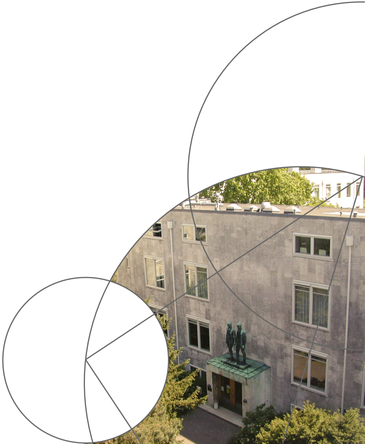
Calculating Correct Compilers

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`



Goal

Calculate a compiler that is correct by construction

Goal

Calculate a compiler that is correct by construction:

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

Goal

Calculate a compiler that is correct by construction:

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

Background

Reasoning about compilers, Hutton & Wright

- Verifying a compiler for a simple language with exceptions (MPC '04)
- Calculating an abstract machine that is correct by construction (TFP '05)

Background

Reasoning about compilers, Hutton & Wright

- Verifying a compiler for a simple language with exceptions (MPC '04)
- Calculating an abstract machine that is correct by construction (TFP '05)

Last 2.1 meeting, Hutton & Danielsson

- Calculating a compiler for a simple language with exceptions
- Use of dependent types during the calculation

This Talk: A Simplified Approach

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. “Target machine has a stack.”)
- scales to wide variety of language features

This Talk: A Simplified Approach

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. "Target machine has a stack.")
- scales to wide variety of language features:
  - arithmetic expressions
  - exceptions (synchronous and asynchronous)
  - state (global and local)
  - lambda calculi (call-by-value, call-by-name, call-by-need)
  - loops (bounded and unbounded)
  - non-determinism

This Talk: A Simplified Approach

- simple calculations without the need for dependent types
- little prior knowledge needed
  (e.g. "Target machine has a stack.")
- scales to wide variety of language features:
  - arithmetic expressions
  - exceptions (synchronous and asynchronous)
  - state (global and local)
  - lambda calculi (call-by-value, call-by-name, call-by-need)
  - loops (bounded and unbounded)
  - non-determinism
- Underlying techniques: continuation-passing style & defunctionalisation (Reynolds, 1972)

How Does it Work?

Calculate a Compiler in 3 Steps:

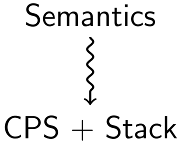
1. Define evaluation function in compositional manner.

Semantics

How Does it Work?

Calculate a Compiler in 3 Steps:

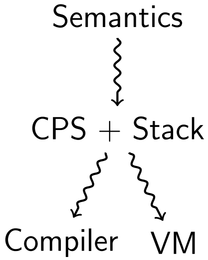
1. Define evaluation function in compositional manner.
2. Calculate a version that uses a stack and continuations.



How Does it Work?

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 & 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

$$\begin{aligned}&\textbf{type}\ \mathit{Stack} = [\mathit{Int}]\\&\textbf{type}\ \mathit{Cont}\ \ = \mathit{Stack} \to \mathit{Stack}\end{aligned}$$

Step 2: Transformation into CPS

Type Definitions

$$\begin{aligned}&\textbf{type}\ \mathit{Stack} = [\mathit{Int}]\\&\textbf{type}\ \mathit{Cont}\ \ = \mathit{Stack} \to \mathit{Stack}\end{aligned}$$

$$\mathit{eval}_{\mathsf{C}} :: \mathit{Expr} \to \mathit{Cont} \to \mathit{Cont}$$

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

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

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

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

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

Specification

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

Constructive induction: “prove” specification by induction on e

$\rightsquigarrow$ definition of eval_C

The easy case: *Val*

$$eval_{C}\ (Val\ n)\ c\ s$$

The easy case: *Val*

$$
\begin{array}{ll}
 & \mathit{eval}_C\ (\mathit{Val}\ n)\ c\ s \\
= & \quad \{\ \text{specification of } \mathit{eval}_C\ \} \\
 & c\ (\mathit{eval}\ (\mathit{Val}\ n) : s)
\end{array}
$$

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*

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

The easy case: *Val*

$eval_C\ (Val\ n)\ c\ s$
$=$ { specification of *eval*
$c\ (eval\ (Val\ n) : s)$
$=$ { definition of *eval* }
$c\ (n : s)$

$eval\ (Val\ n) = n$

The easy case: *Val*

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

The interesting case: *Add*

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

The interesting case: *Add*

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

The interesting case: *Add*

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

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

The interesting case: *Add*

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

The interesting case: *Add*

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

$\mathit{eval}\ (\mathit{Add}\ x\ y) = \mathit{eval}\ x + \mathit{eval}\ y$

The interesting case: *Add*

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

Induction Hypothesis

For all c' and s':

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

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

The interesting case: *Add*

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

The interesting case: *Add*

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

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

The interesting case: *Add*

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

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

Step 2: Transformation into CPS (cont.)

Derived definition

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

Step 2: Transformation into CPS (cont.)

Derived definition

$$\begin{array}{lll} eval_C :: Expr \rightarrow Cont \rightarrow Cont & & \\ eval_C\ (Val\ n) & c & = push\ n\ c \\ eval_C\ (Add\ x\ y) & c & = eval_C\ x\ (eval_C\ y\ (add\ c)) \end{array}$$

Step 2: Transformation into CPS (cont.)

Derived definition

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

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

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

Step 2: Transformation into CPS (cont.)

Derived definition

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

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

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

Identity continuation

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

Step 3: Defunctionalisation

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

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

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

Step 3: Defunctionalisation

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

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

data *Code* **where**

HALT :: Code

PUSH :: *Int* → Code → Code

ADD :: Code → Code

Step 3: Defunctionalisation

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

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

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

Or equivalently:

$$\mathbf{data}\ \mathit{Code} = \mathit{HALT} \mid \mathit{PUSH\ Int\ Code} \mid \mathit{ADD\ Code\ Code}$$

Step 3: Defunctionalisation

$$
\begin{aligned}
&\mathit{eval}_{\mathrm{S}} :: \mathit{Expr} \to \mathrm{Code}\\
&\mathit{eval}_{\mathrm{S}}\ e = \mathit{eval}_{\mathrm{C}}\ e\ \mathrm{HALT}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval}_{\mathrm{C}} :: \mathit{Expr} \to \mathrm{Code} \to \mathrm{Code}\\
&\mathit{eval}_{\mathrm{C}}\ (\mathit{Val}\ n)\quad\ \ c = \mathrm{PUSH}\ n\ c\\
&\mathit{eval}_{\mathrm{C}}\ (\mathit{Add}\ x\ y)\ c = \mathit{eval}_{\mathrm{C}}\ x\ (\mathit{eval}_{\mathrm{C}}\ y\ (\mathrm{ADD}\ c))
\end{aligned}
$$

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

```
comp :: Expr → Code
comp e = comp' e HALT

comp' :: Expr → Code → Code
comp' (Val n)     c = PUSH n c
comp' (Add x y) c = comp' x (comp' y (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

```
comp :: Expr → Code
comp e = comp' e HALT

comp' :: Expr → Code → Code
comp' (Val n)     c = PUSH n c
comp' (Add x y) c = comp' x (comp' y (ADD c))
```

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

Example

comp (*Val* 1 ‘*Add*‘ *Val* 2) $\rightsquigarrow$ *PUSH* 1 \$ *PUSH* 2 \$ *ADD* \$ *HALT*

Step 3: Defunctionalisation (cont.)

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

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

Step 3: Defunctionalisation (cont.)

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

Type *Code* represents the function type *Cont* (= *Stack* → *Stack*).

Interpretation function

```
exec :: Code → Cont
exec HALT         = halt
exec (PUSH n c) = push n (exec c)
exec (ADD c)      = add (exec c)
```

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

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

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* ($= \mathit{Stack} \to \mathit{Stack}$).

Virtual Machine

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathit{exec} :: \mathit{Code} \to \mathit{Cont}} \\
\mathit{exec}\ \mathit{HALT}\qquad\qquad\quad s & = & s \\
\mathit{exec}\ (\mathit{PUSH}\ n\ c)\qquad s & = & \mathit{exec}\ c\ (n : s) \\
\mathit{exec}\ (\mathit{ADD}\ c)\ (n : m : s) & = & \mathit{exec}\ c\ ((m + n) : s)
\end{array}
$$

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

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

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

$$eval_{\mathsf{S}}\ e = eval_{\mathsf{C}}\ e\ halt \quad \text{(Definition of } eval_{\mathsf{S}}\text{)}$$

Compiler Correctness

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

A Language with Exceptions

▸ Skip this

$$
\begin{array}{rll}
\textbf{data}\ \mathit{Expr} = & \mathit{Val}\ \mathit{Int} & \mid \mathit{Add}\quad \mathit{Expr}\ \mathit{Expr} \\
\mid & \mathit{Throw} & \mid \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

A Language with Exceptions

▸ Skip this

```
data Expr = Val Int | Add   Expr Expr
          | Throw   | Catch Expr Expr

eval :: Expr → Maybe Int
eval (Val n)     = Just n
eval (Add x y)   = case eval x of
                     Nothing → Nothing
                     Just n  → case eval y of
                                 Nothing → Nothing
                                 Just m  → Just (n + m)
eval  Throw      = Nothing
eval (Catch x h) = case eval x of
                     Nothing → eval h
                     Just n  → Just n
```

A Language with Exceptions

▸ Skip this

```
data Expr = Val Int | Add   Expr Expr
          | Throw   | Catch Expr Expr

eval :: Expr → Maybe Int
eval (Val n)     = Just n
eval (Add x y)   = case eval x of
                     Nothing → Nothing
                     Just n  → case eval y of
                                 Nothing → Nothing
                                 Just m  → Just (n + m)
eval  Throw      = Nothing
eval (Catch x h) = case eval x of
                     Nothing → eval h
                     Just n  → Just n
```

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

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

Partial Specification of $eval_C$

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

Partial Specifications

Partial Type Definition

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

Partial Specification of $eval_C$

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

Partial Specifications

Partial Type Definition

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

Partial Specification of $eval_C$

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

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

Constructive Induction: *Add*

▸ Skip this

$$
\begin{array}{ll}
 & eval_C\ (Add\ x\ y)\ c\ s \\
= & \quad \{\ \text{specification}\ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Just\ n \quad \to \textbf{case}\ eval\ y\ \textbf{of} \\
 & \qquad\qquad\qquad Just\ m \quad \to c\ (VAL\ (n+m) : s) \\
 & \qquad\qquad\qquad Nothing \to fail\ s \\
 & \quad Nothing \to fail\ s \\
= & \quad \{\ \text{define:}\ add\ c\ (VAL\ m : VAL\ n : s) = c\ (VAL\ (n+m) : s)\ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Just\ n \quad \to \textbf{case}\ eval\ y\ \textbf{of} \\
 & \qquad\qquad\qquad Just\ m \quad \to add\ c\ (VAL\ m : VAL\ n : s) \\
 & \qquad\qquad\qquad Nothing \to fail\ s \\
 & \quad Nothing \to fail\ s
\end{array}
$$

Constructive Induction: *Add* (2)

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad\quad \mathit{Just}\ m \quad \to \mathit{add}\ c\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
 & \qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{fail}\ s \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{VAL}\ n : s) = \mathit{fail}\ s\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
 & \qquad\qquad\qquad\quad \mathit{Just}\ m \quad \to \mathit{add}\ c\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s) \\
 & \qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{fail}\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{induction hypothesis for } y\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad \to \mathit{eval}_C\ y\ (\mathit{add}\ c)\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \to \mathit{fail}\ s \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{eval}_C\ x\ (\mathit{eval}_C\ y\ (\mathit{add}\ c))\ s
\end{array}
$$

Constructive Induction: *Catch*

▸ Skip this

$$
\begin{aligned}
& \mathit{eval}_C\ (\mathit{Catch}\ x\ h)\ c\ s \\
= \quad & \{\ \text{specification}\ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to c\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \to \textbf{case}\ \mathit{eval}\ h\ \textbf{of} \\
& \qquad\qquad\qquad\quad \mathit{Just}\ m \quad \to c\ (\mathit{VAL}\ m : s) \\
& \qquad\qquad\qquad\quad \mathit{Nothing} \to \mathit{fail}\ s \\
= \quad & \{\ \text{induction hypothesis for}\ h\ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Just}\ n \quad \to c\ (\mathit{VAL}\ n : s) \\
& \quad \mathit{Nothing} \to \mathit{eval}_C\ h\ c\ s
\end{aligned}
$$

Constructive Induction: *Catch* (2)

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad\ \rightarrow c\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \rightarrow \mathit{eval}_{\mathsf{C}}\ h\ c\ s \\
= & \quad \{\ \text{define: } \mathit{fail}\ (\mathit{HAN}\ c' : s) = c'\ s\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad\ \rightarrow c\ (\mathit{VAL}\ n : s) \\
 & \quad \mathit{Nothing} \rightarrow \mathit{fail}\ (\mathit{HAN}\ (\mathit{eval}_{\mathsf{C}}\ h\ c) : s) \\
= & \quad \{\ \text{define: } \mathit{unmark}\ c\ (\mathit{VAL}\ n : \mathit{HAN}\ \_ : s) = c\ (\mathit{VAL}\ n : s)\ \} \\
 & \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
 & \quad \mathit{Just}\ n \quad\ \rightarrow \mathit{unmark}\ c\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{eval}_{\mathsf{C}}\ h\ c) : s) \\
 & \quad \mathit{Nothing} \rightarrow \mathit{fail}\ (\mathit{HAN}\ (\mathit{eval}_{\mathsf{C}}\ h\ c) : s) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{eval}_{\mathsf{C}}\ x\ (\mathit{unmark}\ c)\ (\mathit{HAN}\ (\mathit{eval}_{\mathsf{C}}\ h\ c) : s) \\
= & \quad \{\ \text{define: } \mathit{mark}\ c'\ c\ s = c\ (\mathit{HAN}\ c' : s)\ \} \\
 & \mathit{mark}\ (\mathit{eval}_{\mathsf{C}}\ h\ c)\ (\mathit{eval}_{\mathsf{C}}\ x\ (\mathit{unmark}\ c))\ s
\end{array}
$$

Resulting Compiler

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

Resulting Virtual Machine

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

Resulting Virtual Machine

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

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

Key Techniques

- transformation into CPS semantics
- defunctionalisation of CPS semantics

Key Techniques

$$\left.\begin{array}{l}\bullet\ \text{transformation into CPS semantics}\\[1em] \bullet\ \text{defunctionalisation of CPS semantics}\end{array}\right\}\ \text{foundation}$$

Key Techniques

- transformation into CPS semantics
- defunctionalisation of CPS semantics

$\}$ foundation

- partial specifications
- fixpoint induction
- defunctionalisation of semantics

Key Techniques

- transformation into CPS semantics
- defunctionalisation of CPS semantics

} foundation

- partial specifications ⇐ reduce required prior knowledge
- fixpoint induction
- defunctionalisation of semantics

Key Techniques

- transformation into CPS semantics
- defunctionalisation of CPS semantics

$\Big\}$ foundation

- partial specifications ⇐ reduce required prior knowledge
- fixpoint induction ⇐ for recursion and loops
- defunctionalisation of semantics

Key Techniques

- transformation into CPS semantics
- defunctionalisation of CPS semantics

} foundation

- partial specifications ⇐ reduce required prior knowledge
- fixpoint induction ⇐ for recursion and loops
- defunctionalisation of semantics ⇐ for lambda calculi

Summary

- simple, goal-oriented calculations; no magic*
- little prior knowledge needed
  (by using partial specifications)
- full correctness proof
- 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
- 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

Summary

- simple, goal-oriented calculations; no magic*
- little prior knowledge needed
  (by using partial specifications)
- full correctness proof
- 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
- formalisation in Coq

Ongoing and Future Work

- Simplify reasoning for “cyclic” features (recursion, loops)
- Simplify reasoning for register machines
- Support for sharing (i.e. graph structures)
- Abstraction over effects
- Derivation of compilers for fixed instruction sets