

Verified Compilation to Intrinsically Typed Control-Flow Graphs in Agda



TECHNISCHE
UNIVERSITÄT
DARMSTADT

StoreI ($\Gamma : \text{Cxt}$) : $(\Phi \Phi' : ST) \rightarrow \text{Set where}$
 $\text{StoreI } \Gamma (t :: \Phi) \Phi$
 $\text{StoreI } \Gamma \Phi (t :: \Phi)$
 $\text{StoreI } \Gamma \text{ int} \rightarrow \text{StoreI }$
 $\forall t \{$
 $x : \text{Var } \Gamma t \} \rightarrow \text{StoreI } \Gamma (t :: \Phi) \Phi$
 $x : \text{Var } \Gamma t \} \rightarrow \text{StoreI } \Gamma \Phi (t :: \Phi)$
 $(b : \text{IncDec}) (x : \text{Var } \Gamma \text{ int}) \rightarrow \text{StoreI }$
Creating instructions
destroy local variables
Set where $([] :: \Gamma)$

Topic of Thesis

- Compiler verification
- Verification of one compiler pass
 - Source language: Imperative programming language
 - Target language: Control-flow graphs
- Started during semester abroad in 2019
at Chalmers University of Technology in Gothenburg
- Supervised by Andreas Abel



- dependently typed programming language
- implements extension of Martin-Löf type theory
- developed in the programming logic group at Chalmers

<https://wiki.portal.chalmers.se/agda/pmwiki.php>
<https://agda.readthedocs.io/>

Table of Contents

Preliminaries

While Language

Control Flow Graphs

Translation

Conclusion

Table of Contents

Preliminaries

While Language

Control Flow Graphs

Translation

Conclusion

Preliminaries

- Preliminaries:
 1. de Bruijn Indices
 2. Intrinsically Typed Syntax
- Both explained with the example of the simply typed lambda calculus

Simply Typed Lambda Calculus

Terms:

$e ::= x \mid \lambda x : t. e \mid e e \mid \dots$

- variables: x, y, z
- function abstractions:

$\lambda n : \mathbb{N}. n + n$
 $\lambda n : \mathbb{N}. n^2 + 3 * n$
 $\lambda f : \mathbb{N} \rightarrow \mathbb{N}. f 1$

Types:

$t ::= t \rightarrow t \mid \dots$

- function application:
 - $(\lambda n : \mathbb{N}. n + n) 5$
 - $(\lambda n : \mathbb{N}. n^2 + 3 * n) 4$
 - $(\lambda f : \mathbb{N} \rightarrow \mathbb{N}. f 1) (\lambda n : \mathbb{N}. n + n)$

Simply Typed Lambda Calculus

beta-Reduction:

$$(\lambda x : t. e) e' \longrightarrow e[x := e']$$

Example

- $(\lambda n : \mathbb{N}. n + n) 5 \longrightarrow 5 + 5$
- $(\lambda n : \mathbb{N}. n^2 + 3 * n) 4 \longrightarrow 4^2 + 3 * 4$
- $(\lambda f : \mathbb{N} \rightarrow \mathbb{N}. f 1) (\lambda n : \mathbb{N}. n + n) \longrightarrow (\lambda n : \mathbb{N}. n + n) 1 \longrightarrow 1 + 1$

De Bruijn Indices

- Use of natural numbers to represent variables
- $n \triangleq$ number of surrounding lambda terms (counting started by 0)

$$e ::= x \mid \lambda x : t. e \mid e e$$

becomes

$$e ::= n \mid \lambda t. e \mid e e$$

De Bruijn Indices - Example

$\lambda f : (t \rightarrow t). \lambda x : t. fx$

becomes:

$\lambda t \rightarrow t. \lambda t. 10$



Simply Typed Lambda Calculus in Agda

$e ::= n \mid \lambda t. e \mid ee$

```
data Term (n : ℕ) : Set where
  ind : Fin n → Term n
  abs : Type → Term (suc n) → Term n
  app : Term n → Term n → Term n
```

$n \triangleq$ number of free variables

$\text{Fin } n \triangleq \{0, 1, \dots, n - 1\}$

Example

$\lambda f : (t \rightarrow t). \lambda x : t. fx$

with de Bruijn indices:

$\lambda t \rightarrow t. \lambda t. 10$

$$\begin{array}{c} \lambda t \rightarrow t \\ | \\ \lambda t \\ | \\ \text{app} \\ 1 \swarrow 0 \end{array}$$

in Agda:

$$\underbrace{\text{abs}}_{\lambda} \underbrace{(\text{arrow } t \ t)}_{t \rightarrow t} \underbrace{(\underbrace{\text{abs}}_{\lambda} \underbrace{t}_{t})}_{t} \underbrace{(\text{app } (\underbrace{\text{ind } 1}_1) \underbrace{(\text{ind } 0}_0))}_{0}$$

Type System of Simply Typed Lambda Calculus

Types:

$$t ::= b \mid t \rightarrow t$$

Contexts:

$$\Gamma = x_1 : t_1, x_2 : t_2, \dots, x_n : t_n$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \text{ (var)}$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t. e : t \rightarrow t'} \text{ (abs)}$$

$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash e e' : t'} \text{ (app)}$$

Intrinsically Typed Syntax of the Simply Typed Lambda Calculus

```
data Type : Set where
  base : Type
  arrow : Type → Type → Type
```

Context = List Type

```
data Term (Γ : Context) : Type → Set where
  ind : ∀ {t} → t ∈ Γ → Term Γ t
  abs : ∀ t {t'} → Term (t :: Γ) t' → Term Γ (arrow tt')
  app : ∀ {t t'} → Term Γ (arrow tt') → Term Γ t → Term Γ t'
```

Term Γt : Agda type of lambda terms which have in context Γ the type t

Usual Abstract Syntax vs. Intrinsically Typed Syntax

```
data Term (n : ℕ) : Set where
  ind : Fin n → Term n
  abs : Type → Term (suc n) → Term n
  app : Term n → Term n → Term n
```

```
data Term (Γ : Context) : Type → Set where
  ind : ∀ {t} → t ∈ Γ → Term Γ t
  abs : ∀ t {t'} → Term (t :: Γ) t' → Term Γ (arrow tt')
  app : ∀ {t t'} → Term Γ (arrow tt') → Term Γ t → Term Γ t'
```

Term Γt : Agda type of lambda terms which have in context Γ the type t

De Bruijn Indices in Intrinsically Typed Syntax

```
data Term (n : ℕ) : Set where  
  ind : Fin n → Term n
```

```
data Term (Γ : Context) : Type → Set where  
  ind : ∀ {t} → t ∈ Γ → Term Γ t
```

- Parameter $(n : \mathbb{N})$ generalizes to $(\Gamma : \text{Context})$
- Constructor argument $\text{Fin } n$ generalizes to pair of type t and proof of $t \in \Gamma$

De Bruijn Indices in Intrinsically Typed Syntax

```
data Term (n : ℕ) : Set where  
  ind : Fin n → Term n
```

```
data Term (Γ : Context) : Type → Set where  
  ind : ∀ {t} → t ∈ Γ → Term Γ t
```

$t \in \Gamma$: Constructive proof that type t is contained in context Γ
 \Rightarrow contains information where t occurs in Γ

Example

$$\lambda f : (t \rightarrow t). \lambda x : t. fx$$

with de Bruijn indices:

$$\lambda t \rightarrow t. \lambda t. 10$$

abs (arrow t t) (abs t (app (ind 1) (ind 0)))

$$\begin{array}{c} \lambda t \rightarrow t \\ | \\ \lambda t \\ | \\ \text{app} \\ 1 \swarrow 0 \end{array}$$

in intrinsically typed syntax:

$$\underbrace{\text{abs}}_{\lambda} \underbrace{(\text{arrow} \ t \ t)}_{t \rightarrow t} \ (\underbrace{\text{abs}}_{\lambda} \underbrace{t}_{t} \ \underbrace{(\text{app} \ (\text{ind} \ (\text{there} \ (\text{here} \ \text{refl}))))}_{1} \ \underbrace{(\text{ind} \ (\text{here} \ \text{refl})))}_{0})$$

Table of Contents

Preliminaries

While Language

Control Flow Graphs

Translation

Conclusion

While Language

- Prototypical imperative programming language

Expressions

$$e ::= x \mid n \mid b \mid e + e \mid e * e \mid e = e \mid e \leq e$$

- Variables
- Number and boolean literals
- Arithmetic operations

Statements

$$S ::= x := e \mid \text{if } e \text{ then } S^* \text{ else } S^* \mid \text{while } e \text{ do } S^*$$

- Assignments
- Conditionals
- While Loops

Intrinsically Typed Syntax of Expressions and Statements

Types in While language:

```
data Type : Set where
  num bool : Type
```

Expr Γ t : Agda type of expressions which have in context Γ type t

Stmt Γ : Agda type of statements which are typeable in context Γ

Assignments: Type of assigned expression matches type of variable
Conditionals, while loops: Condition has type **bool**

Intrinsically Typed Syntax of Expressions and Statements

```
data Expr ( $\Gamma$  : Context) : Type → Set where
  var : {t : Type} → t ∈  $\Gamma$  → Expr  $\Gamma$  t
  lit : {t : Type} → Val t → Expr  $\Gamma$  t
  add : (e e' : Expr  $\Gamma$  num) → Expr  $\Gamma$  num
  mul : (e e' : Expr  $\Gamma$  num) → Expr  $\Gamma$  num
  eq : (e e' : Expr  $\Gamma$  num) → Expr  $\Gamma$  bool
  le : (e e' : Expr  $\Gamma$  num) → Expr  $\Gamma$  bool
```

```
data Stmt ( $\Gamma$  : Context) : Set where
  assign : {t : Type} → t ∈  $\Gamma$  → Expr  $\Gamma$  t → Stmt  $\Gamma$ 
  if      : (cond : Expr  $\Gamma$  bool) (then else : List (Stmt  $\Gamma$ )) → Stmt  $\Gamma$ 
  while   : (cond : Expr  $\Gamma$  bool) (body : List (Stmt  $\Gamma$ )) → Stmt  $\Gamma$ 
```

Summary: Expressions

- $x:$ $(\text{var } x)$
- $\text{val}:$ (lit val)
- $e + e':$ $(\text{add } e e')$
- $e * e':$ $(\text{mul } e e')$
- $e = e':$ $(\text{eq } e e')$
- $e \leq e':$ $(\text{le } e e')$

Summary: Statements

- $x := \text{expr}$: (**assign** x expr)
- $\text{if } cond \text{ then } then \text{ else else}$: (**if** $cond$ $\text{then } then$ else)
- $\text{while } cond \text{ do } body$: (**while** $cond$ $body$)

Semantics

- Semantics of statements given by big-step operational semantics
- Relates initial state to final state for statement S
- Notation: $\sigma \xrightarrow{S} \sigma'$

Example:

$$\text{Assign } \frac{e \Downarrow_{\sigma} \text{val}}{\sigma \xrightarrow{x := e} \sigma[x := \text{val}]}$$

$$\text{If-True } \frac{\begin{array}{c} b \Downarrow_{\sigma} \text{true} \quad \sigma \xrightarrow{S_1} \sigma' \\ \sigma \xrightarrow{\text{if } b \text{ then } S_1 \text{ else } S_2} \sigma' \end{array}}{\sigma \xrightarrow{\text{if } b \text{ then } S_1 \text{ else } S_2} \sigma'}$$

Semantics

Remaining Rules

$$\text{If-False} \frac{b \Downarrow_{\sigma} \text{false} \quad \sigma \xrightarrow{S_2} \sigma'}{\sigma \xrightarrow{\text{if } b \text{ then } S_1 \text{ else } S_2} \sigma'}$$

$$\text{While-True} \frac{b \Downarrow_{\sigma} \text{true} \quad \sigma \xrightarrow{\text{body}} \sigma' \quad \sigma' \xrightarrow{\text{while } b \text{ do body}} \sigma''}{\sigma \xrightarrow{\text{while } b \text{ do body}} \sigma''}$$

$$\text{While-False} \frac{b \Downarrow_{\sigma} \text{false}}{\sigma \xrightarrow{\text{while } b \text{ do body}} \sigma}$$

Agda Formalization of Semantics

`EvalStmt` σ `stmt` σ'

- inductively defined relation
- expresses $\sigma \xrightarrow{\text{stmt}} \sigma'$

$$\text{If-True} \frac{\begin{array}{c} \text{cond} \Downarrow_{\sigma} \text{true} \quad \sigma \xrightarrow{\text{then}} \sigma' \\ \hline \sigma \xrightarrow{\text{if cond then then else else}} \sigma' \end{array}}{\sigma \xrightarrow{\text{if cond then then else else}} \sigma'}$$

becomes

```
eval-if-true : ∀ {cond then σ'} →  
  EvalExpr σ cond true →  
  EvalStmts σ then σ' →  
  (else : List (Stmt Γ)) →  
  EvalStmt σ (if cond then else) σ'
```

Table of Contents

Preliminaries

While Language

Control Flow Graphs

Translation

Conclusion

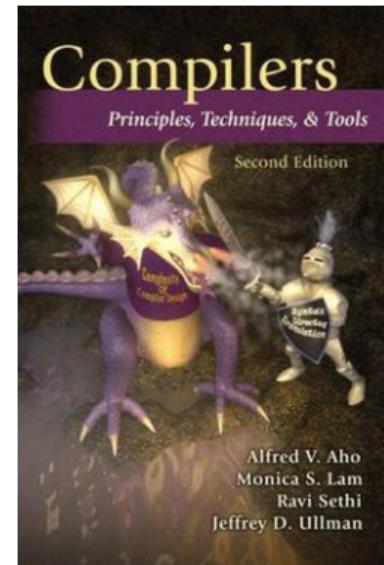
Recap: Control-Flow Graphs

- Intermediate Representation during compilation
- Directed graph that depicts control flow of a program
 - Basic Blocks (Nodes): Instructions without Jumps
 - Edges: Jumps

Recap: Control-Flow Graphs

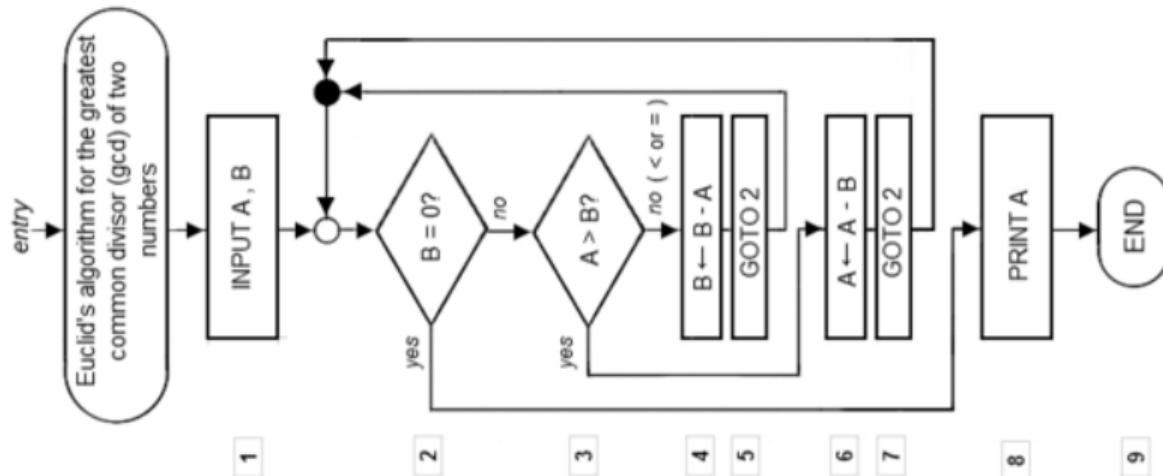
- “1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that
- (a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
 - (b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a *flow graph*, whose edges indicate which blocks can follow which other blocks.”

[Aho et al. (Chapter 8.4)]



https://upload.wikimedia.org/wikipedia/en/a/a3/Purple_dragon_book_b.jpg

Bsp. Control-Flow Graph



http://commons.wikimedia.org/wiki/File:Euclid_flowchart_1.png

Control-Flow Graphs

- Basic idea: Use of de Bruijn indices to represent labels
- Intrinsically typed syntax: label not number but types of stack elements
⇒ jumps only possible if current stack types match target stack types

Memory Model

Two memories:

- Environment (size and element types fixed)
- Stack

Types of Elements in Memories

Types of:

- elements in environment: EnvTypes = List Type
- elements on stack: StackTypes = List Type

Example Environments and Stacks

Agda Type of environments and stacks

- Env (num :: num :: [])
- Stack (num :: bool :: [])

Environments and Stacks

- (+ 1 , - + 2 , tt)
- (+ 3 , true , tt)

Let and Fix

Constructs in functional programming languages to bind variables:

- `let res := fib 5 in (res + res)`
- `fix (lambda fac n. if n = 0 then 1 else n * fac (n-1))`

We use them not to introduce variables but to declare labels

Label Declarations

First View

```
let l:  
    Instr A;  
    Instr B;  
    ...  
in  
    Instr X;  
    Instr Y;  
...[jumps to l]...
```

```
fix l:  
    Instr 1;  
    Instr 2;  
...[jumps to l]...
```

Label Declarations

Second Assembly like View

```
Instr X;  
Instr Y;  
...[jumps to l]...  
l: Instr A;  
Instr B;  
...
```

```
l: Instr 1;  
Instr 2;  
...[jumps to l]...
```

Jumps

All (sub-)flow graphs end with one of:

- stop
- goto l
- branch $l_1 \ l_2$

branch $l_1 \ l_2$ discards a boolean value from the stack
and jumps depending on it to either l_1 or l_2 .

Jump-Free Instructions

- **(load cell):** loads content of environment cell and pushes it on stack
- **(store cell):** discards top stack element and stores into environment cell
- **(push val):** pushes value on top of the stack
- **pop:** removes top element from stack
- **add, mul, eq, le:** performs operation with the two top elements on stack

Appending Jump Free Instructions to a Flow Graph

$fg ::= \text{stop} \mid \text{goto } l \mid \text{branch } l/l \mid \text{let } l : fg \text{ in } fg \mid \text{fix } l : fg \mid jfinstr; fg$

- One more kind of flow graphs: $jfinstr; fg$
- Prepends jump-free instruction to a flow graph

Intrinsically Typed Syntax of Flow Graphs

```
data FlowGraph (labels : List StackTypes) (env-types : EnvTypes) :  
  StackTypes → Set where ...
```

defines Agda type: `FlowGraph` labels env-types stack-types

of (Sub-)Flow Graphs that

- may contain jumps with labels in: labels
- starts with environment with types: env-types
- starts with stack with types: stack-types

Intrinsically Typed Syntax of Flow Graphs

```
data FlowGraph (labels : List StackTypes) (env-types : EnvTypes) :  
  StackTypes → Set where ...
```

Paramater labels: list of labels that can be used as targets for jumps in (sub-)flow graph

One label:

- is not just a number
- but is the list of all stack types in the jump destination
⇒ has Agda type: `StackTypes`

Jumps in Intrinsically Typed Flow Graph Syntax

```
data FlowGraph (labels : List StackTypes) (env-types : EnvTypes) :  
  StackTypes → Set where
```

...

```
goto : ∀ {stack-types} →  
  stack-types ∈ labels →  
  FlowGraph labels env-types stack-types
```

...

Remark about List of Labels for Entire Flow Graphs

- Labels are introduced by let and fix
- Jumps go always out of a given Sub-Flow Graph
⇒ Overall Flow Graphs are defined relative to the empty list of labels
(and have Agda Type: `FlowGraph [] env-types stack-types`)

Summary: Flow Graphs

$fg ::= \text{stop} \mid \text{goto } l \mid \text{branch } ll' \mid \text{let } l : fg \text{ in } fg \mid \text{fix } l : fg \mid \text{jfinstr; } fg$

Abstract syntax in Agda:

- `stop`
- `(goto l)`
- `(branch ll')`
- `(fglet fg fg')`
- `(fgfix fg)`
- `(jf-instr instr fg)`

Flow Graph Example with Jump-Free Instructions

Schematic:

```
load x;  
load y;  
store x;  
store y;  
stop
```

In Agda:

```
exmpl1 : FlowGraph [] (num :: num :: []) []  
exmpl1 =  
    jf-instr (load (here refl))  
        (jf-instr (load (there (here refl)))  
            (jf-instr (store (here refl))  
                (jf-instr (store (there (here refl)))  
                    stop)))
```

Flow Graph Example with Branching

Assembly-like:

```
branch l1 l2
l1: push 3;
    stop
l2: stop
```

Schematic:

```
let l2:
    stop
in let l1:
    push 3;
    stop
in
    branch l1 l2
```

In Agda:

```
exmpl2 : FlowGraph [] []
exmpl2 =
  fglet stop
    (fglet (jf-instr (push (+ 3)) stop)
      (branch (here refl) (there (here refl))))
```

Endless Loop Example

Assembly-like:

```
l: goto l
```

Schematic:

```
fix l:  
    goto l
```

In Agda:

```
exmpl3 : FlowGraph [] [] []  
exmpl3 =  
  fgfix  
    (goto (here refl))
```

Nonexample

Assembly-like:

```
l: push 2;  
    goto l
```

Schematic:

```
fix l:  
  push 2;  
  goto l
```

In Agda:

Yet an Example

Assembly-like:

```
l: push 2;  
    stop
```

Schematic:

```
fix l:  
    push 2;  
    stop
```

In Agda:

```
exmpl5 : FlowGraph [] [] []  
exmpl5 =  
  ffix  
    (jf-instr (push (+ 2))  
      stop)
```

Execution of Flow Graphs

Execute

```
let l:  
    Instr A;  
    Instr B;  
    ...  
in  
    Instr X;  
    Instr Y;  
    ...  
    goto l
```

...

Execution of Flow Graphs

by executing

```
Instr X;  
Instr Y;  
...  
goto l
```

- Where to jump?
- We need to store sub-flow graphs

Storing Sub-Flow Graphs

Helper function that determines Agda type of flow graphs that matches given list of labels:

```
flowgraphs-for-labels : EnvTypes → List StackTypes → Set
flowgraphs-for-labels _ [] = ⊤
flowgraphs-for-labels env-types (l :: labels) =
  FlowGraph labels env-types l × flowgraphs-for-labels env-types labels
```

Semantics

EvalFlowGraph $fgs \text{ env } st \xrightarrow{fg} \text{env}' \text{ st}'$

- inductively defined relation
- for now let's denote it as: $|fgs| \langle \text{env}, st \rangle \xrightarrow{fg} \langle \text{env}', st' \rangle$
- fgs : list of sub-flow graphs that correspond to labels

Semantics of Let

to execute `let fg1 in fg2`:

- add $fg1$ to fgs
- and execute $fg2$

$$\text{Eval-Let} \frac{|fg1, fgs| \langle \text{env}, \text{st} \rangle \xrightarrow{fg2} \langle \text{env}', \text{st}' \rangle}{|fgs| \langle \text{env}, \text{st} \rangle \xrightarrow{\text{let } fg1 \text{ in } fg2} \langle \text{env}', \text{st}' \rangle}$$

`eval-let` : $\forall \{ \text{stack-types} \text{ stack-types}' \text{ stack-types}'' \text{ env}'' \}$
 $\{ \text{fg} : \text{FlowGraph} \text{ labels env-types stack-types}' \}$
 $\{ \text{fg}' : \text{FlowGraph} \{ \text{stack-types}' :: \text{labels} \} \text{ env-types stack-types} \}$
 $\{ \text{st} : \text{Stack} \text{ stack-types} \} \{ \text{st}'' : \text{Stack} \text{ stack-types}'' \} \rightarrow$
`EvalFlowGraph` (fg , fgs) env st fg' env'' $\text{st}'' \rightarrow$
`EvalFlowGraph` fgs env st (`fglet` fg fg') env'' st''

Semantics of Fix

to execute $\text{fix } fg$:

- add $\text{fix } fg$ to fgs
- and execute fg

$$\text{Eval-Fix} \frac{| \text{fix } fg, fgs | \langle \text{env}, \text{st} \rangle \xrightarrow{fg} \langle \text{env}', \text{st}' \rangle}{| fgs | \langle \text{env}, \text{st} \rangle \xrightarrow{\text{fix } fg} \langle \text{env}', \text{st}' \rangle}$$

eval-fix : $\forall \{ \text{stack-types stack-types' env'} \}$
 $\{ fg : \text{FlowGraph} (\text{stack-types} :: \text{labels}) \text{ env-types stack-types} \}$
 $\{ st : \text{Stack stack-types} \} \{ st' : \text{Stack stack-types'} \} \rightarrow$
 $\text{EvalFlowGraph} (\text{fgfix } fg, fgs) \text{ env st fg env' st'} \rightarrow$
 $\text{EvalFlowGraph} fgs \text{ env st } (\text{fgfix } fg) \text{ env' st'}$

Semantics of Stop and Jump-Free Instr. Prepended FGs

$$\text{Eval-Stop} \quad \frac{}{|fgs| \langle env, st \rangle \xrightarrow{\text{stop}} \langle env, st \rangle}$$

$$\text{Eval-JFInstr} \quad \frac{\langle env, st \rangle \xrightarrow{jfinstr} \langle env', st' \rangle \quad |fg| \langle env', st' \rangle \xrightarrow{fg} \langle env'', st'' \rangle}{|fgs| \langle env, st \rangle \xrightarrow{jfinstr; fg} \langle env'', st'' \rangle}$$

Semantics of Goto

$$\text{Eval-Goto} \quad \frac{|fgs_I| \langle env, st \rangle \xrightarrow{fg_I} \langle env', st' \rangle}{|fgs| \langle env, st \rangle \xrightarrow{\text{goto } I} \langle env', st' \rangle}$$

- fg_I is the flow graph that is stored in fgs for label I
- fgs_I are the flow graphs that correspond to the labels of fg_I

Semantics of Branchings

$$\text{Eval-Branch-True} \frac{|fgs_{I1}| \langle env, st \rangle \xrightarrow{fg_{I1}} \langle env', st' \rangle}{|fgs| \langle env, \frac{\text{true}}{st} \rangle \xrightarrow{\text{branch } I1\ I2} \langle env', st' \rangle}$$

$$\text{Eval-Branch-False} \frac{|fgs_{I2}| \langle env, st \rangle \xrightarrow{fg_{I2}} \langle env', st' \rangle}{|fgs| \left\langle env, \frac{\text{false}}{st} \right\rangle \xrightarrow{\text{branch } I1\ I2} \langle env', st' \rangle}$$

Table of Contents

Preliminaries

While Language

Control Flow Graphs

Translation

Conclusion

Translator

- Defined using continuation passing style

```
translateStmt : {Γ : Context} → Stmt Γ → {labels : List StackTypes} →  
FlowGraph labels Γ [] → FlowGraph labels Γ []
```

```
translateStmt (assign x expr) fg =  
  translateExpr expr (jf-instr (store x) fg)
```

```
translateStmt (if cond then else) fg =  
  fglet fg  
    (fglet (translateStmts then (goto (here refl)))  
      (fglet (translateStmts else (goto (there (here refl))))  
        (translateExpr cond (branch (there (here refl)) (here refl))))))
```

```
translateStmt (while cond body) fg =  
  fglet fg  
    (fgfix  
      (fglet (translateStmts body (goto (here refl)))  
        (translateExpr cond (branch (here refl) (there (there (here refl))))))))
```

Translation of If and While Statements

Schematic View

$\llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \rrbracket =$

let I_0 :
 continuation
in let I_1 :
 $\llbracket S_1 \rrbracket$
 goto I_0
in let I_2 :
 $\llbracket S_2 \rrbracket$
 goto I_0
in
 $\llbracket b \rrbracket$
 branch $I_1 I_2$

$\llbracket \text{while } b \text{ do } S \rrbracket =$

let I_0 :
 continuation
in fix I_1 :
 let I_2 :
 $\llbracket S \rrbracket$
 goto I_1
 in
 $\llbracket b \rrbracket$
 branch $I_2 I_0$

Translation of If and While Statements

Assembly-like View

$\llbracket \text{if } b \text{ then } S_1 \text{ else } S_2 \rrbracket =$

$\llbracket b \rrbracket$
branch $I_1 I_2$
 $I_2: \llbracket S_2 \rrbracket$
goto I_0
 $I_1: \llbracket S_1 \rrbracket$
goto I_0
 $I_0: \text{continuation}$

$\llbracket \text{while } b \text{ do } S \rrbracket =$

$I_1: \llbracket b \rrbracket$
branch $I_2 I_0$
 $I_2: \llbracket S \rrbracket$
goto I_1
 $I_0: \text{continuation}$

Correctness

- Correctness statement:

$$\sigma \xrightarrow{\text{prog}} \sigma' \Rightarrow |_{-} \langle \sigma, - \rangle \xrightarrow{[\text{prog}]} \langle \sigma', - \rangle$$

- In Agda:

```
 $\forall \{\Gamma\} \{\text{prog} : \text{List } (\text{Stmt } \Gamma)\} \{\sigma \sigma'\} \rightarrow$ 
   $\text{EvalStmts } \sigma \text{ prog } \sigma' \rightarrow$ 
   $\text{EvalFlowGraph tt } (\text{translateState } \sigma) \text{ tt } (\text{translate prog}) (\text{translateState } \sigma') \text{ tt}$ 
```

- Proof is an Agda function definition
- Defined again using continuation passing style

Table of Contents

Preliminaries

While Language

Control Flow Graphs

Translation

Conclusion

Conclusion

- Intrinsically typed syntax and big-step semantics of both:
 - While language
 - Control-flow graph representation
- Translation from while programs to flow graphs
- Correctness proof of translation

Contributions

Ideas by Andreas:

- Flow graph representation
- Translation to flow graphs

Authors Contributions:

- Semantics definition of flow graphs
- Correctness proof
- Further development of flow graph representation

from $fg ::= \text{return} \mid \text{goto } l \mid \text{branch } fg \ fg \mid \text{let } fg \ \text{in } fg \mid \text{fix } fg \mid jfinstr; fg$
to $fg ::= \text{stop} \mid \text{goto } l \mid \text{branch } l \ l \mid \text{let } fg \ \text{in } fg \mid \text{fix } fg \mid jfinstr; fg$

Further Work

- Variable declarations
 - ⇒ environment not fixed
 - ⇒ label: list of stack types plus list of environment types
 - done by Andreas
- Function definitions
 - ⇒ return instead of stop flow graph
 - done by Andreas
- Second compiler pass
 - done by Andreas
 - not yet verified?