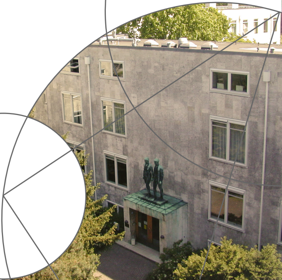
Faculty of Science

Reasoning over compilers using structured graphs

Patrick Bahr
University of Copenhagen,
Department of Computer Science
paba@diku.dk



2nd December, 2013

Goals and Motivation

Goal

Simplify implementation of and reasoning over compilers.

Goals and Motivation

Goal

Simplify implementation of and reasoning over compilers.

Method: Calculating Compilers

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

Goals and Motivation

Goal

Simplify implementation of and reasoning over compilers.

Method: Calculating Compilers

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

Goals and Motivation

Goal

Simplify implementation of and reasoning over compilers.

Method: Calculating Compilers

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

Problem: Representing Branching Control Flow

- tree-structured code vs. explicit labels and jumps

Goals and Motivation

Goal

Simplify implementation of and reasoning over compilers.

Method: Calculating Compilers

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

Problem: Representing Branching Control Flow

- tree-structured code vs. explicit labels and jumps
- Our proposal: use structured graphs (Oliveira & Cook, 2012)
- purely functional representation using variable binders

Outline

1. Calculating Compilers

2. Structured Graphs

Outline

Calculating Correct Compilers

(Bahr & Hutton, 2013)

History

- Underlying techniques: continuation-passing style & defunctionalisation (Reynolds, 1972)
- Origins: Wand (1982); Meijer (1992); Ager et al. (2003)

Calculating Correct Compilers

(Bahr & Hutton, 2013)

History

- Underlying techniques: continuation-passing style & defunctionalisation (Reynolds, 1972)
- Origins: Wand (1982); Meijer (1992); Ager et al. (2003)

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

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

Step 2: Transformation into CPS

Type Definitions

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

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

Step 2: Transformation into CPS

Type Definitions

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

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

Specification

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

Step 2: Transformation into CPS

Type Definitions

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

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

Specification

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

Constructive induction: "prove" specification by induction

Step 2: Transformation into CPS

Type Definitions

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

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

Specification

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

Constructive induction: “prove” specification by induction

$\leadsto$ definition of eval_C

Step 2: Transformation into CPS (cont.)

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

Step 2: Transformation into CPS (cont.)

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

Step 2: Transformation into CPS (cont.)

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

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

Step 2: Transformation into CPS (cont.)

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

Step 2: Transformation into CPS (cont.)

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

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

Step 2: Transformation into CPS (cont.)

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

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

Step 2: Transformation into CPS (cont.)

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

Step 2: Transformation into CPS (cont.)

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

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

Step 2: Transformation into CPS (cont.)

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

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

Step 2: Transformation into CPS (cont.)

Derived definition

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

Step 2: Transformation into CPS (cont.)

Derived definition

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

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

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

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

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

Step 3: Defunctionalisation

Compiler

Defunctionalisation of $eval_S$ and $eval_C$:

$$
\begin{aligned}
&comp :: Expr \to Code\\
&comp\ e = comp^A\ e\ HALT
\end{aligned}
$$

$$
\begin{aligned}
&comp^A :: Expr \to Code \to\ Code\\
&comp^A\ (Val\ n)\qquad c = PUSH\ n\ c\\
&comp^A\ (Add\ x\ y)\ c = comp^A\ x\ (comp^A\ y\ (ADD\ c))
\end{aligned}
$$

Step 3: Defunctionalisation

Compiler

Defunctionalisation of $eval_S$ and ev

$$comp :: Expr \to Code$$
$$comp\ e = comp^A\ e\ HALT$$

$$comp^A :: Expr \to Code \to Code$$
$$comp^A\ (Val\ n)\quad c = PUSH\ n\ c$$
$$comp^A\ (Add\ x\ y)\ c = comp^A\ x\ (comp^A\ y\ (ADD\ c))$$

data *Code* **where**

$HALT :: Code$

$PUSH :: Int \to Code \to Code$

$ADD\ :: Code \to Code$

Step 3: Defunctionalisation

Compiler

Defunctionalisation of $eval_S$ and ev

$$
\begin{aligned}
&comp :: Expr \to Code\\
&comp\ e = comp^A\ e\ HALT
\end{aligned}
$$

$$
\begin{aligned}
&comp^A :: Expr \to Code \to Code\\
&comp^A\ (Val\ n)\quad c = PUSH\ n\ c\\
&comp^A\ (Add\ x\ y)\ c = comp^A\ x\ (comp^A\ y\ (ADD\ c))
\end{aligned}
$$

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

Virtual Machine

$$
\begin{aligned}
&exec :: Code \to Cont\\
&exec\ HALT \qquad\quad = halt\\
&exec\ (PUSH\ n\ c) = push\ n\ (exec\ c)\\
&exec\ (ADD\ c) \qquad = add\ (exec\ c)
\end{aligned}
$$

Compiler Correctness

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

Compiler Correctness

$$
\begin{array}{llr}
 & \mathit{eval}_{\mathsf{C}}\ e\ c\ s = c\ (\mathit{eval}\ e : s) & \text{(Specification)} \\
+ & \mathit{exec}\ (\mathit{comp}\ e)\ s = \mathit{eval}_{\mathsf{S}}\ e\ s & \text{(Defuntionalisation)}
\end{array}
$$

Compiler Correctness

$$
\begin{array}{rrll}
 & \mathit{eval}_{\mathsf{C}}\ e\ c\ s = c\ (\mathit{eval}\ e : s) & & \text{(Specification)} \\
+ & \mathit{exec}\ (\mathit{comp}\ e)\ s = \mathit{eval}_{\mathsf{S}}\ e\ s & & \text{(Defuntionalisation)} \\
+ & \mathit{eval}_{\mathsf{S}}\ e = \mathit{eval}_{\mathsf{C}}\ e\ \mathit{halt} & & \text{(Definition of } \mathit{eval}_{\mathsf{S}}\text{)}
\end{array}
$$

Compiler Correctness

$$
\begin{array}{rrl}
 & eval_{\mathsf{C}}\ e\ c\ s = c\ (eval\ e : s) & \text{(Specification)} \\
+ & exec\ (comp\ e)\ s = eval_{\mathsf{S}}\ e\ s & \text{(Defuntionalisation)} \\
+ & eval_{\mathsf{S}}\ e = eval_{\mathsf{C}}\ e\ halt & \text{(Definition of } eval_{\mathsf{S}}\text{)} \\
\hline\hline
= & exec\ (comp\ e)\ s = eval\ e : s & \text{(Compiler correctness)}
\end{array}
$$

A Language with Exceptions

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

A Language with Exceptions

```haskell
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
```

The Derived Compiler

$$\begin{aligned}\textbf{data}\ \mathit{Code} = {} & \mathit{PUSH\ Int\ Code} \mid \mathit{ADD\ Code} \mid \mathit{HALT} \\ \mid {} & \mathit{MARK\ Code\ Code} \mid \mathit{UNMARK\ Code} \mid \mathit{THROW}\end{aligned}$$

The Derived Compiler

$$\begin{array}{llll}\textbf{data}\ \mathit{Code} = & \mathit{PUSH\ Int\ Code} & \mid \mathit{ADD\ Code} & \mid \mathit{HALT} \\ \mid & \mathit{MARK\ Code\ Code} & \mid \mathit{UNMARK\ Code} & \mid \mathit{THROW}\end{array}$$

$$\begin{array}{l} \mathit{comp}^{\mathsf{A}} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code} \\ \mathit{comp}^{\mathsf{A}}\ (\mathit{Val}\ n)\quad c = \mathit{PUSH}\ n\ c \\ \mathit{comp}^{\mathsf{A}}\ (\mathit{Add}\ x\ y)\quad c = \mathit{comp}^{\mathsf{A}}\ x\ (\mathit{comp}^{\mathsf{A}}\ y\ (\mathit{ADD}\ c)) \\ \mathit{comp}^{\mathsf{A}}\ \mathit{Throw}\quad c = \mathit{THROW} \\ \mathit{comp}^{\mathsf{A}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{MARK}\ (\mathit{comp}^{\mathsf{A}}\ h\ c)\ (\mathit{comp}^{\mathsf{A}}\ x\ (\mathit{UNMARK}\ c)) \end{array}$$

$$\begin{array}{l} \mathit{comp} :: \mathit{Expr} \to \mathit{Code} \\ \mathit{comp}\ e = \mathit{comp}^{\mathsf{A}}\ e\ \mathit{HALT} \end{array}$$

The Derived Compiler

$$
\begin{aligned}
&\textbf{data}\ \mathit{Code} = \mathit{PUSH}\ \mathit{Int}\ \mathit{Code} \mid \mathit{ADD}\ \mathit{Code} \mid \mathit{HALT}\\
&\qquad\qquad \mid \mathit{MARK}\ \mathit{Code}\ \mathit{Code} \mid \mathit{UNMARK}\ \mathit{Code} \mid \mathit{THROW}
\end{aligned}
$$

$$
\begin{aligned}
&comp^{\mathsf{A}} :: \mathit{Expr} \to \mathit{Code} \to \mathit{Code}\\
&comp^{\mathsf{A}}\ (\mathit{Val}\ n) \quad\ \ c = \mathit{PUSH}\ n \triangleright c\\
&comp^{\mathsf{A}}\ (\mathit{Add}\ x\ y) \ \ c = comp^{\mathsf{A}}\ x \triangleright comp^{\mathsf{A}}\ y \triangleright \mathit{ADD} \triangleright c\\
&comp^{\mathsf{A}}\ \mathit{Throw} \quad\ \ c = \mathit{THROW}\\
&comp^{\mathsf{A}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{MARK}\ (comp^{\mathsf{A}}\ h \triangleright c) \triangleright comp^{\mathsf{A}}\ x \triangleright \mathit{UNMARK} \triangleright c
\end{aligned}
$$

$$
\begin{aligned}
&comp :: \mathit{Expr} \to \mathit{Code}\\
&comp\ e = comp^{\mathsf{A}}\ e \triangleright \mathit{HALT}
\end{aligned}
$$

Outline

1. Calculating Compilers

2. Structured Graphs

Structured Graphs (Oliveira & Cook, 2012)

Structured Graphs

- Trees with explicit let bindings
- (parametric) higher-order abstract syntax

Structured Graphs (Oliveira & Cook, 2012)

Structured Graphs

- Trees with explicit let bindings
- (parametric) higher-order abstract syntax

Example

$$
\begin{aligned}
&\mathit{comp}^{\mathsf{A}}\ (\mathit{Val}\ n) && c = \mathit{PUSH}\ n \triangleright c \\
&\mathit{comp}^{\mathsf{A}}\ (\mathit{Add}\ x\ y) && c = \mathit{comp}^{\mathsf{A}}\ x \triangleright \mathit{comp}^{\mathsf{A}}\ y \triangleright \mathit{ADD} \triangleright c \\
&\mathit{comp}^{\mathsf{A}}\ \mathit{Throw} && c = \mathit{THROW} \\
&\mathit{comp}^{\mathsf{A}}\ (\mathit{Catch}\ x\ h) && c = \mathit{MARK}\ (\mathit{comp}^{\mathsf{A}}\ h \triangleright c) \\
& && \qquad \triangleright \mathit{comp}^{\mathsf{A}}\ x \triangleright \mathit{UNMARK} \triangleright c
\end{aligned}
$$

Structured Graphs (Oliveira & Cook, 2012)

Structured Graphs

- Trees with explicit let bindings
- (parametric) higher-order abstract syntax

Example

$$
\begin{aligned}
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Val}\ n) && c = \mathit{PUSH}_{\mathsf{G}}\ n \triangleright c \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Add}\ x\ y) && c = \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ y \triangleright \mathit{ADD}_{\mathsf{G}} \triangleright c \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ \mathit{Throw} && c = \mathit{THROW}_{\mathsf{G}} \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Catch}\ x\ h) && c = \mathit{Let}\ c\ (\lambda c' \rightarrow \mathit{MARK}_{\mathsf{G}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright \mathit{Var}\ c') \\
& && \qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ c')
\end{aligned}
$$

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

$$\textbf{data}\ \mathit{Tree}\ f = \mathit{In}\ (f\ (\mathit{Tree}\ f))$$

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

$$\textbf{data}\ \mathit{Tree}\ f = \mathit{In}\ (f\ (\mathit{Tree}\ f))$$

$$\begin{aligned}\textbf{data}\ \mathit{Code}\ a = {} & \mathit{PUSH}\ \mathit{Int}\ a \mid \mathit{ADD}\ a \quad\ \mid \mathit{HALT} \\ \mid {} & \mathit{MARK}\ a\ a \ \mid \mathit{UNMARK}\ a \mid \mathit{THROW}\end{aligned}$$

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

$$\textbf{data}\ \mathit{Tree}\ f = \mathit{In}\ (f\ (\mathit{Tree}\ f))$$

$$\begin{aligned}\textbf{data}\ \mathit{Code}\ a = {} & \mathit{PUSH}\ \mathit{Int}\ a \mid \mathit{ADD}\ a \mid \mathit{HALT} \\ \mid {} & \mathit{MARK}\ a\ a \mid \mathit{UNMARK}\ a \mid \mathit{THROW}\end{aligned}$$

$$\begin{aligned}
&\mathit{comp}^{\mathsf{A}}_{\mathsf{T}} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Code} \to \mathit{Tree}\ \mathit{Code} \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Val}\ n)\quad c = \mathit{PUSH}_{\mathsf{T}}\ n \triangleright c \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ y \triangleright \mathit{ADD}_{\mathsf{T}} \triangleright c \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ \mathit{Throw}\quad c = \mathit{THROW}_{\mathsf{T}} \\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{MARK}_{\mathsf{T}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \triangleright c) \\
&\qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{UNMARK}_{\mathsf{T}} \triangleright c
\end{aligned}$$

$$\begin{aligned}
&\mathit{comp}_{\mathsf{T}} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Code} \\
&\mathit{comp}_{\mathsf{T}}\ e = \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ e \triangleright \mathit{HALT}_{\mathsf{T}}
\end{aligned}$$

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

$$\textbf{data}\ \mathit{Tree}\ f = \mathit{In}\ (f\ (\mathit{Tree}\ f))$$

$$\begin{aligned}
\textbf{data}\ \mathit{Code}\ a = {} & \mathit{PUSH}\ \mathit{Int}\ a \mid \mathit{ADD}\ a \mid \mathit{HALT} \\
\mid {} & \mathit{MARK}\ a\ a \mid \mathit{UNMARK}\ a \mid \mathit{THROW}
\end{aligned}$$

$$\mathit{PUSH}_{\mathsf{T}}\ n\ c = \mathit{In}\ (\mathit{PUSH}\ n\ c)$$

$$\begin{aligned}
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Code} \to \mathit{Tree}\ \mathit{Code} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Val}\ n)\ c = \mathit{PUSH}_{\mathsf{T}}\ n \triangleright c \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Add}\ x\ y)\ c = \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ y \triangleright \mathit{ADD}_{\mathsf{T}} \triangleright c \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ \mathit{Throw}\ c = \mathit{THROW}_{\mathsf{T}} \\
& \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{MARK}_{\mathsf{T}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ h \triangleright c) \\
& \qquad \triangleright\ \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright \mathit{UNMARK}_{\mathsf{T}} \triangleright c
\end{aligned}$$

$$\begin{aligned}
& \mathit{comp}_{\mathsf{T}} :: \mathit{Expr} \to \mathit{Tree}\ \mathit{Code} \\
& \mathit{comp}_{\mathsf{T}}\ e = \mathit{comp}^{\mathsf{A}}_{\mathsf{T}}\ e \triangleright \mathit{HALT}_{\mathsf{T}}
\end{aligned}$$

Structured Graphs

Definition

$$\begin{aligned}\textbf{data}\ \mathit{Graph}'\ f\ v = {} & \mathit{GIn}\ (f\ (\mathit{Graph}'\ f\ v))\\ \mid {} & \mathit{Let}\ (\mathit{Graph}'\ f\ v)\ (v \to \mathit{Graph}'\ f\ v)\\ \mid {} & \mathit{Var}\ v\end{aligned}$$

Structured Graphs

Definition

$$
\begin{aligned}
&\textbf{data}\ \mathit{Graph}'\ f\ v = \mathit{GIn}\ (f\ (\mathit{Graph}'\ f\ v))\\
&\qquad\qquad\qquad\quad \mid\ \mathit{Let}\ (\mathit{Graph}'\ f\ v)\ (v \to \mathit{Graph}'\ f\ v)\\
&\qquad\qquad\qquad\quad \mid\ \mathit{Var}\ v
\end{aligned}
$$

$$\textbf{newtype}\ \mathit{Graph}\ f = \mathit{Graph}\ (\forall\ v\ .\ \mathit{Graph}'\ f\ v)$$

Structured Graphs

Definition

$$
\begin{aligned}
&\textbf{data}\ \mathit{Graph}'\ f\ v = \mathit{GIn}\ (f\ (\mathit{Graph}'\ f\ v))\\
&\qquad\qquad\qquad\quad \mid\ \mathit{Let}\ (\mathit{Graph}'\ f\ v)\ (v \to \mathit{Graph}'\ f\ v)\\
&\qquad\qquad\qquad\quad \mid\ \mathit{Var}\ v\\
&\textbf{newtype}\ \mathit{Graph}\ f = \mathit{Graph}\ (\forall\ v\ .\ \mathit{Graph}'\ f\ v)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}} :: \mathit{Expr} \to \mathit{Graph}'\ \mathit{Code}\ v \to \mathit{Graph}'\ \mathit{Code}\ v\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Val}\ n)\qquad\ \ c = \mathit{PUSH}_{\mathsf{G}}\ n \triangleright c\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Add}\ x\ y)\quad c = \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ y \triangleright \mathit{ADD}_{\mathsf{G}} \triangleright c\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Throw})\qquad c = \mathit{THROW}_{\mathsf{G}}\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{Let}\ c\ (\lambda c' \to \mathit{MARK}_{\mathsf{G}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright \mathit{Var}\ c')\\
&\qquad\qquad\qquad\qquad\qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ c')
\end{aligned}
$$

Structured Graphs

Definition

$$
\begin{aligned}
&\textbf{data}\ \mathit{Graph}'\ f\ v = \mathit{GIn}\ (f\ (\mathit{Graph}'\ f\ v))\\
&\qquad\qquad\qquad\quad \mid\ \mathit{Let}\ (\mathit{Graph}'\ f\ v)\ (v \to \mathit{Graph}'\ f\ v)\\
&\qquad\qquad\qquad\quad \mid\ \mathit{Var}\ v
\end{aligned}
$$

$$\textbf{newtype}\ \mathit{Graph}\ f = \mathit{Graph}\ (\forall\ v\ .\ \mathit{Graph}'\ f\ v)$$

$$
\begin{aligned}
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}} :: \mathit{Expr} \to \mathit{Graph}'\ \mathit{Code}\ v \to \mathit{Graph}'\ \mathit{Code}\ v\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Val}\ n)\qquad c = \mathit{PUSH}_{\mathsf{G}}\ n \triangleright c\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Add}\ x\ y)\ \ c = \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ y \triangleright \mathit{ADD}_{\mathsf{G}} \triangleright c\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Throw})\qquad c = \mathit{THROW}_{\mathsf{G}}\\
&\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{Let}\ c\ (\lambda c' \to \mathit{MARK}_{\mathsf{G}}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright \mathit{Var}\ c')\\
&\qquad\qquad\qquad\qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright \mathit{UNMARK}_{\mathsf{G}} \triangleright \mathit{Var}\ c')\\
&\mathit{comp}_{\mathsf{G}} :: \mathit{Expr} \to \mathit{Graph}\ \mathit{Code}\\
&\mathit{comp}_{\mathsf{G}}\ e = \mathit{Graph}\ (\mathit{comp}^{\mathsf{A}}_{\mathsf{G}}\ e \triangleright \mathit{HALT}_{\mathsf{G}})
\end{aligned}
$$

Virtual Machine as a Fold

Fold over Trees

$$\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (f\ r \to r) \to \mathit{Tree}\ f \to r$$
$$\mathit{fold}\ \mathit{alg}\ (\mathit{In}\ t) = \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{alg})\ t)$$

Virtual Machine as a Fold

Fold over Trees

$$fold :: Functor\ f \Rightarrow (f\ r \rightarrow r) \rightarrow Tree\ f \rightarrow r$$
$$fold\ alg\ (In\ t) = alg\ (fmap\ (fold\ alg)\ t)$$

Virtual Machine as a Fold

$$exec_{\mathsf{T}} :: Tree\ Code \rightarrow Stack \rightarrow Stack$$
$$exec_{\mathsf{T}} = fold\ execAlg$$

Virtual Machine as a Fold

Fold over Trees

$$
\begin{aligned}
&\mathit{fold} :: \mathit{Functor}\ f \Rightarrow (f\ r \to r) \to \mathit{Tree}\ f \to r\\
&\mathit{fold}\ \mathit{alg}\ (\mathit{In}\ t) = \mathit{alg}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{alg})\ t)
\end{aligned}
$$

Virtual Machine as a Fold

$$
\begin{aligned}
&\mathit{exec}_{\mathsf{T}} :: \mathit{Tree}\ \mathit{Code} \to \mathit{Stack} \to \mathit{Stack}\\
&\mathit{exec}_{\mathsf{T}} = \mathit{fold}\ \mathit{execAlg}
\end{aligned}
$$

Folds on Graphs

$$
\begin{aligned}
&\mathit{ufold} :: \mathit{Functor}\ f \Rightarrow (f\ r \to r) \to \mathit{Graph}\ f \to r\\
&\mathit{ufold}\ \mathit{alg}\ (\mathit{Graph}\ g) = \mathit{ufold}'\ g\ \mathbf{where}\\
&\quad \mathit{ufold}'\ (\mathit{GIn}\ t) \quad = \mathit{alg}\ (\mathit{fmap}\ \mathit{ufold}'\ t)\\
&\quad \mathit{ufold}'\ (\mathit{Let}\ e\ f) = \mathit{ufold}'\ (f\ (\mathit{ufold}'\ e))\\
&\quad \mathit{ufold}'\ (\mathit{Var}\ x) \quad = x
\end{aligned}
$$

Virtual Machine as a Fold

Fold over Trees

$$fold :: Functor\ f \Rightarrow (f\ r \to r) \to Tree\ f \to r$$
$$fold\ alg\ (In\ t) = alg\ (fmap\ (fold\ alg)\ t)$$

Virtual Machine as a Fold

$$exec_{\mathsf{G}} :: Graph\ Code \to Stack \to Stack$$
$$exec_{\mathsf{G}} = ufold\ execAlg$$

Folds on Graphs

$$
\begin{aligned}
&ufold :: Functor\ f \Rightarrow (f\ r \to r) \to Graph\ f \to r\\
&ufold\ alg\ (Graph\ g) = ufold'\ g\ \mathbf{where}\\
&\quad ufold'\ (GIn\ t) \quad\ = alg\ (fmap\ ufold'\ t)\\
&\quad ufold'\ (Let\ e\ f) = ufold'\ (f\ (ufold'\ e))\\
&\quad ufold'\ (Var\ x) \quad = x
\end{aligned}
$$

Correctness proof

Correctness of tree-based compiler (from calculation)

$$\mathit{exec}_{\mathsf{T}}\ (\mathit{comp}_{\mathsf{T}}\ e)\ [\,] = \mathit{conv}\ (\mathit{eval}\ e)$$

Correctness proof

Correctness of tree-based compiler (from calculation)

$$exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ [\,] = conv\ (eval\ e)$$

It suffices to show that

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s = exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ s$$

Correctness proof

Correctness of tree-based compiler (from calculation)

$$exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ [\,] = conv\ (eval\ e)$$

It suffices to show that

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s = exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ s$$

Proof.

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s \overset{(1)}{=} exec_{\mathsf{T}}\ (unravel\ (comp_{\mathsf{G}}\ e))\ s$$

Correctness proof

Correctness of tree-based compiler (from calculation)

$$exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ [\,] = conv\ (eval\ e)$$

It suffices to show that

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s = exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ s$$

Proof.

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s \overset{(1)}{=} exec_{\mathsf{T}}\ (unravel\ (comp_{\mathsf{G}}\ e))\ s$$

Theorem

$$ufold\ alg = fold\ alg \circ unravel$$

Correctness proof

Correctness of tree-based compiler (from calculation)

$$exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ [\,] = conv\ (eval\ e)$$

It suffices to show that

$$exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s = exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ s$$

Proof.

$$\begin{aligned} exec_{\mathsf{G}}\ (comp_{\mathsf{G}}\ e)\ s &\overset{(1)}{=} exec_{\mathsf{T}}\ (unravel\ (comp_{\mathsf{G}}\ e))\ s \\ &\overset{(2)}{=} exec_{\mathsf{T}}\ (comp_{\mathsf{T}}\ e)\ s \end{aligned}$$

Theorem

$$ufold\ alg = fold\ alg \circ unravel$$

Proof of (2)

Lemma

$$\mathit{unravel}\ (\mathit{comp}_{\mathsf{G}}\ e) = \mathit{comp}_{\mathsf{T}}\ e$$

Proof of (2)

Lemma

$$\mathit{unravel}\ (\mathit{comp}_{\mathsf{G}}\ e) = \mathit{comp}_{\mathsf{T}}\ e$$

Proof.

By induction on e.

Proof of (2)

Lemma

$$unravel\ (comp_{\mathsf{G}}\ e) = comp_{\mathsf{T}}\ e$$

Proof.

By induction on e.

The interesting part:

$$\begin{aligned}
&unravel\ (Let\ c\ (\lambda c' \rightarrow \\
&\quad MARK_{\mathsf{G}}\ (comp^{\mathsf{A}}_{\mathsf{G}}\ h \triangleright Var\ c') \\
&\qquad \triangleright comp^{\mathsf{A}}_{\mathsf{G}}\ x \triangleright UNMARK_{\mathsf{G}} \triangleright Var\ c')) \\
= {} & MARK_{\mathsf{T}}\ (comp^{\mathsf{A}}_{\mathsf{T}}\ h \triangleright unravel\ c) \\
&\qquad \triangleright comp^{\mathsf{A}}_{\mathsf{T}}\ x \triangleright UNMARK_{\mathsf{T}} \triangleright unravel\ c
\end{aligned}$$

$\square$

But ...

Things are not as nice as they seem on the outside

- HOAS is a nice interface to construct graphs
- But: HOAS is difficult to reason over

But ...

Things are not as nice as they seem on the outside

- HOAS is a nice interface to construct graphs
- But: HOAS is difficult to reason over

Alternative: “Names for free” (Bernardy & Pouillard, 2013)

- provides the same HOAS interface
- But: it’s de Bruin indices under the hood

Summary

Calculating Correct Compilers

- 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

Calculating Correct Compilers

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

Structured Graphs/Names for free

- improve calculated compilers
- avoid reasoning over explicit labels and jumps
- simple reasoning principle

Open Questions / Future Work

Beyond folds

- What if the virtual machine is not a fold?
- This seems impossible with HOAS-style graphs
- Ad hoc reasoning for "Names for free"-style graphs possible

Open Questions / Future Work

Beyond folds

- What if the virtual machine is not a fold?
- This seems impossible with HOAS-style graphs
- Ad hoc reasoning for "Names for free"-style graphs possible

Cyclic graphs

- Our method is restricted to acyclic graphs.
- Cyclic graphs require different reasoning principle.
  (fixed-point induction?)