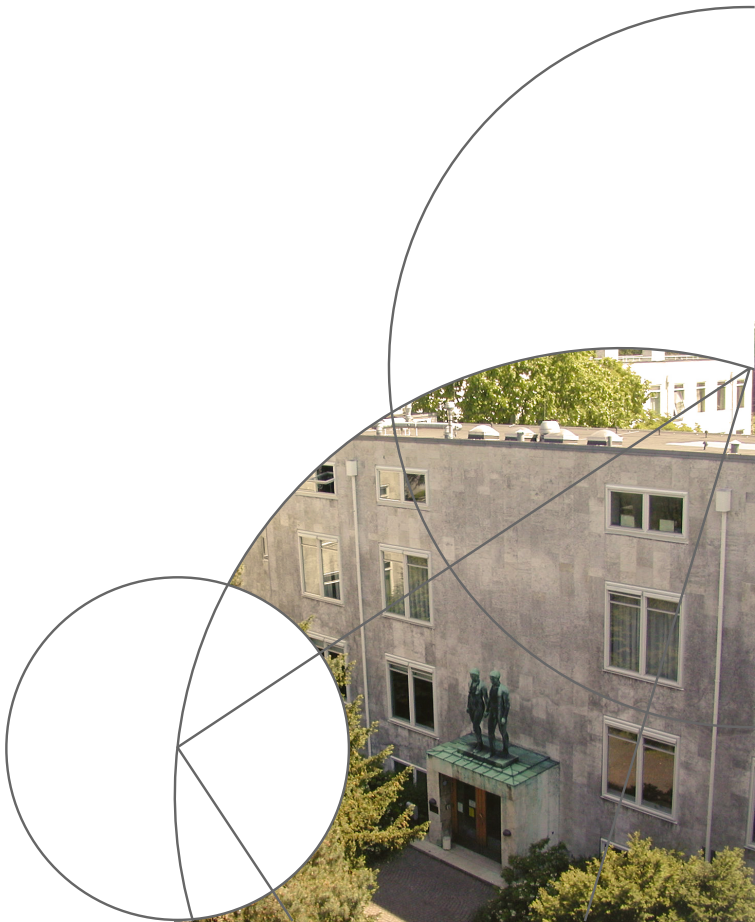
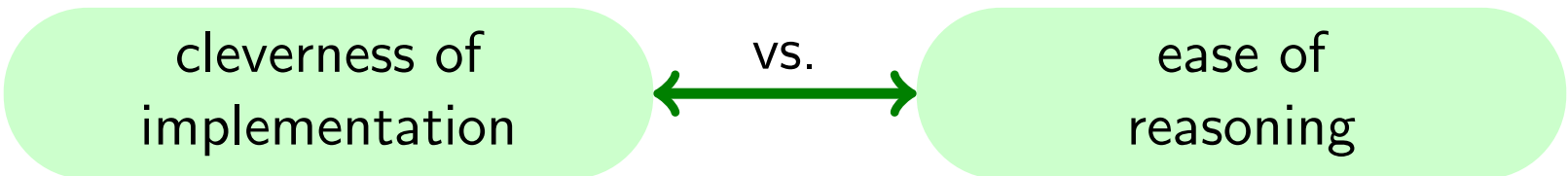
Proving Correctness of Compilers Using Structured Graphs

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



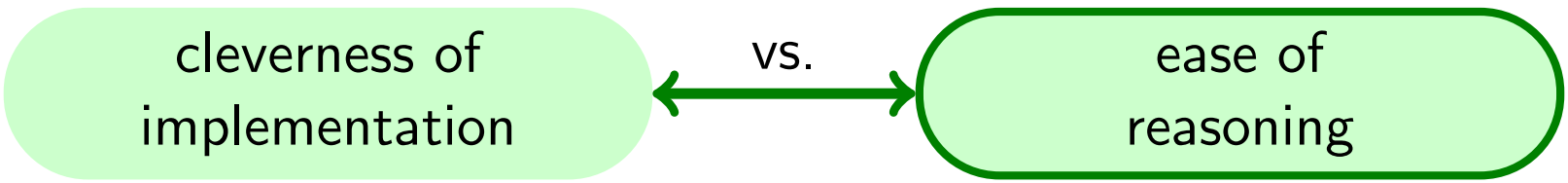
Introduction

Trade-off in software verification:



Introduction

Trade-off in software verification:



Trade-off in Compiler Verification

Example: Hutton & Wright "Compiling Exceptions Correctly"

Two compilers for a simple language with exceptions:

Trade-off in Compiler Verification

Example: Hutton & Wright "Compiling Exceptions Correctly"

Two compilers for a simple language with exceptions:

- Simple but unrealistic compiler (tree shaped code!)
  $\rightsquigarrow$ simple proofs

Trade-off in Compiler Verification

Example: Hutton & Wright "Compiling Exceptions Correctly"

Two compilers for a simple language with exceptions:

- Simple but unrealistic compiler (tree shaped code!)
  $\rightsquigarrow$ simple proofs
- More realistic compiler with explicit jumps
  $\rightsquigarrow$ much more complicated proofs

Trade-off in Compiler Verification

Example: Hutton & Wright "Compiling Exceptions Correctly"

Two compilers for a simple language with exceptions:

- Simple but unrealistic compiler (tree shaped code!)
  $\rightsquigarrow$ simple proofs
- More realistic compiler with explicit jumps
  $\rightsquigarrow$ much more complicated proofs

Our Proposal: an intermediate approach

- Transform compiler: use (acyclic) graphs instead of trees
- Lift the correctness property from the tree-based to the graph-based compiler.

Example: A Simple Language with Exceptions

Based on Hutton & Wright "Compiling Exceptions Correctly"

Source Language

Arithmetic expressions + exceptions:

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

Example: A Simple Language with Exceptions

Based on Hutton & Wright "Compiling Exceptions Correctly"

Source Language

Arithmetic expressions + exceptions:

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

Target Language

Instruction set for a simple stack machine:

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

A Simple Compiler

Targeting A Stack Machine

$$comp^{A} :: Expr \to Code \to Code$$

A Simple Compiler

Targeting A Stack Machine

$$comp^{A} :: Expr \to Code \to Code$$

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

A Simple Compiler

Targeting A Stack Machine

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

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

A Simple Compiler

Targeting A Stack Machine

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

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

Semantics & Correctness

Semantics

Given by evaluator *eval* & virtual machine *exec*

$$eval :: Expr \rightarrow Maybe\ \ Int$$

$$exec :: Code \rightarrow Stack \rightarrow Stack$$

Semantics & Correctness

Semantics

Given by evaluator *eval* & virtual machine *exec*

$$eval :: Expr \rightarrow Maybe\ Int$$

$$exec :: Code \rightarrow Stack \rightarrow Stack$$

data *Maybe a* = *Just a*
| *Nothing*

Semantics & Correctness

Semantics

Given by evaluator *eval* & virtual machine *exec*

$$eval :: Expr \rightarrow Maybe\ \ Int$$

$$exec :: Code \rightarrow Stack \rightarrow Stack$$

Semantics & Correctness

Semantics

Given by evaluator *eval* & virtual machine *exec*

$$eval :: Expr \rightarrow Maybe\ Int$$

$$exec :: Code \rightarrow Stack \rightarrow Stack$$

Theorem (compiler correctness)

$$exec\ (comp\ e)\ [\,] = \begin{cases} [Val\ n] & \text{if } eval\ e = Just\ n \\ [\,] & \text{if } eval\ e = Nothing \end{cases}$$

Semantics & Correctness

Semantics

Given by evaluator *eval* & virtual machine *exec*

$$eval :: Expr \to Maybe\ Int$$

$$exec :: Code \to Stack \to Stack$$

Theorem (compiler correctness)

$$exec\ (comp\ e)\ [\,] = \begin{cases} [Val\ n] & \text{if } eval\ e = Just\ n \\ [\,] & \text{if } eval\ e = Nothing \end{cases}$$

Goal

- Avoid the code duplication produced by the compiler.
- Retain the simple equational reasoning to prove correctness.

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)
2. The VM is a fold, i.e.
   $exec = fold\ execAlg$

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)
2. The VM is a fold, i.e.
   $exec = fold\ execAlg$
3. On graphs, the VM is defined as a fold with the same algebra:
   $exec_G = fold_G\ execAlg$

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)
2. The VM is a fold, i.e.
   $exec = fold\ execAlg$
3. On graphs, the VM is defined as a fold with the same algebra:
   $exec_G = fold_G\ execAlg$
4. By parametricity, we obtain:

$$fold_G\ alg = fold\ alg \circ unravel \qquad \text{for all } alg$$

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)
2. The VM is a fold, i.e.
   $exec = fold\ execAlg$
3. On graphs, the VM is defined as a fold with the same algebra:
   $exec_G = fold_G\ execAlg$
4. By parametricity, we obtain:

$$exec_G = exec \circ unravel$$

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)
2. The VM is a fold, i.e.
   $exec = fold\ execAlg$
3. On graphs, the VM is defined as a fold with the same algebra:
   $exec_G = fold_G\ execAlg$
4. By parametricity, we obtain:
   $$exec_G = exec \circ unravel$$
5. By simple equational reasoning we show
   $$comp = unravel \circ comp_G$$

How Do We Achieve This?

1. trees $\Rightarrow$ structured graphs (trees + explicit let bindings)
2. The VM is a fold, i.e.
   $exec = fold\ execAlg$
3. On graphs, the VM is defined as a fold with the same algebra:
   $exec_G = fold_G\ execAlg$
4. By parametricity, we obtain:
   $$exec_G = exec \circ unravel$$
5. By simple equational reasoning we show
   $$comp = unravel \circ comp_G$$
6. Hence: $exec \circ comp = exec_G \circ comp_G$

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

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

Code data type

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

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

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

Code data type

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

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

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

Code data type

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

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

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

Code data type

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

$$\Rightarrow \quad \mathit{Code} \simeq \mathit{Tree}\ \mathit{Code}_{\mathsf{F}}$$

Explicit Representation of Tree Types

Tree Type: fixed point of a functor

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

Code data type

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

$$\Rightarrow \quad \mathit{Code} \simeq \mathit{Tree}\ \mathit{Code}_\mathsf{F}$$

Smart Constructors

$$\begin{aligned}&\mathit{PUSH}_\mathsf{T} :: \mathit{Int} \to \mathit{Tree}\ \mathit{Code}_\mathsf{F} \to \mathit{Tree}\ \mathit{Code}_\mathsf{F} \\ &\mathit{PUSH}_\mathsf{T}\ n\ c = \mathit{In}\ (\mathit{PUSH}_\mathsf{F}\ n\ c) \\ &\qquad\qquad \vdots\end{aligned}$$

Compiler with Explicit Tree Type

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

Compiler with Explicit Tree Type

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

Compiler with Explicit Tree Type

$$
\begin{aligned}
&comp^{\mathsf{A}} :: Expr \to Tree\ Code_{\mathsf{F}} \to Tree\ Code_{\mathsf{F}} \\
&comp^{\mathsf{A}}\ (Val\ n) \quad\quad c = PUSH_{\mathsf{T}}\ \ n \triangleright c \\
&comp^{\mathsf{A}}\ (Add\ x\ y) \quad c = comp^{\mathsf{A}}\ x \triangleright comp^{\mathsf{A}}\ y \triangleright ADD_{\mathsf{T}} \triangleright c \\
&comp^{\mathsf{A}}\ \ Throw \quad\quad c = THROW_{\mathsf{T}} \\
&comp^{\mathsf{A}}\ (Catch\ x\ h)\ c = MARK_{\mathsf{T}}\ (comp^{\mathsf{A}}\ h \triangleright c) \\
&\qquad\qquad\qquad\qquad \triangleright comp^{\mathsf{A}}\ x \triangleright UNMARK_{\mathsf{T}} \triangleright c \\
&comp :: Expr \to Tree\ Code \\
&comp\ e = comp^{\mathsf{A}}\ e \triangleright HALT_{\mathsf{T}}
\end{aligned}
$$

Structured Graphs (Oliveira & Cook, 2012)

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 (Oliveira & Cook, 2012)

Definition

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

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

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

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

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

$$\begin{aligned}&\mathit{comp}^{\mathsf{A}}\ :: \mathit{Expr} \to \mathit{Graph'}\ \mathit{Code}_{\mathsf{F}}\ v \to \mathit{Graph'}\ \mathit{Code}_{\mathsf{F}}\ v\\ &\mathit{comp}^{\mathsf{A}}\ (\mathit{Val}\ n)\quad\ \ c = \mathit{PUSH}_{\mathsf{T}}\quad n \triangleright c\\ &\mathit{comp}^{\mathsf{A}}\ (\mathit{Add}\ x\ y)\quad c = \mathit{comp}^{\mathsf{A}}\ x \triangleright \mathit{comp}^{\mathsf{A}}\ y \triangleright \mathit{ADD}_{\mathsf{T}} \triangleright c\\ &\mathit{comp}^{\mathsf{A}}\ \ \mathit{Throw}\quad\ \ c = \mathit{THROW}_{\mathsf{T}}\\ &\mathit{comp}^{\mathsf{A}}\ (\mathit{Catch}\ x\ h)\ c = \mathit{MARK}_{\mathsf{T}}\ (\mathit{comp}^{\mathsf{A}}\ h \triangleright c)\\ &\qquad\qquad\qquad\qquad\qquad \triangleright \mathit{comp}^{\mathsf{A}}\ x \triangleright \mathit{UNMARK}_{\mathsf{T}} \triangleright c\end{aligned}$$

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

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

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

$$\mathit{PUSH}_\mathsf{G}\ n\ c = \mathit{GIn}\ (\mathit{PUSH}_\mathsf{F}\ n\ c)$$

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

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

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

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

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

Structured Graphs (Oliveira & Cook, 2012)

Definition

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

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

Example

$$\mathit{comp}\ (\mathit{Add}\ (\mathit{Catch}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{MARK}_\mathsf{T}\ (\mathit{PUSH}_\mathsf{T}\ 2 \triangleright \mathit{PUSH}_\mathsf{T}\ 3 \triangleright \mathit{ADD}_\mathsf{T} \triangleright \mathit{HALT}_\mathsf{T})$$
$$\triangleright\ \mathit{PUSH}_\mathsf{T}\ 1 \triangleright \mathit{UNMARK}_\mathsf{T} \triangleright \mathit{PUSH}_\mathsf{T}\ 3 \triangleright \mathit{ADD}_\mathsf{T} \triangleright \mathit{HALT}_\mathsf{T}$$

$$\mathit{comp}_\mathsf{G}\ (\mathit{Add}\ (\mathit{Catch}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2))\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{MkGraph}\ (\mathit{Let}\ (\mathit{PUSH}_\mathsf{G}\ 3 \triangleright \mathit{ADD}_\mathsf{G} \triangleright \mathit{HALT}_\mathsf{G})\ (\lambda v \rightarrow$$
$$\mathit{MARK}_\mathsf{G}\ (\mathit{PUSH}_\mathsf{G}\ 2 \triangleright \mathit{Var}\ v)$$
$$\triangleright\ \mathit{PUSH}_\mathsf{G}\ 1 \triangleright \mathit{UNMARK}_\mathsf{G} \triangleright \mathit{Var}\ v))$$

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

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

$$exec\ :: Tree\ \ Code \to Stack \to Stack$$
$$exec\ = fold\ \ execAlg$$

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

$$exec\ :: Tree\ \ Code \to Stack \to Stack$$
$$exec\ = fold\ \ execAlg$$

Folds on Graphs

$$fold_G :: Functor\ f \Rightarrow (f\ r \to r) \to Graph\ f \to r$$
$$fold_G\ alg\ (Graph\ g) = fold'_G\ g\ \textbf{where}$$
$$fold'_G\ (GIn\ t) = alg\ (fmap\ fold'_G\ t)$$
$$fold'_G\ (Let\ e\ f) = fold'_G\ (f\ (fold'_G\ e))$$
$$fold'_G\ (Var\ x) = x$$

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

$$exec_G :: Graph\ Code \to Stack \to Stack$$
$$exec_G = fold_G\ execAlg$$

Folds on Graphs

$$fold_G :: Functor\ f \Rightarrow (f\ r \to r) \to Graph\ f \to r$$
$$fold_G\ alg\ (Graph\ g) = fold'_G\ g\ \textbf{where}$$
$$fold'_G\ (GIn\ t) = alg\ (fmap\ fold'_G\ t)$$
$$fold'_G\ (Let\ e\ f) = fold'_G\ (f\ (fold'_G\ e))$$
$$fold'_G\ (Var\ x) = x$$

Correctness Argument for $comp_G$

Since, we know that *comp* is correct, it suffices to show that

$$exec_G \circ comp_G = exec \circ comp$$

Correctness Argument for $comp_G$

Since, we know that *comp* is correct, it suffices to show that

$$exec_G \circ comp_G = exec \circ comp$$

Proof.

$$exec_G \circ comp_G \overset{(1)}{=} exec \circ unravel \circ comp_G$$

Correctness Argument for $comp_G$

Since, we know that *comp* is correct, it suffices to show that

$$exec_G \circ comp_G = exec \circ comp$$

Proof.

$$exec_G \circ comp_G \overset{(1)}{=} exec \circ unravel \circ comp_G$$

Theorem

$$fold_G\ alg = fold\ alg \circ unravel \tag{1}$$

Correctness Argument for $comp_G$

Since, we know that *comp* is correct, it suffices to show that

$$exec_G \circ comp_G = exec \circ comp$$

Proof.

$$exec_G \circ comp_G \overset{(1)}{=} exec \circ unravel \circ comp_G$$
$$\overset{(2)}{=} exec \circ comp$$

Theorem

$$fold_G\ alg = fold\ alg \circ unravel \quad (1)$$

Proof of (2)

Lemma

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

Proof of (2)

Lemma

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

Proof.

By induction on e.

Proof of (2)

Lemma

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

Proof.

By induction on e.

The interesting part:

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

□

Summary

Our Approach

- Replace tree type with graph type
- Relate semantics of graph-based compiler via unravelling
- Exploit parametricity to drastically simplify proof

Summary

Our Approach

- Replace tree type with graph type
- Relate semantics of graph-based compiler via unravelling
- Exploit parametricity to drastically simplify proof

Motivation: Derive Compiler from Specification

- Compilers can be derived by formal calculation
- The result is often unsatisfactory (e.g. code duplication)
- Goal: improve compilers by simple equational reasoning

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

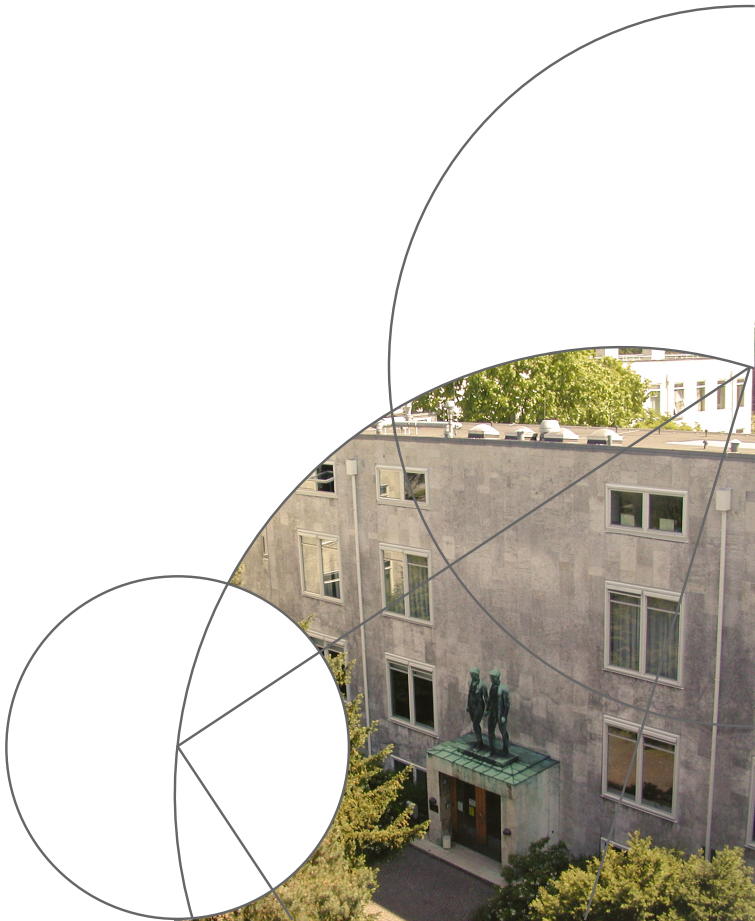
- What if the virtual machine is not a fold?
- This seems impossible with HOAS-style graphs
- Ad hoc reasoning for "Names for free"-style graphs possible

Cyclic graphs

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

Proving Correctness of Compilers Using Structured Graphs

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



Bonus Slides

Example

$$\mathit{comp}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{PUSH}\ 2 \triangleright \mathit{PUSH}\ 3 \triangleright \mathit{ADD} \triangleright \mathit{HALT}$$

Example

$$\mathit{comp}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{PUSH}\ 2 \triangleright \mathit{PUSH}\ 3 \triangleright \mathit{ADD} \triangleright \mathit{HALT}$$

$$\mathit{comp}\ (\mathit{Catch}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{MARK}\ (\mathit{PUSH}\ 3 \triangleright \mathit{HALT}) \triangleright \mathit{PUSH}\ 2 \triangleright \mathit{UNMARK} \triangleright \mathit{HALT}$$

Example

$$\mathit{comp}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{PUSH}\ 2 \triangleright \mathit{PUSH}\ 3 \triangleright \mathit{ADD} \triangleright \mathit{HALT}$$

$$\mathit{comp}\ (\mathit{Catch}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{MARK}\ (\mathit{PUSH}\ 3 \triangleright \mathit{HALT}) \triangleright \mathit{PUSH}\ 2 \triangleright \mathit{UNMARK} \triangleright \mathit{HALT}$$

$$\mathit{comp}\ (\mathit{Catch}\ \mathit{Throw}\ (\mathit{Val}\ 3))$$
$$\rightsquigarrow \mathit{MARK}\ (\mathit{PUSH}\ 3 \triangleright \mathit{HALT}) \triangleright \mathit{THROW}$$

Short Cut Fusion for Graphs

Theorem (Short Cut Fusion Law)

$$b\ alg = fold\ alg\ (b\ In) \quad for\ all\ b :: \forall\ c\ .\ (f\ c \to c) \to c$$

Short Cut Fusion for Graphs

Theorem (Short Cut Fusion Law)

$$b\ alg = fold\ alg\ (b\ In) \quad for\ all\ b :: \forall\ c\ .\ (f\ c \to c) \to c$$

- For any $g :: Graph\ f$, instantiate $b = \lambda a \to fold_{\mathsf{G}}\ a\ g$:

$$(\lambda a \to fold_{\mathsf{G}}\ a\ g)\ alg = fold\ alg\ ((\lambda a \to fold_{\mathsf{G}}\ a\ g)\ In)$$

Short Cut Fusion for Graphs

Theorem (Short Cut Fusion Law)

$$b\ alg = fold\ alg\ (b\ In) \quad for\ all\ b :: \forall\ c\ .\ (f\ c \to c) \to c$$

- For any $g :: Graph\ f$, instantiate $b = \lambda a \to fold_{\mathsf{G}}\ a\ g$:

$$(\lambda a \to fold_{\mathsf{G}}\ a\ g)\ alg = fold\ alg\ ((\lambda a \to fold_{\mathsf{G}}\ a\ g)\ In)$$

- After beta reduction:

$$fold_{\mathsf{G}}\ alg\ g = fold\ alg\ (fold_{\mathsf{G}}\ In\ g)$$

Short Cut Fusion for Graphs

Theorem (Short Cut Fusion Law)

$$b\ alg = fold\ alg\ (b\ In) \quad for\ all\ b :: \forall\ c\ .\ (f\ c \to c) \to c$$

- For any $g :: Graph\ f$, instantiate $b = \lambda a \to fold_G\ a\ g$:

$$(\lambda a \to fold_G\ a\ g)\ alg = fold\ alg\ ((\lambda a \to fold_G\ a\ g)\ In)$$

- After beta reduction:

$$fold_G\ alg\ g = fold\ alg\ (fold_G\ In\ g)$$

- By definition of *unravel*:

$$fold_G\ alg\ g = fold\ alg\ (unravel\ g)$$