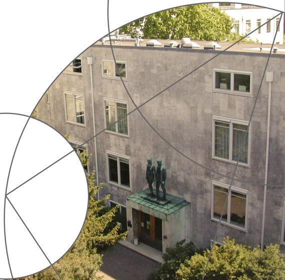
Calculating Correct Compilers

Patrick Bahr[1] Graham Hutton[2]

[1]University of Copenhagen,
Department of Computer Science
paba@diku.dk

[2]University of Nottingham,
Functional Programming Laboratory
graham.hutton@nottingham.ac.uk



Introduction

Goals

- Derive compiler implementation from denotational semantics
- Derivation by formal calculations

Introduction

Goals

- Derive compiler implementation from denotational semantics
- Derivation by formal calculations
- Result: compiler + virtual machine + correctness proof

Introduction

Goals

- Derive compiler implementation from denotational semantics
- Derivation by formal calculations
- Result: compiler + virtual machine + correctness proof

Our approach

- simple, goal-oriented calculations
- little prior knowledge needed
  (e.g. "Target machine has a stack.")
- full correctness proof as a byproduct
- wide variety of language features: arithmetic, exceptions, state, lambda calculi, loops, non-determinism, interrupts

Calculate a Compiler in 3 Steps

1. Define evaluation function in compositional manner.

Calculate a Compiler in 3 Steps

1. Define evaluation function in compositional manner.
2. Calculate a version that uses a stack and continuations.

Calculate a Compiler in 3 Steps

1. Define evaluation function in compositional manner.
2. Calculate a version that uses a stack and continuations.
3. Defunctionalise to produce a compiler and a virtual machine.

Toy Example: Simple Arithmetic Language

Step 1: Semantics of the language

Syntax

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Toy Example: Simple Arithmetic Language

Step 1: Semantics of the language

Syntax

$$\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

Semantics

$$\begin{aligned}
&\mathit{eval} && :: \mathit{Expr} \to \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) && = n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) && = \mathit{eval}\ x + \mathit{eval}\ y
\end{aligned}$$

Step 2: Transformation into CPS

Type Definitions

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$$
$$\textbf{type}\ \mathit{Cont} = \mathit{Stack} \to \mathit{Stack}$$

Step 2: Transformation into CPS

Type Definitions

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$$
$$\textbf{type}\ \mathit{Cont} = \mathit{Stack} \rightarrow \mathit{Stack}$$

$$\mathit{eval}_C :: \mathit{Expr} \rightarrow \mathit{Cont} \rightarrow \mathit{Cont}$$

Step 2: Transformation into CPS

Type Definitions

$$\textbf{type}\ Stack = [Int]$$
$$\textbf{type}\ Cont\ = Stack \to Stack$$

$$eval_C :: Expr \to Cont \to Cont$$

Specification

$$eval_C\ e\ c\ s = c\ (eval\ e : s)$$

Step 2: Transformation into CPS

Type Definitions

$$\textbf{type}\ \mathit{Stack} = [\mathit{Int}]$$
$$\textbf{type}\ \mathit{Cont} = \mathit{Stack} \rightarrow \mathit{Stack}$$

$$\mathit{eval}_C :: \mathit{Expr} \rightarrow \mathit{Cont} \rightarrow \mathit{Cont}$$

Specification

$$\mathit{eval}_C\ e\ c\ s = c\ (\mathit{eval}\ e : s)$$

Constructive induction: “prove” specification by induction on e

Step 2: Transformation into CPS

Type Definitions

type *Stack* = [*Int*]
type *Cont* = *Stack* → *Stack*

$eval_C :: Expr \to Cont \to Cont$

Specification

$$eval_C\ e\ c\ s = c\ (eval\ e : s)$$

Constructive induction: "prove" specification by induction on e

⇝ definition of $eval_C$

The easy case: *Val*

$eval_C\ (Val\ n)\ c\ s$

The easy case: *Val*

$$
\begin{aligned}
& eval_C\ (Val\ n)\ c\ s \\
= \quad & \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Val\ n) : s)
\end{aligned}
$$

The easy case: *Val*

$$eval_C\ e\ c\ s = c\ (eval\ e : s)$$

$$
\begin{aligned}
& eval_C\ (Val\ n)\ c\ s \\
= \quad & \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Val\ n) : s)
\end{aligned}
$$

The easy case: *Val*

$$
\begin{aligned}
& eval_C\ (Val\ n)\ c\ s \\
= \quad & \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Val\ n) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& c\ (n : s)
\end{aligned}
$$

The easy case: *Val*

$$
\begin{aligned}
& eval_C\ (Val\ n)\ c\ s \\
=\ & \quad \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Val\ n) : s) \\
=\ & \quad \{\ \text{definition of } eval\ \} \\
& c\ (n : s)
\end{aligned}
$$

$eval\ (Val\ n) = n$

The easy case: *Val*

$$
\begin{aligned}
& eval_C\ (Val\ n)\ c\ s \\
= \quad & \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Val\ n) : s) \\
= \quad & \{\ \text{definition of } eval\ \} \\
& c\ (n : s) \\
= \quad & \{\ \text{define: } push\ n\ c\ s = c\ (n : s)\ \} \\
& push\ n\ c\ s
\end{aligned}
$$

The interesting case: *Add*

$$eval_C\ (Add\ x\ y)\ c\ s$$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= & \quad \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Add\ x\ y) : s)
\end{aligned}
$$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= \quad & \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Add\ x\ y) : s)
\end{aligned}
$$

$eval_C\ e\ c\ s = c\ (eval\ e : s)$

The interesting case: *Add*

$$
\begin{aligned}
& \mathit{eval}_{\mathsf{C}}\ (\mathit{Add}\ x\ y)\ c\ s \\
=\quad & \{\ \text{specification of } \mathit{eval}_{\mathsf{C}}\ \} \\
& c\ (\mathit{eval}\ (\mathit{Add}\ x\ y) : s) \\
=\quad & \{\ \text{definition of } \mathit{eval}\ \} \\
& c\ ((\mathit{eval}\ x + \mathit{eval}\ y) : s)
\end{aligned}
$$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= & \quad \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Add\ x\ y) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& c\ ((eval\ x + eval\ y) : s)
\end{aligned}
$$

$eval\ (Add\ x\ y) = eval\ x + eval\ y$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= & \quad \{\ \text{specification of } eva \\
& c\ (eval\ (Add\ x\ y) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& c\ ((eval\ x + eval\ y) : s)
\end{aligned}
$$

Induction Hypothesis

For all c' and s':

$eval_C\ x\ c'\ s' = c'\ (eval\ x : s')$

$eval_C\ y\ c'\ s' = c'\ (eval\ y : s')$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= & \quad \{ \text{ specification of } eval_C \ \} \\
& c\ (eval\ (Add\ x\ y) : s) \\
= & \quad \{ \text{ definition of } eval \ \} \\
& c\ ((eval\ x + eval\ y) : s) \\
= & \quad \{ \text{ define: } add\ c\ (n : m : s) = c\ ((m + n) : s) \ \} \\
& add\ c\ (eval\ y : eval\ x : s)
\end{aligned}
$$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= &\quad \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Add\ x\ y) : s) \\
= &\quad \{\ \text{definition of } eval\ \} \\
& c\ ((eval\ x + eval\ y) : s) \\
= &\quad \{\ \text{define: } add\ c\ (n : m : s) = c \\
& add\ c\ (eval\ y : eval\ x : s) \\
= &\quad \{\ \text{induction hypothesis for } y\ \} \\
& eval_C\ y\ (add\ c)\ (eval\ x : s)
\end{aligned}
$$

$eval_C\ y\ c'\ s' = c'\ (eval\ y : s')$

The interesting case: *Add*

$$
\begin{aligned}
& eval_C\ (Add\ x\ y)\ c\ s \\
= & \quad \{\ \text{specification of } eval_C\ \} \\
& c\ (eval\ (Add\ x\ y) : s) \\
= & \quad \{\ \text{definition of } eval\ \} \\
& c\ ((eval\ x + eval\ y) : s) \\
= & \quad \{\ \text{define: } add\ c\ (n : m : s) = c\ ((m + n) : s)\ \} \\
& add\ c\ (eval\ y : eval\ x : s) \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
& eval_C\ y\ (add\ c)\ (eval\ x : s) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
& eval_C\ x\ (eval_C\ y\ (add\ c))\ s
\end{aligned}
$$

$eval_C\ x\ c'\ s' = c'\ (eval\ x : s')$

Step 2: Transformation into CPS (cont.)

Derived definition

$$
\begin{aligned}
&eval_C :: Expr \to Cont \to Cont\\
&eval_C\ (Val\ n)\quad\ \ c\ \mathsf{s} = push\ n\ c\ \mathsf{s}\\
&eval_C\ (Add\ x\ y)\ c\ \mathsf{s} = eval_C\ x\ (eval_C\ y\ (add\ c))\ \mathsf{s}
\end{aligned}
$$

Step 2: Transformation into CPS (cont.)

Derived definition

$$
\begin{aligned}
&\mathit{eval}_C :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Cont} \\
&\mathit{eval}_C\ (\mathit{Val}\ n)\quad\ \ c \quad = \mathit{push}\ n\ c \\
&\mathit{eval}_C\ (\mathit{Add}\ x\ y)\ c \quad = \mathit{eval}_C\ x\ (\mathit{eval}_C\ y\ (\mathit{add}\ c))
\end{aligned}
$$

Step 2: Transformation into CPS (cont.)

Derived definition

$$
\begin{aligned}
&eval_C :: Expr \to Cont \to Cont \\
&eval_C\ (Val\ n)\quad c = push\ n\ c \\
&eval_C\ (Add\ x\ y)\ c = eval_C\ x\ (eval_C\ y\ (add\ c))
\end{aligned}
$$

$$
\begin{aligned}
&push :: Int \to Cont \to Cont \\
&push\ n\ c\ s = c\ (n : s)
\end{aligned}
$$

$$
\begin{aligned}
&add :: Cont \to Cont \\
&add\ c\ (n : m : s) = c\ ((m + n) : s)
\end{aligned}
$$

Step 2: Transformation into CPS (cont.)

Derived definition

$$
\begin{aligned}
&eval_C :: Expr \to Cont \to Cont\\
&eval_C\ (Val\ n)\quad c\ = push\ n\ c\\
&eval_C\ (Add\ x\ y)\ c\ = eval_C\ x\ (eval_C\ y\ (add\ c))
\end{aligned}
$$

$$
\begin{aligned}
&push :: Int \to Cont \to Cont\\
&push\ n\ c\ s = c\ (n : s)
\end{aligned}
$$

$$
\begin{aligned}
&add :: Cont \to Cont\\
&add\ c\ (n : m : s) = c\ ((m + n) : s)
\end{aligned}
$$

Identity continuation

$$
\begin{aligned}
&eval_S :: Expr \to Cont\\
&eval_S\ e = eval_C\ e\ halt
\end{aligned}
$$

$$
\begin{aligned}
&halt :: Cont\\
&halt\ s = s
\end{aligned}
$$

Step 3: Defunctionalisation

$$
\begin{aligned}
&\mathit{eval}_{\mathsf{S}} :: \mathit{Expr} \to \mathsf{Cont} \\
&\mathit{eval}_{\mathsf{S}}\ e = \mathit{eval}_{\mathsf{C}}\ e\ \mathsf{halt}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval}_{\mathsf{C}} :: \mathit{Expr} \to \mathsf{Cont} \to \mathsf{Cont} \\
&\mathit{eval}_{\mathsf{C}}\ (\mathit{Val}\ n)\ c = \mathsf{push}\ n\ c \\
&\mathit{eval}_{\mathsf{C}}\ (\mathit{Add}\ x\ y)\ c = \mathit{eval}_{\mathsf{C}}\ x\ (\mathit{eval}_{\mathsf{C}}\ y\ (\mathsf{add}\ c))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{halt} :: \mathsf{Cont} \\
&\mathsf{push} :: \mathit{Int} \to \mathsf{Cont} \to \mathsf{Cont} \\
&\mathsf{add} :: \mathsf{Cont} \to \mathsf{Cont}
\end{aligned}
$$

Step 3: Defunctionalisation

$$
\begin{aligned}
&eval_S :: Expr \to \mathsf{Cont} \\
&eval_S\ e = eval_C\ e\ \mathsf{halt}
\end{aligned}
$$

$$
\begin{aligned}
&eval_C :: Expr \to \mathsf{Cont} \to \mathsf{Cont} \\
&eval_C\ (Val\ n)\ c = \mathsf{push}\ n\ c \\
&eval_C\ (Add\ x\ y)\ c = eval_C\ x\ (eval_C\ y\ (\mathsf{add}\ c))
\end{aligned}
$$

data *Code* **where**

HALT :: Code

PUSH :: *Int* → Code → Code

ADD :: Code → Code

Step 3: Defunctionalisation

$$eval_S :: Expr \to \mathrm{Cont}$$
$$eval_S\ e = eval_C\ e\ \mathrm{halt}$$

$$eval_C :: Expr \to \mathrm{Cont} \to \mathrm{Cont}$$
$$eval_C\ (Val\ n)\ c = \mathrm{push}\ n\ c$$
$$eval_C\ (Add\ x\ y)\ c = eval_C\ x\ (eval_C\ y\ (\mathrm{add}\ c))$$

data *Code* **where**
HALT :: Code
PUSH :: *Int* → Code → Code
ADD :: Code → Code

Or equivalently:

data *Code = HALT | PUSH Int Code | ADD Code Code*

Step 3: Defunctionalisation

$$eval_S :: Expr \to \text{Code}$$
$$eval_S\ e = eval_C\ e\ \text{HALT}$$

$$eval_C :: Expr \to \text{Code} \to \text{Code}$$
$$eval_C\ (Val\ n)\ c = \text{PUSH}\ n\ c$$
$$eval_C\ (Add\ x\ y)\ c = eval_C\ x\ (eval_C\ y\ (\text{ADD}\ c))$$

data *Code* **where**
HALT :: Code
PUSH :: *Int* → Code → Code
ADD :: Code → Code

Or equivalently:

data *Code = HALT | PUSH Int Code | ADD Code Code*

Step 3: Defunctionalisation

$$\text{comp} :: \mathit{Expr} \to \text{Code}$$
$$\text{comp}\ e = \text{comp'}\ e\ \text{HALT}$$

$$\text{comp'} :: \mathit{Expr} \to \text{Code} \to \text{Code}$$
$$\text{comp'}\ (\mathit{Val}\ n)\quad c = \text{PUSH}\ n\ c$$
$$\text{comp'}\ (\mathit{Add}\ x\ y)\ c = \text{comp'}\ x\ (\text{comp'}\ y\ (\text{ADD}\ c))$$

data *Code* **where**
HALT :: Code
PUSH :: *Int* → Code → Code
ADD :: Code → Code

Or equivalently:

data *Code* = *HALT* | *PUSH Int Code* | *ADD Code Code*

Step 3: Defunctionalisation

$$\text{comp} :: \mathit{Expr} \to \text{Code}$$
$$\text{comp}\ e = \text{comp'}\ e\ \text{HALT}$$

$$\text{comp'} :: \mathit{Expr} \to \text{Code} \to \text{Code}$$
$$\text{comp'}\ (\mathit{Val}\ n)\quad c = \text{PUSH}\ n\ c$$
$$\text{comp'}\ (\mathit{Add}\ x\ y)\ c = \text{comp'}\ x\ (\text{comp'}\ y\ (\text{ADD}\ c))$$

data *Code* **where**

$$\text{HALT} :: \text{Code}$$
$$\text{PUSH} :: \mathit{Int} \to \text{Code} \to \text{Code}$$
$$\text{ADD} :: \text{Code} \to \text{Code}$$

Example

$$\mathit{comp}\ (\mathit{Val}\ 1\ \text{`}\mathit{Add}\text{`}\ \mathit{Val}\ 2) \rightsquigarrow \mathit{PUSH}\ 1\ \$\ \mathit{PUSH}\ 2\ \$\ \mathit{ADD}\ \$\ \mathit{HALT}$$

Step 3: Defunctionalisation (cont.)

data *Code* **where**
$HALT :: Code$
$PUSH :: Int \to Code \to Code$
$ADD :: Code \to Code$

Type *Code* represents the function type *Cont* ($= Stack \to Stack$).

Step 3: Defunctionalisation (cont.)

$$\begin{array}{l}\textbf{data}\ \mathit{Code}\ \textbf{where} \\ \quad \mathit{HALT} :: \mathit{Code} \\ \quad \mathit{PUSH} :: \mathit{Int} \to \mathit{Code} \to \mathit{Code} \\ \quad \mathit{ADD} \ \ :: \mathit{Code} \to \mathit{Code}\end{array}$$

Type *Code* represents the function type *Cont* ($= \mathit{Stack} \to \mathit{Stack}$).

Interpretation function

$$\begin{array}{ll}\mathit{exec} :: \mathit{Code} \to \mathit{Cont} & \\ \mathit{exec}\ \mathit{HALT} & = \mathit{halt} \\ \mathit{exec}\ (\mathit{PUSH}\ n\ c) & = \mathit{push}\ n\ (\mathit{exec}\ c) \\ \mathit{exec}\ (\mathit{ADD}\ c) & = \mathit{add}\ (\mathit{exec}\ c)\end{array}$$

Step 3: Defunctionalisation (cont.)

$$\begin{array}{l}
\textbf{data}\ \mathit{Code}\ \textbf{where} \\
\quad \mathit{HALT} :: \mathit{Code} \\
\quad \mathit{PUSH} :: \mathit{Int} \rightarrow \mathit{Code} \rightarrow \mathit{Code} \\
\quad \mathit{ADD}\ \ :: \mathit{Code} \rightarrow \mathit{Code}
\end{array}$$

Type *Code* represents the function type *Cont* ($= \mathit{Stack} \rightarrow \mathit{Stack}$).

Interpretation function

$$\begin{array}{lcl}
\mathit{exec} :: \mathit{Code} \rightarrow \mathit{Cont} & & \\
\mathit{exec}\ \mathit{HALT} \qquad\qquad\quad s & = & s \\
\mathit{exec}\ (\mathit{PUSH}\ n\ c) \qquad s & = & \mathit{exec}\ c\ (n : s) \\
\mathit{exec}\ (\mathit{ADD}\ c)\ (n : m : s) & = & \mathit{exec}\ c\ ((m + n) : s)
\end{array}$$

Step 3: Defunctionalisation (cont.)

$$
\begin{aligned}
&\textbf{data } \mathit{Code} \textbf{ where}\\
&\quad \mathit{HALT} :: \mathit{Code}\\
&\quad \mathit{PUSH} :: \mathit{Int} \to \mathit{Code} \to \mathit{Code}\\
&\quad \mathit{ADD} \;\; :: \mathit{Code} \to \mathit{Code}
\end{aligned}
$$

Type *Code* represents the function type *Cont* ($= \mathit{Stack} \to \mathit{Stack}$).

Virtual Machine

$$
\begin{aligned}
&\mathit{exec} :: \mathit{Code} \to \mathit{Cont}\\
&\mathit{exec}\ \mathit{HALT} \qquad\qquad\quad s \;= s\\
&\mathit{exec}\ (\mathit{PUSH}\ n\ c) \qquad s \;= \mathit{exec}\ c\ (n : s)\\
&\mathit{exec}\ (\mathit{ADD}\ c)\ (n : m : s) = \mathit{exec}\ c\ ((m + n) : s)
\end{aligned}
$$

Compiler Correctness

$$eval_C\ e\ c\ s = c\ (eval\ e : s) \quad \text{(Specification)}$$

Compiler Correctness

proved by constructive induction

$$eval_C\ e\ c\ s = c\ (eval\ e : s) \quad \text{(Specification)}$$

Compiler Correctness

$$eval_{\mathsf{C}}\ e\ c\ s = c\ (eval\ e : s) \quad \text{(Specification)}$$

$$exec\ (comp\ e)\ s = eval_{\mathsf{S}}\ e\ s \quad \text{(Defunctionalisation)}$$

Compiler Correctness

$$
\begin{aligned}
eval_{\mathsf{C}}\ e\ c\ s &= c\ (eval\ e : s) && \text{(Specification)}\\
exec\ (comp\ e)\ s &= eval_{\mathsf{S}}\ e\ s && \text{(Defunctionalisation)}\\
eval_{\mathsf{S}}\ e &= eval_{\mathsf{C}}\ e\ halt && \text{(Definition of } eval_{\mathsf{S}}\text{)}
\end{aligned}
$$

Compiler Correctness

$$\begin{array}{rl} eval_C\ e\ c\ s = c\ (eval\ e : s) & \text{(Specification)} \\ exec\ (comp\ e)\ s = eval_S\ e\ s & \text{(Defunctionalisation)} \\ eval_S\ e = eval_C\ e\ halt & \text{(Definition of } eval_S\text{)} \\ \hline\hline exec\ (comp\ e)\ s = eval\ e : s & \text{(Compiler correctness)} \end{array}$$

A Language with Exceptions

Skip this

$$\textbf{data}\ Expr = Val\ Int \mid Add\ \ Expr\ Expr \mid Throw \mid Catch\ Expr\ Expr$$

A Language with Exceptions

Skip this

$$\begin{aligned}
&\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}\\
&\qquad\qquad \mid \mathit{Throw} \mid \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}$$

$$\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Int}\\
&\mathit{eval}\ (\mathit{Val}\ n) = \mathit{Just}\ n\\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\
&\qquad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of}\\
&\qquad\qquad \mathit{Nothing} \to \mathit{Nothing}\\
&\qquad\qquad \mathit{Just}\ m \to \mathit{Just}\ (n + m)\\
&\mathit{eval}\ \mathit{Throw} = \mathit{Nothing}\\
&\mathit{eval}\ (\mathit{Catch}\ x\ h) = \textbf{case}\ \mathit{eval}\ x\ \textbf{of}\\
&\qquad \mathit{Nothing} \to \mathit{eval}\ h\\
&\qquad \mathit{Just}\ n \to \mathit{Just}\ n
\end{aligned}$$

A Language with Exceptions

▸ Skip this

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\
\mid \mathit{Throw} \mid \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Int} \\
&\mathit{eval}\ (\mathit{Val}\ n) = \mathit{Just}\ n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) = \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
&\qquad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad \mathit{Just}\ n \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
&\qquad\qquad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad\qquad \mathit{Just}\ m \to \mathit{Just}\ (n + m) \\
&\mathit{eval}\ \mathit{Throw} = \mathit{Nothing} \\
&\mathit{eval}\ (\mathit{Catch}\ x\ h) = \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
&\qquad \mathit{Nothing} \to \mathit{eval}\ h \\
&\qquad \mathit{Just}\ n \to \mathit{Just}\ n
\end{aligned}
$$

Partial Specifications

Partial Type Definition

$$\textbf{type}\ \mathit{Stack} = [\mathit{Elem}]$$
$$\textbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Int} \mid \ldots$$

Partial Specifications

Partial Type Definition

$$\textbf{type } \mathit{Stack} = [\mathit{Elem}]$$
$$\textbf{data } \mathit{Elem} = \mathit{VAL}\ \mathit{Int} \mid \ldots$$

Partial Specification of $eval_C$

$$eval_C\ e\ c\ s = c\ (eval\ e : s)$$

Partial Specifications

Partial Type Definition

$$\textbf{type}\ Stack = [Elem]$$
$$\textbf{data}\ Elem = VAL\ Int \mid \ldots$$

Partial Specification of $eval_C$

$$eval_C\ e\ c\ s = c\ (VAL\ n : s) \qquad \text{if } eval\ e = Just\ n$$
$$eval_C\ e\ c\ s = ?? \qquad \text{if } eval\ e = Nothing$$

Partial Specifications

Partial Type Definition

$$\textbf{type}\ \mathit{Stack} = [\mathit{Elem}]$$
$$\textbf{data}\ \mathit{Elem} = \mathit{VAL}\ \mathit{Int} \mid \ldots$$

Partial Specification of $eval_C$

$$eval_C\ e\ c\ s = c\ (VAL\ n : s) \qquad \text{if } eval\ e = Just\ n$$
$$eval_C\ e\ c\ s = fail\ s \qquad \text{if } eval\ e = Nothing$$

where $fail :: Stack \rightarrow Stack$ is left unspecified

Resulting Compiler

$$
\begin{aligned}
&\mathit{comp} && :: \mathit{Expr} \rightarrow \mathit{Code} \\
&\mathit{comp}\ e && = \mathit{comp}'\ e\ \mathit{HALT} \\
&\mathit{comp}' && :: \mathit{Expr} \rightarrow \mathit{Code} \rightarrow \mathit{Code} \\
&\mathit{comp}'\ (\mathit{Val}\ n)\ c && = \mathit{PUSH}\ n\ c \\
&\mathit{comp}'\ (\mathit{Add}\ x\ y)\ c && = \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD}\ c)) \\
&\mathit{comp}'\ \mathit{Throw}\ c && = \mathit{FAIL} \\
&\mathit{comp}'\ (\mathit{Catch}\ x\ h)\ c && = \mathit{MARK}\ (\mathit{comp}'\ h\ c)\ (\mathit{comp}'\ x\ (\mathit{UNMARK}\ c))
\end{aligned}
$$

Resulting Virtual Machine

$$
\begin{aligned}
&\mathit{exec} && :: \mathit{Code} \rightarrow \mathit{Cont} \\
&\mathit{exec}\ (\mathit{PUSH}\ n\ c) && s\ = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
&\mathit{exec}\ (\mathit{MARK}\ h\ c) && s\ = \mathit{exec}\ c\ (\mathit{HAN}\ h : s) \\
& && \quad\vdots \\
&\mathit{exec}\ \mathit{FAIL} && s\ = \mathit{fail}\ s
\end{aligned}
$$

Resulting Virtual Machine

$$
\begin{aligned}
&\mathit{exec} && :: \mathit{Code} \to \mathit{Cont} \\
&\mathit{exec}\ (\mathit{PUSH}\ n\ c)\ \ s && = \mathit{exec}\ c\ (\mathit{VAL}\ n : s) \\
&\mathit{exec}\ (\mathit{MARK}\ h\ c)\ \ s && = \mathit{exec}\ c\ (\mathit{HAN}\ h : s) \\
& && \vdots \\
&\mathit{exec}\ \mathit{FAIL}\ \ s && = \mathit{fail}\ s
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fail} :: \mathit{Cont} \\
&\mathit{fail}\ (\mathit{VAL}\ n : s) && = \mathit{fail}\ s \\
&\mathit{fail}\ (\mathit{HAN}\ h : s) && = \mathit{exec}\ h\ s \\
&\mathit{fail}\ [\,] && = [\,]
\end{aligned}
$$

Summary

- simple, goal-oriented calculations; no magic
- little prior knowledge needed
  (by using partial specifications)
- full correctness proof
- formalisation in Coq
- scales to wide variety of language features

Summary

- simple, goal-oriented calculations; no magic
- little prior knowledge needed
  (by using partial specifications)
- full correctness proof
- formalisation in Coq
- scales to wide variety of language features
  - arithmetic
  - exceptions (synchronous, asynchronous)
  - state (local, global)
  - lambda calculi (call-by-value, -name, -need)
  - loops (bounded, unbounded)
  - non-determinism

Future work

- Simplify reasoning for "cyclic" features (fixed points, loops)
- Simplify reasoning register machines
- Support for sharing (i.e. graph structures)
- Derivation of compilers for fixed instruction sets