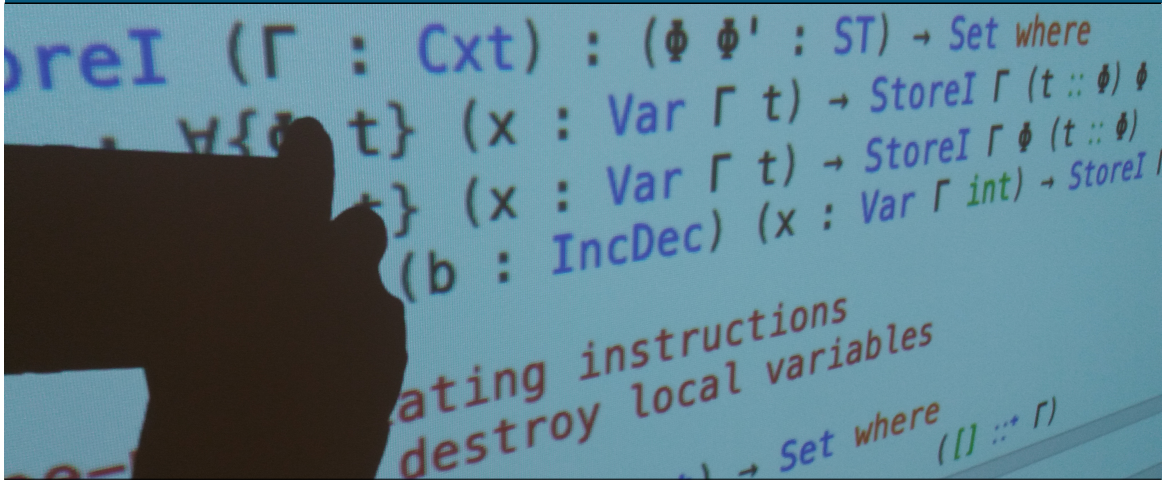
Verified Compilation to Intrinsically Typed Control-Flow Graphs in Agda

TECHNISCHE UNIVERSITÄT DARMSTADT



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

Agda

- 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

Table of Contents

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:

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

Types:

$$t ::= t \to t \mid \dots$$

- variables: x, y, z
- function abstractions:
  - $\lambda n : \mathbb{N}.\, n + n$
  - $\lambda n : \mathbb{N}.\, n^2 + 3 * n$
  - $\lambda f : \mathbb{N} \to \mathbb{N}.\, f\, 1$
- function application:
  - $(\lambda n : \mathbb{N}.\, n + n)\, 5$
  - $(\lambda n : \mathbb{N}.\, n^2 + 3 * n)\, 4$
  - $(\lambda f : \mathbb{N} \to \mathbb{N}.\, f\, 1)\, (\lambda n : \mathbb{N}.\, n + n)$

Simply Typed Lambda Calculus

beta-Reduction:

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

Example

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

De Bruijn Indices

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

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

becomes

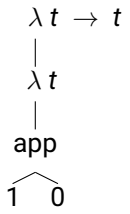
$$e ::= n \mid \quad \lambda t.\, e \mid e\, e$$

De Bruijn Indices - Example

$$\lambda f : (t \to t).\, \lambda x : t.\, f\, x$$

becomes:

$$\lambda\, t \to t.\, \lambda\, t.\, 1\, 0$$



Simply Typed Lambda Calculus in Agda

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

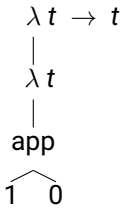
```
data Term (n : ℕ) : Set where
  ind : Fin n → Term n
  abs : Type → Term (suc n) → Term n
  app : Term n → Term n → Term n
```

n $\hat{=}$ number of free variables

Fin n $\hat{=} \{0, 1, \ldots, n-1\}$

Example

$$\lambda f : (t \to t).\, \lambda x : t.\, f\, x$$



with de Bruijn indices:

$$\lambda\, t \to t.\, \lambda\, t.\, 1\, 0$$

in Agda:

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

Type System of Simply Typed Lambda Calculus

Types:

$$t ::= b \mid t \to t$$

Contexts:

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

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

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

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash e\, e' : t'} \text{ (app)}$$

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 t t′)
  app : ∀ {t t′} → Term Γ (arrow t t′) → Term Γ t → Term Γ t′
```

`Term Γ t` : 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 t t′)
  app : ∀ {t t′} → Term Γ (arrow t t′) → Term Γ t → Term Γ t′
```

`Term Γ t` : 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 : ℕ) generalizes to (Γ : Context)
- Constructor argument Fin n generalizes to pair of type t and proof of t ∈ Γ

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 ∈ Γ` : Constructive proof that type t in contained in context Γ
⇒ contains information where t occurs in Γ

Example

$$\lambda f : (t \to t).\, \lambda x : t.\, f\, x$$

with de Bruijn indices:

$$\lambda\, t \to t.\, \lambda\, t.\, 1\, 0$$

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

in intrinsically typed syntax:

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

Table of Contents

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

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

- 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 contect 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 (Γ : Context) : Type → Set where
  var :  {t : Type} → t ∈ Γ → Expr Γ t
  lit :  {t : Type} → Val t → Expr Γ t
  add : (e e′ : Expr Γ num) → Expr Γ num
  mul : (e e′ : Expr Γ num) → Expr Γ num
  eq : (e e′ : Expr Γ num) → Expr Γ bool
  le  : (e e′ : Expr Γ num) → Expr Γ bool

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

Summary: Expressions

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

Summary: Statements

- *x* := *expr*: (assign x expr)
- `if` *cond* `then` *then* `else` *else*: (if cond then else)
- `while` *cond* `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:

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

$$\text{If-True}\ \frac{b \Downarrow_\sigma true \qquad \sigma \xrightarrow{S_1} \sigma'}{\sigma \xrightarrow{\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2} \sigma'}$$

Semantics
Remaining Rules

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

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

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

Agda Formalization of Semantics

EvalStmt σ stmt σ'

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

$$\text{If-True}\ \frac{cond \Downarrow_\sigma true \qquad \sigma \xrightarrow{then} \sigma'}{\sigma \xrightarrow{\texttt{if}\ cond\ \texttt{then}\ then\ \texttt{else}\ else} \sigma'}$$

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

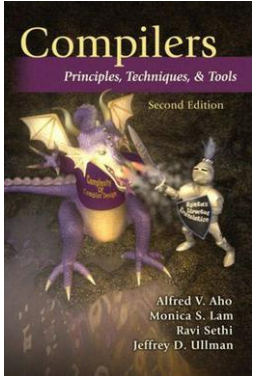
Recap: Control-Flow Graphs

- 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

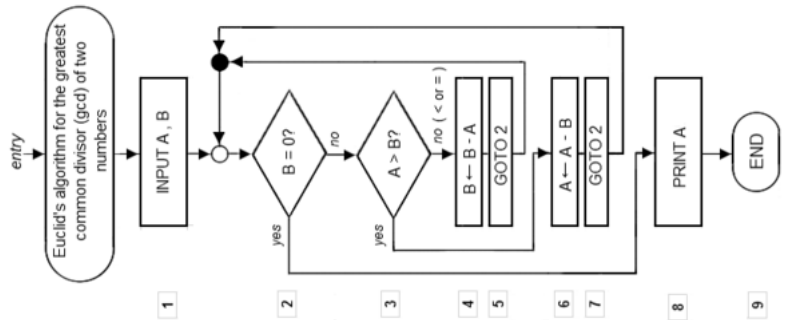
"1. Partition the intermediate code into *basic blocks*, which are maximal sequences of consecutive three-address instructions with the properties that
(a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the 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
  $\Rightarrow$ 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 pushs 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

Prepending Jump Free Instructions to a Flow Graph

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

- One more kind of flow graphs: $\mathit{jfinstr};\ fg$
- 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
  $\Rightarrow$ Overall Flow Graphs are defined relative to the empy list of labels
  (and have Agda Type: FlowGraph [] env-types stack-types)

Summary: Flow Graphs

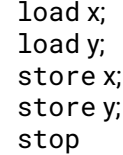
$$fg ::= \texttt{stop} \mid \texttt{goto}\ l \mid \texttt{branch}\ l\ l \mid \texttt{let}\ l : fg\ \texttt{in}\ fg \mid \texttt{fix}\ l : fg \mid jfinstr;\ fg$$

Abstract syntax in Agda:

- stop
- (goto l)
- (branch l l′)
- (jf-instr instr fg)
- (fglet fg fg′)
- (fgfix fg)

Flow Graph Example with Jump-Free Instructions

Schematic:



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 [] [] (bool :: [])
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
```

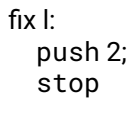
In Agda:

Yet an Example

Assembly-like:

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

Schematic:



In Agda:

```
exmpl5 : FlowGraph [] [] []
exmpl5 =
  fgfix
    (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 env st fg env′ st′

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

Semantics of Let

to execute `let` *fg*1 `in` *fg*2:

- add *fg*1 to *fgs*
- and execute *fg*2

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

```
eval-let : ∀ {stack-types stack-types′ stack-types″ env″}
  {fg : FlowGraph labels env-types stack-types′}
  {fg′ : FlowGraph (stack-types′ :: labels) env-types stack-types}
  {st : Stack stack-types} {st″ : Stack stack-types″} →
  EvalFlowGraph (fg , fgs) env st fg′ env″ st″ →
  EvalFlowGraph fgs env st (fglet fg fg′) env″ st″
```

Semantics of Fix

to execute `fix` *fg*:

- add `fix` *fg* to *fgs*
- and execute *fg*

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

```
eval-fix : ∀ {stack-types stack-types′ env′}
  {fg : FlowGraph (stack-types :: labels) env-types stack-types}
  {st : Stack stack-types} {st′ : Stack stack-types′} →
  EvalFlowGraph (fgfix fg , fgs) env st fg env′ st′ →
  EvalFlowGraph fgs env st (fgfix fg) env′ st′
```

Semantics of Stop and Jump-Free Instr. Prepended FGs

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

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

Semantics of Goto

$$\text{Eval-Goto}\ \frac{|fgs_l|\ \langle env, st\rangle \xrightarrow{fg_l} \langle env', st'\rangle}{|fgs|\ \langle env, st\rangle \xrightarrow{\texttt{goto}\ l} \langle env', st'\rangle}$$

- fg_l is the flow graph that is stored in fgs for label l
- fgs_l are the flow graphs that correspond to the labels of fg_l

Semantics of Branchings

$$\text{Eval-Branch-True}\ \frac{|fgs_{l1}|\ \langle env, st\rangle \xrightarrow{fg_{l1}} \langle env', st'\rangle}{|fgs|\ \left\langle env, \frac{true}{st}\right\rangle \xrightarrow{\texttt{branch}\ l1\ l2} \langle env', st'\rangle}$$

$$\text{Eval-Branch-False}\ \frac{|fgs_{l2}|\ \langle env, st\rangle \xrightarrow{fg_{l2}} \langle env', st'\rangle}{|fgs|\ \left\langle env, \frac{false}{st}\right\rangle \xrightarrow{\texttt{branch}\ l1\ l2} \langle env', st'\rangle}$$

Table of Contents

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

$[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] =$

let l_0:
    continuation
in let l_1:
        $[\![S_1]\!]$
        `goto` l_0
    in let l_2:
            $[\![S_2]\!]$
            `goto` l_0
        in
            $[\![b]\!]$
            `branch` l_1 l_2

$[\![\texttt{while}\ b\ \texttt{do}\ S]\!] =$

let l_0:
    continuation
in fix l_1:
        let l_2:
            $[\![S]\!]$
            `goto` l_1
        in
            $[\![b]\!]$
            `branch` l_2 l_0

Translation of If and While Statements

Assembly-like View

$[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]$ =

$[\![b]\!]$
  `branch` l_1 l_2
l_2: $[\![S_2]\!]$
  `goto` l_0
l_1: $[\![S_1]\!]$
  `goto` l_0
l_0: continuation

$[\![\texttt{while } b \texttt{ do } S]\!]$ =

l_1: $[\![b]\!]$
  `branch` l_2 l_0
l_2: $[\![S]\!]$
  `goto` l_1
l_0: continuation

Correctness

- Correctness statement:

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

- In Agda:

```
∀ {Γ} {prog : List (Stmt Γ)} {σ σ′} →
  EvalStmts σ prog σ′ →
  EvalFlowGraph tt (translateState σ) tt (translate prog) (translateState σ′) tt
```

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

Table of Contents

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 $fg ::= \texttt{return} \mid \texttt{goto}\ l \mid \texttt{branch}\ fg\ fg \mid \texttt{let}\ fg\ \texttt{in}\ fg \mid \texttt{fix}\ fg \mid jfinstr;\ fg$

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

Further Work

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