

More details on the following:

- De Bruijn notation and Explicit substitutions
- Lazy big-step environment semantics
- The G-, ZINC, and Krivine machines
- (Non)-memoizing thunk and call-by-push-value semantics
- Closure-conversion of thunks

A section on correctness:

- Machine Reflection
- Type preservation of transformations
- Logical relations for closure-conversion
- Issues with reasoning about memoizing heaps

Future Work

Our recent work involved non-strict closure-conversions.

Future Work

Our recent work involved non-strict closure-conversions.

Transformed a non-strict language into a strict one out of necessity.

Future Work

Our recent work involved non-strict closure-conversions.

Transformed a non-strict language into a strict one out of necessity.

Research directions:

Future Work

Our recent work involved non-strict closure-conversions.

Transformed a non-strict language into a strict one out of necessity.

Research directions:

- Partial closure-conversion

Future Work

Our recent work involved non-strict closure-conversions.

Transformed a non-strict language into a strict one out of necessity.

Research directions:

- Partial closure-conversion
- Logical relations for memoizing heaps

Future Work

Our recent work involved non-strict closure-conversions.

Transformed a non-strict language into a strict one out of necessity.

Research directions:

- Partial closure-conversion
- Logical relations for memoizing heaps

Thanks

Reasoning about Implementations

There are many ways to specify a semantics for a program:

There are many ways to specify a semantics for a program:

- Reduction theory
- Operational Semantics
- Combinator and abstract machines
- Compilation to low-level languages

There are many ways to specify a semantics for a program:

- Reduction theory
- Operational Semantics
- Combinator and abstract machines
- Compilation to low-level languages

How do we know that these are equivalent methods?

Machine Reflection

Define a translation $[\![C]\!] : \mathit{Mach} \to \Lambda$

Machine Reflection

Define a translation $\llbracket C \rrbracket : \mathit{Mach} \to \Lambda$

Theorem

If $C \longmapsto C'$, then $\llbracket C \rrbracket = \llbracket C' \rrbracket$.

Machine Reflection

Define a translation $[\![C]\!] : Mach \to \Lambda$

Theorem

If $C \longmapsto C'$, then $[\![C]\!] = [\![C']\!]$.

Only shows that machine states respect the source; nothing about whether equalities of the source are preserved.

Type Preservation

A type system for a language can guarantee properties of our source program

Type Preservation

A type system for a language can guarantee properties of our source program

Type preservation can give us a typing derivation in our target language

- Help us prove things like strong normalization
- Type information can inform code-generation and runtime systems

Logical Relations

Proving the correctness of closure-conversion is hard.

Logical Relations

Proving the correctness of closure-conversion is hard.

e.g.

$$\mathrm{CC}[\![\lambda f.\, f\ x]\!] = (x, \lambda(x, f).\, \texttt{case}\ f\ \texttt{of}\ (e, g) \rightarrow g\ (e, x))$$

Logical Relations

Proving the correctness of closure-conversion is hard.

e.g.

$$\mathrm{CC}[\![\lambda f.\, f\ x]\!] = (x, \lambda(x, f).\, \texttt{case}\ f\ \texttt{of}\ (e, g) \to g\ (e, x))$$

- These values are not closely related by our reduction rules

Logical Relations

Proving the correctness of closure-conversion is hard.

e.g.

$$\mathrm{CC}[\![\lambda f.\, f\ x]\!] = (x, \lambda(x, f).\, \texttt{case}\ f\ \texttt{of}\ (e, g) \rightarrow g\ (e, x))$$

- These values are not closely related by our reduction rules
- The parameter f does not behave like a source function

Logical Relations

Proving the correctness of closure-conversion is hard.

e.g.

$$\mathrm{CC}[\![\lambda f.\, f\ x]\!] = (x, \lambda(x, f).\, \texttt{case}\ f\ \texttt{of}\ (e, g) \rightarrow g\ (e, x))$$

- These values are not closely related by our reduction rules
- The parameter f does not behave like a source function

We must discuss relations of values and expressions.

Logical Relations

Proving the correctness of closure-conversion is hard.

e.g.

$$\mathrm{CC}[\![\lambda f.\, f\ x]\!] = (x, \lambda(x, f).\, \texttt{case}\ f\ \texttt{of}\ (e, g) \rightarrow g\ (e, x))$$

- These values are not closely related by our reduction rules
- The parameter f does not behave like a source function

We must discuss relations of values and expressions.

$$\begin{array}{rcl} \mathcal{V}[\![\tau \rightarrow \sigma]\!] & = & \{((\Sigma, V), V) \mid \forall (W, W') \in \mathcal{V}[\![\tau]\!]. \\ & & \quad (\langle \Sigma \parallel V\ W \rangle, \langle \varepsilon \parallel \pi_0(V)\ (\pi_1(V), W) \rangle) \in \mathcal{C}[\![\sigma]\!]\} \end{array}$$

Logical Relations

Proving the correctness of closure-conversion is hard.

e.g.

$$\mathrm{CC}[\![\lambda f.\, f\ x]\!] = (x, \lambda(x, f).\,\texttt{case}\ f\ \texttt{of}\ (e, g) \to g\ (e, x))$$

- These values are not closely related by our reduction rules
- The parameter f does not behave like a source function

We must discuss relations of values and expressions.

$$\begin{array}{rcl} \mathcal{V}[\![\tau \to \sigma]\!] & = & \{((\Sigma, V), V) \mid \forall (W, W') \in \mathcal{V}[\![\tau]\!]. \\ & & \qquad (\langle \Sigma \parallel V\ W \rangle, \langle \varepsilon \parallel \pi_0(V)\ (\pi_1(V), W)\rangle) \in \mathcal{C}[\![\sigma]\!]\} \end{array}$$

We found that call-by-name reasoning easily adapted to call-by-value logical relations.

Reasoning about Memoizing Heaps

Reasoning about Memoizing Heaps

As big-step and machine semantics for call-by-need require the addition of heaps.

Reasoning about Memoizing Heaps

As big-step and machine semantics for call-by-need require the addition of heaps.

There are still open questions regarding memoizing heaps:

Reasoning about Memoizing Heaps

As big-step and machine semantics for call-by-need require the addition of heaps.

There are still open questions regarding memoizing heaps:

- Objects are updated within according to the semantics

Reasoning about Memoizing Heaps

As big-step and machine semantics for call-by-need require the addition of heaps.

There are still open questions regarding memoizing heaps:

- Objects are updated within according to the semantics
- In many C dynamic memory, these are unordered structures
- Applies to both delay-force and call-by-need