

Verified Compilation to Intrinsically Typed Control-Flow Graphs in Agda

Master thesis by Alexander Fuhs
Date of submission: April 27, 2020

1. Review: Prof. Dr. rer. nat. Reiner Hähnle
2. Review: Dr. habil. Andreas Abel
Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Software Engineering Group

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Alexander Fuhs, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 27. April 2020

A. Fuhs

Contents

1	Introduction	4
2	Agda	6
3	Preliminaries	11
3.1	De Bruijn Indices	12
3.2	Intrinsically Typed Syntax	14
4	While Language	17
4.1	Syntax	17
4.2	Semantics	20
5	Control-Flow Graphs	24
5.1	Label Declarations	25
5.2	Jumps	26
5.3	Memory Model	27
5.4	Jump-free Instructions	28
5.5	Flow Graph Representation	31
5.5.1	Syntax	31
5.5.2	Semantics	37
6	Translation	42
6.1	Implementation	42
6.2	Correctness	50
7	Related Work	56
8	Conclusion	60

1 Introduction

Compilers are fundamental software products since they enable the use of high-level programming languages in the development of computer software. Compilers translate those abstract and more human readable languages into machine code that can be executed by computer processors or to bytecode that can be interpreted by virtual machines. The correctness of compilers is a critical ingredient in the development of computer systems since faulty compilers may produce faulty programs even for programs that have a correct source code. But as writing software products in general also developing compilers is an error-prone task. Yang et al.[59] reported more than 325 bugs (among them compiler crashes as well as wrong generated code) in 11 C[22] compilers that were found with the help of the randomized test-case generator tool Csmith[14]. They focused on the gcc[15] and LLVM[56, 25] compilers for which they reported alone 79 resp. 202 bugs. Le et al.[26] counted 147 confirmed and unique bug reports for gcc and LLVM (79 for gcc and 68 for LLVM) using a validation methodology for optimizing compilers called equivalence modulo inputs.

To ensure the correctness of computer programs tools were developed which allow to formally verify the correctness of source code. Although theoretical reasons prohibit the full automation of checking that programs fulfill their specifications, tools like KeY[55, 2] and Why3[57, 9] have been developed that achieve this goal by requiring different degrees of interaction. Faulty compilers run contrary to those efforts to verify a programs source code since they might change the semantics of the compiled programs.

An obvious solution to guarantee that trustworthy source programs are translated to trustworthy machine code or bytecode is to apply verification to compilers as well. One compiler verification project is CompCert[30]. In it a verified compiler from a subset of the programming language C to a subset of the PowerPC assembly language is developed in the proof assistant Coq[54, 5]. The before mentioned works by both Yang et al. and Le et al. both appreciate that they could not find any bugs in the verified parts of the CompCert compiler. This shows that compiler verification can dramatically reduce the

number of errors in the development of compilers. Nevertheless not the entire CompCert compiler is verified and indeed both works found bugs in its unverified parts.

In this thesis we consider control-flow graphs as an intermediate representation in the compilation of imperative programs and use the tool Agda[53] to verify the translation of programs in a prototypical imperative programming language, also called while language¹, into these control-flow graphs. We present a flow graph representation which basic idea is to represent labels by de Bruijn indices. We use a technique called intrinsically typed syntax[4] to define the abstract syntax of both the while language and our flow graph representation. We do not consider parsing but only deal with the abstract syntax. The ideas of how to represent control-flow graphs and how to translate while programs to it were developed by the thesis supervisor who is also the second reviewer of the thesis. The contribution of the thesis author is to give a semantics for this representation and to use it to prove the correctness of the translation².

In Chapter 2 we explain the features of Agda that we use in this thesis. In Chapter 3 we introduce some further technical preliminaries. We define the syntax and semantics of the while language in Chapter 4. In Chapter 5 we introduce the representation of control-flow graphs and define its syntax and semantics. In Chapter 6 we define the translator and proof its correctness. Finally we discuss related work in Chapter 7 and conclude in Chapter 8.

¹A similar language was called “IMP” by Winskel[58] and “While” by Nielson and Nielson[40].

²All the Agda code in this thesis is written by the author of the thesis except for the definitions in the first five figures in Chapter 2. They are taken from the Agda Built-in library, except for the definition of vectors in Figure 2.4 which is taken from the Agda standard library[52].

2 Agda

This chapter introduces some basic Agda[53, 1] features that are used throughout this thesis. Agda is a functional programming language whose type system is based on Martin-Löf’s intuitionistic type theory. According to the Curry-Howard correspondence or propositions-as-types paradigm, which considers constructive proofs as programs of a functional programming languages, Agda can also be used to state propositions and to prove them.

In Agda, like in Coq[54, 5], it is possible to introduce new data types or propositions by inductive definitions[13]. Figure 2.1 shows how the natural numbers can be defined in Agda. The inductive definition introduces a new type `Nat` as well as two terms: `zero` of type `Nat` and `suc` of type `Nat → Nat` which represent the natural number 0 resp. the successor function of natural numbers which assigns to each natural number its successor. The natural numbers can now be enumerated by `zero`, `suc zero`, `suc (suc zero)`, ...¹.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat

{-# BUILTIN NATURAL Nat #-}
```

Figure 2.1: Natural numbers.

The terms introduced by an inductive definition are called “constructors” since they can be used to construct the elements of the introduced data type. The constructors of each inductive definition fulfill further properties, like: (1) the disjointedness of the set of constructors of an inductive definition, (2) the injectivity of the constructors and (3) the

¹The directive in the last line of Figure 2.1 allows one to use also number literals `0`, `1`, `2`,

totality of the constructors: there are no elements of the introduced data type other than the ones that can be constructed by the constructors. In the case of the data type `Nat` this reflects the so called “Peano Axioms”, which are: (1) 0 is not the successor of any natural number, (2) the injectivity of the successor function and (3) the induction scheme of natural numbers.

Also the booleans in Agda are defined in this way (see Figure 2.2).

```
data Bool : Set where
  false true : Bool
```

Figure 2.2: Booleans.

Furthermore one can use inductive definitions to define polymorphic data types like polymorphic lists in Figure 2.3. Here the introduced data type `List` depends on some type `A` that is the type of the elements in the list. `List Nat` for example is the type of lists of natural numbers, `List Bool` is the type of lists of booleans and `List (List Nat)` is the type of lists of lists of natural numbers. The constructors of lists are the empty list `[]`, and `_::_`, which prepends a list with a further element. Agda has a feature called mixfix operators, that allows one e.g. to define infix operators by writing underscores for the positions where the arguments can occur. The underscores in the constructor `_::_` for example allows one to write the list with element 2 as `2 :: []`. The directive in the last line makes the operator `::` right-associative, which allows to write lists with more than one element like `0 :: 1 :: 2 :: []` and `0 :: 1 :: 1 :: 0 :: []` without the need for parentheses.

```
data List {a} (A : Set a) : Set a where
  [] : List A
  _::_ : (x : A) (xs : List A) → List A

infixr 5 _::_
```

Figure 2.3: Lists.

Agda has a hierarchy of types `Set0`², `Set1`, `Set2`, The parameter `a` in Figure 2.3 indicates a level in this hierarchy and allows one to use e.g. lists of types like `Bool :: Nat ::`

²`Set0` can be abbreviated as `Set`.

`[]` of type `List Set`. The curly brackets around `a` make it to an implicit parameter. Implicit parameters do not need to be stated but are inferred by Agda.

Another feature of Agda are dependent types, that are types that depend on terms of some other type. Figure 2.4 shows the definition of so called vectors that are just lists of a certain length. Compared to the definition of lists in Figure 2.3 it has in addition to the parameter `A` a so called index of type `Nat` which denotes the length of the vector. `Vec A n` is the type of all vectors with elements of type `A` that have the length `n`. Agda allows constructor overloading. Thus the same constructor names `[]` and `_::_` as in the definition of lists in Figure 2.3 can be used. `[]` is a term of type `Vec Bool 0` and `0 :: 1 :: 2 :: []` is a term of type `Vec Nat 3`.

```
data Vec {a} (A : Set a) : Nat → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

Figure 2.4: Vectors.

It is possible to define recursive functions on inductively defined data types. In this way one can define addition of natural numbers (Figure 2.5). The first line is the type declaration³, which tells us that addition takes two natural numbers as arguments and yields another natural number as result. The function is defined by pattern matching on the first argument. In the successor case the function has a recursive function call. In the recursive function call the first argument is `n` instead of `suc n`, i.e. it is structurally smaller. This principle called “structural recursion” has to be fulfilled in all Agda function definitions. Hence all Agda functions are assured to be terminating⁴. This is important since, following the Curry-Howard correspondence, proofs in Agda are given by function definitions. Ignoring the structural recursion principle would for example in the induction step for a proposition about natural numbers allow to use the induction hypothesis on $n + 1$ instead of n . This would be an unsound reasoning.

Inductive definitions cannot only be used to define data types but also to define propositions.

³The given type declaration is the curried form of $\text{Nat} \times \text{Nat} \rightarrow \text{Nat}$. In Agda the \rightarrow -operation is associated to the right. Thus both declaration mean the same. It is not necessary to know the concept of currying for the understanding of this thesis but just to keep in mind that the argument types in an Agda function declaration are separated by \rightarrow instead of \times .

⁴Actually there are pragmas in Agda that allow one to switch off the termination checking and to write non-terminating functions.

```
_+_ : Nat → Nat → Nat
```

```
zero + m = m  
suc n + m = suc (n + m)
```

Figure 2.5: Addition.

Figure 2.6 shows the mutual definition of two natural number predicates `Even` and `Odd` which express the properties of a natural number to be even resp. odd.

The constructors of both inductive definitions determine which natural numbers are even resp. odd: `even-zero` determines that 0 is even, `even-suc` determines that the successor of an odd number is even and `odd-suc` determines that the successor of an even number is odd. These constructors can be used to prove the evenness and oddness of natural numbers. Proofs of `Even 0`, `Odd 1` and `Even 2` are given by the terms `even-zero`, `odd-suc even-zero` resp. `even-suc (odd-suc even-zero)`.

```
mutual  
data Even : Nat → Set where  
  even-zero : Even zero  
  even-suc : {n : Nat} → Odd n → Even (suc n)  
  
data Odd : Nat → Set where  
  odd-suc : {n : Nat} → Even n → Odd (suc n)
```

Figure 2.6: Even and odd.

Proofs of propositions can be given by function definitions. Recursive function calls serve as induction hypotheses of inductive proofs. Figure 2.7 shows how we can prove that the addition of two even numbers is again even. We prove it by implementing the function `even+even` which has the following type declaration:

```
even+even : {n m : Nat} → Even n → Even m → Even (n + m)
```

To prove this proposition we mutually prove a second proposition, which states that the addition of an odd and an even number is again an odd number:

`odd+even : {n m : Nat} → Odd n → Even m → Odd (n + m)`

We prove both propositions together by mutual induction. This is done by mutual recursive function calls in the definition of both functions.

Addition is defined recursively on the first argument in Figure 2.5. Hence we do both proofs by induction on the evenness resp. oddness of the first argument n . This is done by pattern matching on the argument of type `Even n` resp. `Odd n`. In the definition of the first function `even+even` we have one induction base and one induction step. In the induction base we have that the first even argument n is 0. By definition of addition (Figure 2.5) the addition of 0 and m is m . Hence the proof of the evenness of the sum is given by the proof of the evenness of m . In the induction step we have that the first even argument is the successor of odd number n . Now we can prove the evenness of the sum $((\text{suc } n) + m)$ which is by definition of addition equal to $(\text{suc } (n + m))$ by proving the oddness of $n + m$. This is achieved by the induction hypothesis in form of a mutual recursive function call of `odd+even`. In the definition of the second function `odd+even` we have no induction base but only an induction step, which is proven analogous with the induction hypothesis given by the function call to `even+even`.

`even+even : {n m : Nat} → Even n → Even m → Even (n + m)`

`odd+even : {n m : Nat} → Odd n → Even m → Odd (n + m)`

`even+even even-zero even-m = even-m`

`even+even (even-suc odd-n) even-m = even-suc (odd+even odd-n even-m)`

`odd+even (odd-suc even-n) even-m = odd-suc (even+even even-n even-m)`

Figure 2.7: Proof in Agda.

3 Preliminaries

In this chapter we explain de Bruijn indices and intrinsically typed syntax using the example of the simply typed lambda calculus. We will use these techniques in the subsequent chapters to define the syntax of both the while language and the control flow graphs.

The simply typed lambda calculus can be seen as a tiny typed functional programming language. Its programs which are also called lambda terms comprise variables, function abstractions and function applications. Its grammar is given by:

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

A function abstraction $\lambda x : t. e$ can be seen as an anonymous¹ function definition where the variable x is the parameter of the function and has type t and e is the function body. A function application is denoted by the juxtaposition of two lambda terms, the left one being the function and the right one being the argument.

The types can be specified by another grammar:

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

There is a base type b and for any types t_1 and t_2 there is the type $t_1 \rightarrow t_2$ of functions from t_1 to t_2 .

The following reduction rule, called beta reduction, explains how programs in this programming language are computed: the lambda abstraction $\lambda x : t. e$ is applied to some argument e' by replacing all occurrences of the variable x in the function body e with the argument e' .

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

¹It does not have a name but can only be applied directly on some argument.

For example the term $(\lambda n:\mathbb{N}. n^2 + 3 * n) 4$ which is the application of the function $(\lambda n:\mathbb{N}. n^2 + 3 * n)$ to the argument 4 reduces to $4^2 + 3 * 4$.

A type system is specified by a set of typing rules. Each typing rule consists of a conclusion and a possibly empty set of premises where both the conclusion and all the premises are sequents of the form $\Gamma \vdash e : t$, saying that in a context Γ a lambda term e has type t . A context Γ is just a list of pairs of variables and types and can be seen as a list of variable declarations: $x_1 : t_1, x_2 : t_2, \dots, x_n : t_n$

The type system of the simply typed lambda calculus consists of the following three typing rules:

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

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

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

A variable x has type t if the pair (x, t) occurs in the context Γ (rule var). A function abstraction $\lambda x : t. e$ has type $t \rightarrow t'$ if in the extended context $\Gamma, x : t$ where the pair (x, t) is added to Γ the expression e itself has type t' (rule abs). And a function application $e e'$ has type t' in context Γ if in the same context the expression e has type $t \rightarrow t'$ and expression e' has type t (rule app).

3.1 De Bruijn Indices

De Bruijn indices are a possible way to represent variables. Using de Bruijn indices the syntax of lambda terms looks like the following:

$e ::= n$	de Bruijn index
$\quad \lambda t. e$	function abstraction
$\quad e e$	function application

Rather than using some set of variables one uses natural numbers which are called de Bruijn indices. The de Bruijn index n thereby refers to the n th surrounding lambda abstraction (starting by zero). For example the lambda term

$$\lambda f : (b \rightarrow b). \lambda x : b. f x$$

is represented as:

$$\lambda b \rightarrow b. \lambda b. 1\ 0$$

The index 1 in this example refers to the outermost lambda binder with type $b \rightarrow b$ and index 0 refers to the innermost lambda binder of type b .

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

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
```

Figure 3.1: Syntax of the simply typed lambda calculus.

Figure 3.1 shows how the syntax of the simply typed lambda calculus can be defined in Agda. The first definition determines the types that lambda terms can have. They consists of one base type `base` and one type constructor `arrow` which takes two types as argument and represents the function type, i.e. `(arrow t t')` is the type of functions from `t` to `t'`. Lambda terms are defined by the inductive data type `Term` which has the number of free variables occurring in the lambda term as parameter, i.e. `Term 2` is the Agda type of lambda term with at most two free variables and `Term 0` is the Agda type of a closed lambda terms. In general `Term n` is the Agda type of terms in which at most n free variables occur. De Bruijn indices are represented by the constructor `ind` which has one argument of Agda type `Fin n`. `Fin n` represents a finite set of size n and has the elements `zero`, `suc zero`, \dots , `sucn-1 zero`. Function abstractions and function applications are represented by the constructors `abs` and `app`, i.e. $\lambda t. e$ and $e e'$ are represented as `(abs t e)` resp. `(app e e')`. The constructor `abs` has an argument of Agda type `Term (suc n)` rather than `Term n`. This is because the lambda abstraction binds one additional variable.

The lambda term $\lambda b \rightarrow b. \lambda b. 1\ 0$ in the example above is represented as:

$$\underbrace{\text{abs}}_{\lambda} \underbrace{(\text{arrow base base})}_{b \rightarrow b} \underbrace{(\text{abs base})}_{\lambda \ b} \underbrace{(\text{app} (\text{ind (suc zero)}) (\text{ind zero}))}_{1 \ 0}$$

Figure 3.2 presents the Agda definition of the type system of the simply typed lambda calculus presented earlier in this chapter. A context is now defined as a vector of types, i.e. a context Γ of type `Context n` consists of n types. The type of an index `ind i` is determined by the i -th type in the context Γ .

`Context = Vec Type`

```
data Typing {n : ℕ} (Γ : Context n) :
  Term n → Type → Set where
  typing-ind :
    (i : Fin n) →
    Typing Γ (ind i) (lookup i Γ)
  typing-abs : ∀ {t t' e} →
    Typing (t :: Γ) e t' →
    Typing Γ (abs t e) (arrow t t')
  typing-app : ∀ {t t' e e'} →
    Typing Γ e (arrow t t') →
    Typing Γ e' t →
    Typing Γ (app e e') t'
```

Figure 3.2: Type system of the simply typed lambda.

3.2 Intrinsically Typed Syntax

Figure 3.3 shows a definition of the simply typed lambda calculus using intrinsically typed syntax. Here the Agda type `Term` of lambda terms has a context Γ as parameter and the type of the lambda term in that context Γ as index. `Term Γ t` is thus the Agda type of lambda terms that have in context Γ the type t . The context thereby generalizes the natural number n in Figure 3.1. Similarly the argument of type $t \in \Gamma$ of the constructor

`ind` generalizes the argument of type `Fin n` of the same constructor in Figure 3.1. In general $a \in l$ is the proposition that an element a occurs in a list l . E.g. $2 \in 1 :: 2 :: 3 :: 4 :: []$ is the proposition that the natural number 2 occurs in the list $1 :: 2 :: 3 :: 4 :: []$. A proof of this proposition is given by `there (here refl)`, where `there` proves that an element is in a list provided a proof that the element is in the rest list and `here refl` is a proof that an element is the first element of a list. Remark that there are two different proofs of $1 \in 1 :: 0 :: 1 :: []$ namely `here refl` and `there (there (here refl))`. Thus a given proof of $a \in l$ also contains the information at which position the element a occurs in the list l . This will be important in our flow graph representation in Section 5.5 when we use the relation \in to specify the target label of a jump given the list of all labels in the flow graph.

```

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'

```

Figure 3.3: Intrinsically typed syntax of the simply typed lambda calculus.

Constructor arguments surrounded by curly brackets are implicit arguments, i.e. they do not need to be specified when using the constructors but Agda tries to infer them. For example an index can be written as `ind t-in-Γ` instead of `ind t t-in-Γ`, i.e. the first argument t is omitted. The lambda term $\lambda b \rightarrow b. \lambda b. 1 \ 0$ is now represented as:

```
abs (arrow base base) (abs base (app (ind (there (here refl))) (ind (here refl))))
```

Remark that the indices 0 and 1 are now represented as `ind (here refl)` resp. `ind (there (here refl))`.

The intrinsically typed syntax of the lambda calculus integrates the type system of the lambda calculus into its syntax (compare Figure 3.3 with both Figure 3.1 and Figure 3.2). As a consequence it is not possible to syntactically represent lambda terms that are not typeable. E.g. in the simply typed lambda calculus it is not possible to find a type t such

that the lambda term $\lambda f : t. f f$ is typeable, hence this term cannot be represented by an Agda term of data type [Term](#).

Intrinsic typing is an example of a strong specification as opposed to a weak specification². Strong specifications use dependent types to include the logical properties of a function into its type. For weakly specified functions one needs additional companion lemmas to capture those properties. Intrinsic typing is a strong specification where the property included in the Agda type is typeability of object terms.

²Both notions are also explained by Bertot and Castéran[5] (Chapter 9).

4 While Language

This chapter introduces a prototypical imperative programming language that serves as source language for the compiler in Chapter 6. We define its abstract syntax using the technique of intrinsic typing like explained in Section 3.2 as well as a big-step operational semantics¹ for this programming language.

4.1 Syntax

The language has two types, `num` and `bool`, representing integers and boolean values (cf. Figure 4.1 and Figure 4.2). A (typing) context is defined in Figure 4.1 as a list of types.

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

Context = List Type
```

Figure 4.1: Types and contexts.

```
Val : Type → Set
Val num = ℤ
Val bool = Bool
```

Figure 4.2: Values.

¹It is also called natural semantics[21].

The expressions are given by the inductively defined data type `Expr` in Figure 4.3. Since we use intrinsically typed syntax, `Expr` has a context Γ as parameter as well as an index of Agda type `Type` that specifies the type of the expression in that context, i.e. `Expr Γ t` is the Agda type of expressions that are typeable in a context Γ with type `t`. E.g. the expression that adds the integer literals 1 and 2 which is represented as `add (lit (+ 1)) (lit (+ 2))` has the Agda type `Expr Γ num` for some context Γ .

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

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

Figure 4.3: Syntax of the while language.

The expressions in Figure 4.3 include variables (`var`), literals (`lit`) and the arithmetic built-in functions addition (`add`), multiplication (`mul`) and tests of equality (`eq`) and “less than or equal” (`le`). The types of variables are determined by the context, the types of literals are determined by themselves, i.e. true and false have type `bool` and all integers have type `num`, addition and multiplication are functions that have arguments of type `num` and result type `num`, equality and “less than or equal” comparisons have arguments of type `num` and result type `bool`.

Example 4.1 $x : num, y : num \vdash x \leq y : bool$

Given a context with two integers `num :: num :: []` the expression that tests whether the first variable is smaller or equal than the second variable is represented by `le (var (here refl)) (var (there (here refl)))` and has Agda type `Expr (num :: num :: []) bool`. Variables in this example are represented like explained in Chapter 3.

The language comprises three different statements: assignments, if conditionals and while loops, which occur as the three constructors of the inductively defined data type `Stmt` in Figure 4.3. `Stmt` Γ is the Agda type of statements that are typeable in a context Γ . An assignments (`assign`) contains the target variable in form of a de Bruijn index like explained in Section 3.2 as well as the assigned expression. A conditional (`if`) contains a condition expression `cond` as well as two lists of statements `then` and `else`, which are the then resp. else branch of the conditional. A while loop (`while`) contains a condition expression `cond` and a list of statements `body` that is the body of the while loop. The typeability condition assures that expressions can only be assigned to variables in a program if the expressions are typeable and if the variables are of the same type than the expressions.

Example 4.2 The following program that computes the factorial function

```
r := 1;
while i >= 1
  r := r * i;
  i := i + (-1);
```

is represented as

```
assign
  (there (here refl))
  (lit (+ 1)) ::
while
  (le
    (lit (+ 1))
    (var (here refl)))
  (assign
    (there (here refl))
    (mul (var (there (here refl))) (var (here refl))) ::
  assign
    (here refl)
    (add (var (here refl)) (lit (- + 1))) ::
  []) ::
[]
```

and has the Agda type `List (Stmt (num :: num :: []))` since a program is a list of statements. There occur two variables in the program. Hence the statements are defined according to

a context `num :: num :: []`. The program variable i is represented as `here refl`; r as `there (here refl)`.

4.2 Semantics

The Agda type of program states² (Figure 4.4) depends on the context. For example `State (num :: num :: bool :: [])` is the Agda type of states for contexts of the form `num :: num :: bool :: []` and is by definition of `State` the same as $\mathbb{Z} \times \mathbb{Z} \times \text{Bool} \times \top$ ³.

```
State : Context → Set
State [] = ⊤
State (t :: Γ) = Val t × State Γ
```

Figure 4.4: States.

The semantics of expressions and statements are given by inductive definitions. For expressions the inductive definition of `EvalExpr` is given in Figure 4.5. The judgment `EvalExpr σ expr val` states that an expression `expr` which has in context Γ the type t evaluates in a state σ to the value `val`. Since t and Γ are implicit in the definition of `EvalExpr`, they do not occur in the judgment.

A variable evaluates to the value which is looked up in the state σ (cf. `eval-var`), a literal evaluates just to itself (cf. `eval-lit`) and to evaluate an addition, the arguments are evaluated first and then added (cf. `eval-add`). The evaluation of the remaining arithmetical expressions is defined analogous and therefore omitted in Figure 4.5.

The semantics of statements and statement lists in Figure 4.6 are given by the mutual inductive definition of `EvalStmt` and `EvalStmts`. The judgment `EvalStmt σ stmt σ'` expresses that the execution of a statement `stmt` in state σ yields to state σ' . Analogously `EvalStmts σ stmts σ'` expresses that the execution of a list of statements `stmts` in state σ yields to state σ' .

²A program state is also called an “environment” in the context of functional programming languages. Nevertheless we call it “state” in accordance with the literature about programming language semantics like Winskel[58] and Nielson and Nielson[40].

³ \top represents the set with the empty tuple as single element.

```

data EvalExpr {Γ : Context} (σ : State Γ) :
  {t : Type} → Expr Γ t → Val t → Set where
eval-var : ∀ {t} (x : t ∈ Γ) →
  EvalExpr σ (var x) (lookup-var x σ)
eval-lit : ∀ {t} (val : Val t) →
  EvalExpr σ (lit val) val
eval-add : ∀ {e e' i j} →
  EvalExpr σ e i →
  EvalExpr σ e' j →
  EvalExpr σ (add e e') (i + j)
  ...

```

Figure 4.5: Semantics of expressions.

An assignment (**assign** x $expr$) is executed by first evaluating the expression $expr$ and then updating the state with the resulting value val for the variable x (cf. **eval-assign** in Figure 4.6). To execute a conditional (**if** $cond$ **then** $else$) first the condition $cond$ is evaluated. Depending on whether it evaluates to **true** or **false** the resulting state σ' is the one to which the execution of the then branch **then** resp. the else branch **else** results in (cf. **eval-if-false** resp. **eval-if-true**). A while loop (**while** $cond$ $body$) does not change the state σ if its condition $cond$ evaluates to **false** (cf. **eval-while-false**). If the condition evaluates to **true**, the execution of the first unfolding of the while body $body$ in state σ results in σ' and the further execution of the while statement starting in σ' results in σ'' then the execution of the entire while statement in state σ results in σ'' (cf. **eval-while-true**). The execution of the empty list of statements $[]$ does not change the state (cf. **eval-skip**) and the resulting state of the execution of a non empty list of statements $stmt :: stmts$ is obtained by executing the first statement $stmt$ and then executing the rest statements $stmts$ (cf. **eval-seq**).

Example 4.3 The execution of the factorial program from the last section with arguments $i = 3$ and $r = 0$ ends in a state where $i = 0$ and $r = 6$ (the intermediate states are given in Table 4.1).

This is stated as **EvalStmts** $(+ 3, + 0, tt)$ **factorial** $(+ 0, + 6, tt)^4$.

⁴We assume we gave the factorial program the name **factorial**.

The proof of this statement in Agda starts with the following lines:

```
eval-seq
  (eval-assign
    (eval-lit (+ 1))
    (there (here refl)))
  (eval-seq
    (eval-while-true
      (eval-le (eval-lit (+ 1)) (eval-var (here refl)))
      (eval-seq
        (eval-assign
          (eval-mul (eval-var (there (here refl))) (eval-var (here refl)))
          (there (here refl)))
        (eval-seq
          (eval-assign
            (eval-add (eval-var (here refl)) (eval-lit (- + 1)))
            (here refl))
          eval-skip))
      ...))
    eval-skip)
```

The above extract shows the parts of the proof that are concerned with the assignment of 1 to r (`eval-assign (eval-lit (+ 1)) (there (here refl))`) and the first iteration of the while loop (`eval-while-true ...`), ending in a state where i is decreased to 2 and r has the value 3.

	i	r
start of program	3	0
after assignment	3	1
after first loop iteration	2	3
after second loop iteration	1	6
after last loop iteration	0	6

Table 4.1: States during execution of factorial program.

mutual

```
data EvalStmt {Γ : Context} (σ : State Γ) :  
  Stmt Γ → State Γ → Set where  
  eval-assign : ∀ {t expr val} →  
    EvalExpr σ expr val → (x : t ∈ Γ) →  
    EvalStmt σ (assign x expr) (update-state x σ val)  
  eval-if-false : ∀ {cond else σ'} →  
    EvalExpr σ cond false →  
    EvalStmts σ else σ' →  
    (then : List (Stmt Γ)) →  
    EvalStmt σ (if cond then else) σ'  
  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'  
  eval-while-false : ∀ {cond} →  
    EvalExpr σ cond false →  
    (body : List (Stmt Γ)) →  
    EvalStmt σ (while cond body) σ  
  eval-while-true : ∀ {cond body σ' σ''} →  
    EvalExpr σ cond true →  
    EvalStmts σ body σ' →  
    EvalStmt σ' (while cond body) σ'' →  
    EvalStmt σ (while cond body) σ''  
  
data EvalStmts {Γ : Context} (σ : State Γ) :  
  List (Stmt Γ) → State Γ → Set where  
  eval-skip :  
    EvalStmts σ [] σ  
  eval-seq : ∀ {stmt stmts σ' σ''} →  
    EvalStmt σ stmt σ' →  
    EvalStmts σ' stmts σ'' →  
    EvalStmts σ (stmt :: stmts) σ''
```

Figure 4.6: Semantics of statements.

5 Control-Flow Graphs

In this chapter we explain control-flow graphs and define an intrinsically typed representation for them.

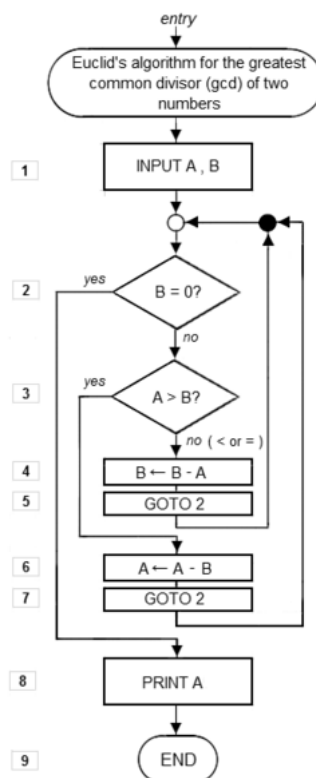


Figure 5.1: Control-flow graph of euclidean algorithm.
http://commons.wikimedia.org/wiki/File:Euclid_flowchart_1.png

A control-flow graph is a possible intermediate representation of a program during the compilation from a high level programming language to machine code. Figure 5.1 shows an example flow graph for the euclidean algorithm which computes the greatest common divisor of two numbers. A control-flow graph is usually a graph whose nodes are called basic blocks and consist of assembly-like instructions that do not contain any jumps and whose edges are jumps between these basic blocks. Unlike in assembly programs where one can set labels¹ at any position in the program and use them as targets of jumps, in a control-flow graph a label can only be defined at the beginning of a basic block and conditional and unconditional jumps can only occur at the end of basic blocks.

5.1 Label Declarations

In this thesis we define a special representation of control-flow graphs. Its basic idea is to use de Bruijn indices to represent labels. We do not explicitly represent the basic blocks and edges between them. Instead we use two binding constructs that are familiar in functional programming languages:

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

In the first example the `let`-construct is used to give the subexpression `fib 5` the new name `res` which is then used in the second subexpression `res + res`. The second example shows the definition of the mathematical factorial function ($n!$). The `fix`-construct is applied to a lambda function that takes a function argument `fac` and gives an implementation of the factorial function where `fac` is used for the recursive function call.²

In our representation of flow graphs we do not use these `let` and `fix`-constructs to introduce new variables like in the examples above but to introduce labels like depicted in Figure 5.2. The `let`-construct on the left-hand side specifies two sub-flow graphs that both contain some instructions. The first sub-flow graph is associated to a label `l`. This label can be used in the second sub-flow graph as target for jumps. The execution of the entire `let`-flow graph starts with the execution of the second sub-flow graph, the first sub-flow graph gets only executed if a jump to it is taken during the execution of the second sub-flow graph. The `fix`-construct on the right-hand side specifies one sub-flow graph which can contain

¹Labels are symbolic names that are translated to memory addresses by an assembler or linker.

²The exact understanding of this definition is not necessary for the purpose of this thesis.

jumps to itself. Thus it gives some kind of cyclic flow graphs. We will use it in Chapter 6 in the translation of while loops. The purpose of the let-construct will also be explained in that chapter.

let l:	
Instr A;	
Instr B;	
...	
in	fix l:
Instr X;	Instr 1;
Instr Y;	Instr 2;
...[jumps to l]...	...[jumps to l]...

Figure 5.2: Let and fix.

Figure 5.3 shows how these let and fix-constructs would be written in an assembly-like language. Remark that in the let-construct on the left hand side the order of the two instruction blocks is reversed compared to Figure 5.2. Thus its execution starts now with the first line. We will use this assembly-like language style in the flow graph examples in Section 5.5.

Instr X;	
Instr Y;	
...[jumps to l]...	
l: Instr A;	l: Instr 1;
Instr B;	Instr 2;
...	...[jumps to l]...

Figure 5.3: Let and fix in assembly language.

5.2 Jumps

In our flow graph representation, (sub-) flow graphs consists of a number of label declarations, jump-free instructions (see Section 5.4) and end with one of the following

instructions³⁴:

- `stop`
- `goto l`
- `branch l1 l2`

A `stop` instruction marks the end of a (sub-) flow graph that does not jump to some other sub-flow graph but terminates. A `goto` instruction represents an unconditional jump and specifies one label that has to be declared before by a `let` or `fix`-construct. The execution of a sub-flow graph that ends with a `goto` instruction continues with the sub-flow graph defined in that `let` resp. `fix`-construct. Finally we do not have a conditional jump that does or does not jump to a certain label. Instead we have a `branch` instruction that specifies two labels. It is only executable if the top most element of the stack is a boolean value. The execution of this instruction starts by taking that boolean value from the stack. If the boolean value is true, the execution continues with the sub-flow graph that was introduced with the first label. If it is false, the execution continues with the sub-flow graph that was introduced with the second label. The intrinsic typing will prevent flow graphs with branch instructions at positions where the stack is empty or has a number as top element.

5.3 Memory Model

Instructions in a flow graph can access two memories: the environment and the stack. The environment is similar to the state in the while language that we defined in Section 4.2. It is a fixed length memory where data can be read from and stored to. The stack is a growing data structure where instructions can only fetch the top element or put data on top of the stack. We use lists of types in Figure 5.4 to represent the types of the elements in the environment as well as the elements on the stack. In the following we will refer to them also as “environment types” and “stack types”.

Environments and stacks are defined in Figure 5.5 according to these environment and stack types. An environment that stores two integers has the Agda type `Env (num :: num :: [])`. Similarly `Stack (num :: bool :: [])` is the Agda type of stacks with one integer and one boolean value where the integer is the top element of the stack. According to the

³*l*, *l1* and *l2* stand for three labels.

⁴These three instructions are the base cases in the inductive definition of our flow graph representation (Figure 5.8) in Section 5.5.1.

definitions in Figure 5.5, `Env (num :: num :: bool :: [])` is the same as $\mathbb{Z} \times \mathbb{Z} \times \text{Bool} \times \top$. Thus an example environment of this Agda type would be `(+ 1 , - + 2 , false , tt)` where `tt` is of Agda type `Env []`⁵ and represents the empty environment. Analogously `tt` is also used to represent the empty stack which is of Agda type `Stack []`. Putting the element `+ 3` on top of a stack `st` of Agda type `Stack stack-types` results in the new stack `(+ 3 , st)` of Agda type `Stack (num :: stack-types)`.

```

EnvTypes = List Type
StackTypes = List Type

```

Figure 5.4: Environment types and stack types.

```

Env : EnvTypes → Set
Env [] = ⊤
Env (type :: env-types) = Val type × Env env-types

Stack : StackTypes → Set
Stack [] = ⊤
Stack (type :: stack-types) = Val type × Stack stack-types

```

Figure 5.5: Environments and stacks.

5.4 Jump-free Instructions

Data type `JFInstr` in Figure 5.6 enumerates all jump-free instructions, that are instructions that do not cause jumps. Since we use intrinsically typed syntax, the data type `JFInstr` has one parameter `env-types` of Agda type `EnvTypes` which represents the types of the elements in the environment before starting the instruction, as well as two indices of Agda type `StackTypes` which represent the types of the elements in the stack before and after the execution of the instruction. Thus `JFInstr env-types stack-types stack-types'` is the Agda type of jump-free instructions that can be executed when the elements in the environment have the types `env-types` and the elements on the stack have the types

⁵`Env []` is by definition equal to `⊤`.

stack-types before the execution resp. stack-types' after the execution ⁶. The jump-free instructions comprise loading from and storing to memory (**load**, **store**), pushing a constant on the stack (**push**), popping the top stack element (**pop**) as well as four numeric instructions (**add**, **mul**, **eq**, **le**) which coincide with the four numeric built-in functions of the while language.

```

data JFInstr (env-types : EnvTypes) : StackTypes → StackTypes → Set where
  load : ∀ {type stack-types} →
    type ∈ env-types →
    JFInstr env-types stack-types (type :: stack-types)
  store : ∀ {type stack-types} →
    type ∈ env-types →
    JFInstr env-types (type :: stack-types) stack-types
  push : ∀ {type stack-types} →
    Val type →
    JFInstr env-types stack-types (type :: stack-types)
  pop : ∀ {type stack-types} →
    JFInstr env-types (type :: stack-types) stack-types
  add : ∀ {stack-types : StackTypes} →
    JFInstr env-types (num :: num :: stack-types) (num :: stack-types)
  mul : ∀ {stack-types : StackTypes} →
    JFInstr env-types (num :: num :: stack-types) (num :: stack-types)
  eq : ∀ {stack-types : StackTypes} →
    JFInstr env-types (num :: num :: stack-types) (bool :: stack-types)
  le : ∀ {stack-types : StackTypes} →
    JFInstr env-types (num :: num :: stack-types) (bool :: stack-types)

```

Figure 5.6: Syntax of jump-free instructions.

The semantics of jump-free instructions is given by **EvalJFInstr** in Figure 5.7. It defines the judgment

$$\text{EvalJFInstr } \text{env } \text{st } \text{instr } \text{env}' \text{ st}'$$

which expresses that the execution of the jump free instruction **instr** in an environment **env** and a stack **st** results in an environment **env'** and a stack **st'**.

⁶The knowledge about the stack types after the execution is necessary to define the constructor **jf-instr** in the definition of the intrinsically typed syntax of flow graphs in Figure 5.8.

The load and store instructions (`load cell`), (`store cell`) have a reference to an environment cell `cell` as argument and put the content of the referenced environment cell on the top of the stack resp. discard the top element of the stack and store it in the environment (cf. `eval-load` resp. `eval-store`). As a consequence of the intrinsic typing, storing is only applicable if the type of the specified environment cell coincides with the type of the top element of the stack. The instruction (`push val`) has one literal `val` as argument and pushes it on top of the stack (cf. `eval-push`), the `pop` instruction just discards the topmost element from the stack (cf. `eval-pop`). The arithmetical instructions `add`, `mul`, `eq` and `le` have no arguments but discard the two topmost elements from the stack, compute the respective arithmetical operation on them and push the result back on the stack (cf. `eval-add`). The intrinsic typing prevents their application if the two top elements on the stack are not of type `num`.

```

data EvalJFInstr {env-types : EnvTypes} (env : Env env-types) :
  {stack-types stack-types' : StackTypes} → Stack stack-types →
  JFInstr env-types stack-types stack-types' →
  Env env-types → Stack stack-types' → Set where
eval-load : ∀ {stack-types type}
  (st : Stack stack-types) (cell : type ∈ env-types) →
  EvalJFInstr env st (load cell) env (env-load cell env , st)
eval-store : ∀ {stack-types type}
  (st : Stack stack-types) (cell : type ∈ env-types) (val : Val type) →
  EvalJFInstr env (val , st) (store cell) (env-store cell env val) st
eval-push : ∀ {stack-types type}
  (st : Stack stack-types) (val : Val type) →
  EvalJFInstr env st (push val) env (val , st)
eval-pop : ∀ {stack-types type}
  (st : Stack stack-types) (val : Val type) →
  EvalJFInstr env (val , st) pop env st
eval-add : ∀ {stack-types}
  (st : Stack stack-types) (n m : Val num) →
  EvalJFInstr env (m , (n , st)) add env (n + m , st)
  ...

```

Figure 5.7: Semantics of jump-free instructions.

5.5 Flow Graph Representation

In this section we define an intrinsically typed syntax as well as a big-step semantics for our control-flow graph representation.

5.5.1 Syntax

The flow graphs are defined by the inductive definition of `FlowGraph` in Figure 5.8. Its first parameter `labels` is a list that saves all the labels that the given (sub-) flow graph can jump to. As explained before, the basic idea of our flow graph representation is the use of de Bruijn indices to represent labels. In our intrinsically typed syntax, a label is not just a number. Instead a label is specified by the list of types of all elements on the stack, i.e. a label has the Agda type `StackTypes`. Accordingly the parameter `labels` has the Agda type `List StackTypes`. This is similar like in Section 3.2 where the Agda type of lambda terms in Figure 3.3 depends on a context.

Additionally the Agda type of flow graphs also depends on the types of all elements both in the environment and on the stack when starting the execution of the flow graph. Hence `FlowGraph` has in addition a parameter `env-types` of Agda type `EnvTypes` as well as an index of Agda type `StackTypes`. In sum, `FlowGraph labels env-types stack-types` is the Agda type of (sub-) flow graphs that can be executed in an environment with types `env-types` and a stack with elements of types `stack-types` and that can jump to labels that occur in the list `labels`.

For any given sub-flow graph the labels in the list `labels` are not declared in the sub-flow graph itself. Instead they are introduced by `let` and `fix`-constructs that occur anywhere outside the sub-flow graph in the overall flow graph. For this reason, the overall flow graphs are always defined relative to an empty list of labels. Hence they have the Agda type `FlowGraph [] env-types stack-types` for some type lists `env-types` and `stack-types`. See also the example flow graphs in Example 5.1ff below in this subsection.

The inductive definition of `FlowGraph` has three base flow graphs: `stop`, `goto` and `branch` that do not contain any sub-flow graphs and represent the end of a flow graph, an unconditional jump, resp. a branching (see Section 5.2). A `goto` specifies the label of the jump destination. It has two arguments: the implicit argument `stack-types` of Agda type `StackTypes` and an explicit argument of Agda type `stack-types ∈ labels` that ensures that the label actually occurs in the list of labels. Since the first argument is implicit, the

```

data FlowGraph (labels : List StackTypes) (env-types : EnvTypes) :
  StackTypes → Set where
stop : ∀ {stack-types} →
  FlowGraph labels env-types stack-types
goto : ∀ {stack-types} →
  stack-types ∈ labels →
  FlowGraph labels env-types stack-types
branch : ∀ {stack-types} →
  stack-types ∈ labels →
  stack-types ∈ labels →
  FlowGraph labels env-types (bool :: stack-types)
jf-instr : ∀ {stack-types stack-types'} →
  JFInstr env-types stack-types stack-types' →
  FlowGraph labels env-types stack-types' →
  FlowGraph labels env-types stack-types
fglet : ∀ {stack-types stack-types'} →
  FlowGraph labels env-types stack-types' →
  FlowGraph (stack-types' :: labels) env-types stack-types →
  FlowGraph labels env-types stack-types
fgfix : ∀ {stack-types} →
  FlowGraph (stack-types :: labels) env-types stack-types →
  FlowGraph labels env-types stack-types

```

Figure 5.8: Syntax of flow graphs.

argument l in `(goto l)` belongs to the second argument, not the first. A `branch` instruction specifies two labels for two jump destinations. However it has only one argument `stack-types` of Agda type `StackTypes` rather than two as well as two arguments of Agda type `stack-types ∈ labels`. The argument `stack-types` can occur at different positions in the list `labels`. Like explained in Section 3.2, the two arguments of Agda type `stack-types ∈ labels` contain the information to which exact label in `labels` they belong to. In that way they can represent two possibly distinct labels. The arguments l and l' in `(branch l l')` belong to these two arguments of Agda type `stack-types ∈ labels`. The intrinsic typing assures that the top most element on the stack is a boolean value. Dependent on whether this element is true or false, the flow graph execution jumps to the first or the second label. The constructor `jf-instr` prepends a jump-free instruction to a given flow graph. By

the repeated use of this constructor one can define a flow graph that comprises a list of jump-free instructions and that ends with a `stop`, `goto` or `branch` (see Example 5.1). The constructors `fglet` and `fgfix` represent the `let` and `fix`-constructs⁷ that were explained in Section 5.1. The constructor `fglet` contains two sub-flow graphs. The second one can contain jumps to the first one, i.e. the second sub-flow graph has an additional label `stack-types'` which has to match the types of the elements of the start stack of the first sub-flow graph. This statically ensures that jumps to the first sub-flow graph can only occur in positions where the types of the elements on the stack match the ones that are required to start the first sub-flow graph. The constructor `fgfix` has one sub-flow graph that can jump to itself, i.e. the sub-flow graph has an additional label `stack-types` which matches its own start stack types. As a consequence, at the start of each loop iteration the stack must have the same form, i.e. it must have the same size and its elements must have the same types. Hence loops with growing stacks are not expressible (see Example 5.4).

The intrinsically typed syntax of the flow graph representation gives us some assurances about the stack, the environment and jumps. A jump to a sub-flow graph is only possible if that sub-flow graph requires a stack that matches the types of the current stack. That means the type system statically prevents from jumps to a sub-flow graph that requires a different stack, i.e. a different stack size or different types of the elements on the stack. Thus it prohibits for example a jump from a position where the stack is empty to a sub-flow graph that starts with an addition instruction and hence requires two numbers as top elements on the stack. The size of the environment as well as the types of its elements are fixed for a given program, that means the size of the environment is constant during the program execution and data can be stored in the environment only if it has the right type, i.e. it is not possible to overwrite a boolean value in the environment with an integer or vice versa.

We will now give some example flow graphs in this flow graph representation.

Example 5.1

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

⁷The name “let” is reserved for let-expressions in Agda. For this reason we named the constructors `fglet` and `fgfix`.

In this example the values of two cells in the environment are swapped by first loading the content of the first cell on top of the stack and loading the content of the second cell on top of the resulting stack and then storing the topmost element of the stack in the first cell and thereafter storing the remaining element in the second cell. Assuming that the flow graph gets started with an environment that contains exactly two integers and the empty stack⁸ this flow graph is represented in Agda as:

```

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))))

```

The flow graph is represented by a cascade of **jf-instr** constructors that contain the jump-free **load** and **store** instructions and ends with a **stop** flow graph. The first argument of **FlowGraph** in the declaration of **exmpl1** is the empty list of labels [], the second argument is the list **num :: num :: []** since the environment saves two integers and the third argument is the empty list of types [] since we want to start the flow graph with the empty stack.

Example 5.2

```

  branch l2 l1
l2:push 3
  stop
l1:stop

```

This flow graph example assumes a stack with one boolean value. In the execution of the flow graph that boolean value is taken from the stack. If it is true the execution of the flow graph jumps to label l2 and pushes the number 3 on the stack before it stops. If it is false the execution jumps to label l1 and stops remaining with an empty stack. In our flow graph representations it looks as follows:

⁸One could also choose to start this flow graph with an environment with more than two cells and with a non-empty stack but for simplicity we chose the smallest possible environments and stacks in all of the examples.

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

We need to use two let-constructs to define the two sub-flow graphs to which the branching can jump to. Like explained above in this subsection, overall flow graphs are always defined relative to an empty list of labels. Hence the first argument of `FlowGraph` is again `[]`. Its second argument is the empty list of types `[]` since the flow graph does not access the environment and we start it with the empty environment. We also start it with exact one boolean value on the stack. Thus the third argument is `bool :: []`. This example also shows that it is possible in our representation to have a branching where the execution of the branches results in different stacks. The execution of the flow graph results either in a stack with exact one number or in the empty stack.

Example 5.3

```
l: goto l
```

This endless loop is represented as:

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

Example 5.4

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

This endless loop where twos are pushed on top of the stack ad infinitum is not expressible in our flow graph representation since the fix-construct allows only jumps back when the

stack form does not change during one iteration, i.e. the stacks at the begin of each loop iteration need to coincide in length and types of its elements.

Nevertheless the stack at the end of the last iteration may differ like in the following example where the loop is left after the first iteration.

Example 5.5

```
l: push 2
  stop
```

Here we modify the last example by replacing the `goto` by a `stop` instruction. Hence we exit the loop by stopping rather than looping back. The flow graph representations is as follows:

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

Example 5.6

```
l2:load x
  branch l1 l2
l1:stop
```

In this final example the content of a boolean cell is loaded from the environment. If its content is true the branching loops back resulting in a non-terminating endless loop. If it is false the execution quits the loop and stops. The flow graph representation is the following:

```

exmpl6 : FlowGraph [] (bool :: []) []
exmpl6 =
  fglet stop
    (fgfix
      (jf-instr (load (here refl))
        (branch (here refl) (there (here refl))))))

```

5.5.2 Semantics

In order to execute a flow graph, one needs to keep track of the sub-flow graphs that are defined by let and fix-constructs, to be able to execute the right sub-flow graph when a jump occurs. For this purpose we define the Agda function `flowgraphs-for-labels` in Figure 5.9. It relates the list of labels to the list of corresponding sub-flow graphs.

```

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

```

Figure 5.9: Flow graphs for labels.

The big-step semantics of flow graphs is now given by the inductive definition `EvalFlowGraph` in Figure 5.10. The judgment

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

states that the execution of a flow graph `fg` with an environment `env` and a stack `st` results in an environment `env'` and a stack `st'` whereby the flow graphs in `fgs` serve as jump destinations for labels that occur in the list `labels`, which is an implicit parameter of `EvalFlowGraph`. The previously defined function `flowgraphs-for-labels` is used to determines the Agda type of `fgs` for the given labels `labels`.

```

data EvalFlowGraph {labels : List StackTypes} {env-types : EnvTypes}
  (fgs : flowgraphs-for-labels env-types labels) (env : Env env-types) :
  {stack-types stack-types' : StackTypes} → Stack stack-types →
  (fg : FlowGraph labels env-types stack-types) →
  Env env-types → Stack stack-types' → Set where
eval-stop : ∀ {stack-types} (st : Stack stack-types) →
  EvalFlowGraph fgs env st stop env st
eval-goto : ∀ {stack-types stack-types' env' l}
  {st : Stack stack-types} {st' : Stack stack-types'} →
  EvalFlowGraph (rest-fgs fgs l) env st (access-fg fgs l) env' st' →
  EvalFlowGraph fgs env st (goto l) env' st'
eval-branch-true : ∀ {stack-types stack-types' env' l l'}
  {st : Stack stack-types} {st' : Stack stack-types'} →
  EvalFlowGraph (rest-fgs fgs l) env st (access-fg fgs l) env' st' →
  EvalFlowGraph fgs env (true , st) (branch l l') env' st'
eval-branch-false : ∀ {stack-types stack-types' env' l l'}
  {st : Stack stack-types} {st' : Stack stack-types'} →
  EvalFlowGraph (rest-fgs fgs l') env st (access-fg fgs l') env' st' →
  EvalFlowGraph fgs env (false , st) (branch l l') env' st'
eval-jf-instr : ∀ {stack-types stack-types' stack-types'' env' env''}
  {instr : JFInstr env-types stack-types stack-types'}
  {fg : FlowGraph labels env-types stack-types'}
  {st : Stack stack-types} {st' : Stack stack-types'} {st'' : Stack stack-types''} →
  EvalJFInstr env st instr env' st' →
  EvalFlowGraph fgs env' st' fg env'' st'' →
  EvalFlowGraph fgs env st (jf-instr instr fg) env'' st''
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''
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'

```

Figure 5.10: Semantics of flow graphs.

The flow graph `stop` represents the end of a flow graph and its execution has no effect on the stack or the environment (cf. `eval-stop` in Figure 5.10). To define the semantics of unconditional jumps and branchings we use the helper function `access-fg`. (`access-fg fgs l`) looks up in the list of flow graphs `fgs` the sub-flow graph that corresponds to the label `l`. This sub-flow graph can have a different list of labels than the flow graph itself and hence a different list of corresponding sub-flow graphs. The helper function `rest-fgs` has the same parameters than `access-fg` and returns the sub-flow graphs of the sub-flow graph returned by `access-fg`. The execution of (`goto l`) is now described by the execution of the corresponding sub-flow graph to label `l` which is looked up with the help of `access-fg`. (cf. `eval-goto`). To execute a branching (`branch l l'`) the top element is taken from the stack. By definition of branchings in Figure 5.8 this has to be a boolean value. If it is true, the execution of the branching is described by the execution of the sub-flow graph corresponding to label `l` (cf. `eval-branch-true`), if it is false, it is described by the execution of the sub-flow graph corresponding to label `l'` (cf. `eval-branch-false`). The execution of (`jf-instr instr fg`) is described by first executing the instruction `instr` and then executing the rest flow graph `fg` in the resulting immediate state (cf. `eval-jf-instr`). The execution of (`fglet fg fg'`) is described by the execution of `fg'` under the presence of an additional label and an additional corresponding flow graph `fg` (cf. `eval-let`). The execution of (`fgfix fg`) is described by the execution of `fg` under the presence of (`fgfix fg`) itself as an additional sub-flow graph (cf. `eval-fix`). Remark that we allow the execution of `fg` to change the form of the stack, i.e. the size of the stack or the types of the elements on the stack. Hence `st'` has Agda type `Stack stack-types'` rather than `Stack stack-types`. This is necessary in order to execute fix-constructs when they do not loop back to themselves but exit this self-calling loop but change the stack before like in the execution of Example 5.5 in Example 5.10.

In the following we give some examples where we state in which result states some of the flow graph examples from the last subsection terminate and we show how those statements are proven according to the big-step semantics of flow graphs.

Example 5.7 The execution of Example 5.1 with the environment `(+ 2 , + 3 , tt)` results in an environment where the two cell contents are swapped, i.e. in `(+ 3 , + 2 , tt)`. This can be proven in Agda with the following Agda function:

```

exmpl1-exe : EvalFlowGraph tt (+ 2 , + 3 , tt) tt exmpl1 (+ 3 , + 2 , tt) tt
exmpl1-exe =
  eval-jf-instr (eval-load _ _)
    (eval-jf-instr (eval-load _ _)
      (eval-jf-instr (eval-store _ _ _)
        (eval-jf-instr (eval-store _ _ _)
          (eval-stop _))))))

```

The proof is done simply by applying the semantics definitions of load and store instructions (`eval-load` resp. `eval-store`).

Example 5.8 The execution of Example 5.2 with a stack with element `true`, i.e. `(true , tt)` results in a stack `(+ 3 , tt)`. First the branch instruction discards `true` from the stack and jumps to the first label. There `+ 3` is pushed on the stack before the execution stops. This is proven in the following Agda function:

```

exmpl2-exe-true : EvalFlowGraph tt tt (true , tt) exmpl2 tt (+ 3 , tt)
exmpl2-exe-true =
  eval-let
    (eval-let
      (eval-branch-true
        (eval-jf-instr (eval-push _ _)
          (eval-stop _))))))

```

The proof is done by applying the semantics definitions of the executed instructions, namely those of the two lets (`eval-let`), the branch on `true` (`eval-branch-true`), the push instruction (`eval-push`) and the stop instruction (`eval-stop`).

Example 5.9 The endless loop in Example 5.3 does not terminate. This can be proven with the following Agda function:



```
exmpl3-exe :  $\forall$  env {stack-types} (st : Stack stack-types)  $\rightarrow$   
   $\neg$  EvalFlowGraph tt tt tt exmpl3 env st  
exmpl3-exe env st (eval-fix (eval-goto exe)) =  
  exmpl3-exe env st exe
```

Since big-step semantics relate initial to final program states, we show the non-termination by proving that no state, consisting of any environment `env` and stack `st` is reachable from the initial state, i.e. the empty environment `tt` and the empty stack `tt`. We do the proof by first unfolding the loop body one time and then applying the induction hypothesis in form of a recursive call of `exmpl3-exe`.

Example 5.10 The execution of Example 5.5 starting with the empty stack `tt` results in the stack `(+ 2 , tt)`. This is proven by

```
exmpl5-exe : EvalFlowGraph tt tt tt exmpl5 tt (+ 2 , tt)  
exmpl5-exe =  
  eval-fix  
    (eval-jf-instr  
      (eval-push _ _)  
      (eval-stop _))
```

6 Translation

In this chapter we define the translator from the while language defined in Chapter 4 to the control-flow graph representation defined in Chapter 5 and prove its correctness.

6.1 Implementation

Since while programs are just lists of statements we are going to define the translator as an Agda function `translate` with the following declaration:

$$\text{translate} : \{\Gamma : \text{Context}\} \rightarrow \text{List (Stmt } \Gamma) \rightarrow \text{FlowGraph } [] \Gamma []$$

It takes a list of statements as argument and returns a flow graph which depends on an empty list of labels (first `[]` in the above declaration), since there are no jumps out of the entire program flow graph. We always start the execution of a translated program with an empty stack. Hence the result flow graph is defined relative to an empty list of stack types (second `[]`). As environment we take the state of the while language. The environment types are therefore given by the context Γ .

We define the translation of expressions in Figure 6.1 as well as the translation of statements and statements lists in Figure 6.4 by continuation-passing style. That means the functions `translateExpr`, `translateStmt` and `translateStmts` have an additional flow graph `fg` as argument that represents the rest of the computation also called continuation.

The first explicit argument of `translateExpr` in Figure 6.1 is an expression `expr` which has type `t` in context Γ and thus has the Agda type `Expr Γ t`. The second one is a continuation of Agda type `FlowGraph labels Γ (t :: stack-types)` which represents further computations¹ after the translation of the expression. The further translation may require the static

¹In our case further computations means further translations.

```

translateExpr : {Γ : Context} {t : Type} (expr : Expr Γ t)
  {labels : List StackTypes} {stack-types : StackTypes} →
  FlowGraph labels Γ (t :: stack-types) →
  FlowGraph labels Γ stack-types

translateExpr (var x) fg = jf-instr (load x) fg

translateExpr (lit val) fg = jf-instr (push val) fg

translateExpr (add e e') fg =
  translateExpr e (translateExpr e' (jf-instr add fg))

...

```

Figure 6.1: Translation of expressions with continuation-passing style.

information that the result of the expression evaluation is put on top of the stack. Consider e.g. the translation of `(add e e')`. The addition operator is translated into the corresponding addition instruction. According to the intrinsically typed syntax of addition (cf. Figure 5.6), this instruction can be inserted into a flow graph only at a position where the two top elements of the stack are numbers. In general the continuation `fg` is a flow graph which starts in a stack where the type `t` of the expression is added to the stack type, i.e. the Agda type of the continuation of `translateExpr` is `FlowGraph labels Γ (t :: stack-types)`.

The Agda function `translateExpr` is defined by pattern matching the expression. A Variable `(var x)` is translated to the instruction `(load x)`, a literal `(lit val)` to `(push val)`. An arithmetic operation with two operands `e` and `e'` is translated by translating the first operand `e` by a recursive function call of `translateExpr` and another recursive function call as continuation. The second recursive function call translates the second operand `e'` and takes the respective arithmetic instruction as continuation.

Figure 6.2 shows schematically how we use `let` and `fix`-constructs to translate conditionals and while loops into flow graphs. A conditional `if b then S1 else S2` is translated like in the following (Figure 6.2, left): by a cascade of `lets` we define a label `l0` for the continuation and labels `l1` and `l2` for the translation of the then branch `S1` resp. of the else branch `S2`². The translation of both branches is followed by a jump to the continuation. The main part

²In Figure 6.2 translated statements `S` and expressions `e` are denoted by `[[S]]` resp. `[[e]]`.

of the flow graph is now the translation of the boolean condition b of the if statement whose execution results in a boolean value that is put on top of the stack and which is used to determine whether the branching jumps to the translated then or the translated else branch. The translation of the while loop `while b do S` (Figure 6.2, right) also starts with the definition of a label l_0 for the continuation. The following fix-construct allows us to define a label l_1 that we can use inside of the subsequent sub-flow graph. There we use another let to define a third label l_2 for the translation of the loop body. The main part of the flow graph starts with the translation of the boolean loop condition b . Depending on the resulting boolean value during execution the branching either jumps to the continuation and thereby exits the loop or jumps to the translated loop body which ends by a jump back to the translated boolean expression. Figure 6.3 shows the same translations in assembly-like style.

<pre> let l_0: continuation in let l_1: $\llbracket S_1 \rrbracket$ goto l_0 in let l_2: $\llbracket S_2 \rrbracket$ goto l_0 in $\llbracket b \rrbracket$ branch l_1 l_2 </pre>	<pre> let l_0: continuation in fix l_1: let l_2: $\llbracket S \rrbracket$ goto l_1 in $\llbracket b \rrbracket$ branch l_2 l_0 </pre>
--	--

Figure 6.2: Translation of `if b then S_1 else S_2` and `while b do S` .

<pre> $\llbracket b \rrbracket$ branch l_1 l_2 l_2: $\llbracket S_2 \rrbracket$ goto l_0 l_1: $\llbracket S_1 \rrbracket$ goto l_0 l_0: continuation </pre>	<pre> l_1: $\llbracket b \rrbracket$ branch l_2 l_0 l_2: $\llbracket S \rrbracket$ goto l_1 l_0: continuation </pre>
---	--

Figure 6.3: Translation of `if b then S_1 else S_2` and `while b do S` in assembly style.

We need to define the translation of statements `translateStmt` and the translation of lists of statements `translateStmts` (both in Figure 6.4) mutually recursive since they depend on each other: the then and else branches of a single if statement as well as the body of a single while statement are itself lists of statements.

The execution of translated statements does not change the stack. The stack grows and shrinks only during the execution of translated expressions. For this reason we define the result as well as the continuation of both functions `translateStmt` and `translateStmts` as flow graphs that start with the empty stack Γ , i.e. result and continuation have the Agda type `FlowGraph` labels Γ .

The Agda function `translateStmt` in Figure 6.4 is defined by pattern matching the statement. An assignment (`assign x expr`) is translated by the translation of the assigned expression `expr` with the instruction (`store x`) prepended to the continuation. If and while statements are translated like explained before and depicted in Figure 6.2. The Agda function `translateStmts` is defined by pattern matching the lists of statement. An empty list of statements Γ is translated by just returning the continuation `fg`; the translation of a non-empty list of statements `stmt :: stmts` is given by translating the first statement `stmt` with the translation of the remaining statements `stmts` as continuation.

Finally in Figure 6.5 we use `translateStmts` to implement the translator. We want the translator to finish after the translation of the program `prog`. Thus we use empty flow graph `stop` as continuation for `translateStmts`.

Example 6.1 To give a fully fledged example of a translation we give in Figure 6.6 the translation of the factorial program from Example 4.2. Figure 6.7 and Figure 6.8 sketches the same flow graph in a schematic version resp. in an assembly-like style. Remark that labels are represented by de Bruijn indices and that, like explained in Section 3.1, the label n refers to the n -th innermost binder, i.e. the n -th innermost `let` or `fix`-construct. Thus the occurrence of `here refl` in the `branch` instruction in the last line of Figure 6.6 refers to label l2 in Figure 6.7 and Figure 6.8 whereas its occurrence in the `goto` instruction four lines above refers to the label l1, since the label l2 is not available at that position.

```

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

translateStmts : {Γ : Context} → List (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)))))))

translateStmts [] fg = fg

translateStmts (stmt :: stmts) fg = translateStmt stmt (translateStmts stmts fg)

```

Figure 6.4: Translation of statements with continuation-passing style.

```

translate : {Γ : Context} → List (Stmt Γ) → FlowGraph [] Γ []
translate prog = translateStmts prog stop

```

Figure 6.5: Translator.

```

jf-instr (push (+ 1))
  (jf-instr (store (there (here refl)))
    (fglet
      stop
      (fgfix
        (fglet
          (jf-instr (load (there (here refl))))
          (jf-instr (load (here refl)))
          (jf-instr mul
            (jf-instr (store (there (here refl))))
            (jf-instr (load (here refl)))
            (jf-instr (push (- + 1))
              (jf-instr add
                (jf-instr (store (here refl)))
                (goto (here refl))))))))))
  (jf-instr (push (+ 1))
    (jf-instr (load (here refl))
      (jf-instr le
        (branch (here refl) (there (there (here refl))))))))

```

Figure 6.6: Translated factorial program.

```
push +1
store r
let l0:
  stop
in
  fix l1:
    let l2:
      load r
      load i
      mul
      store r
      load i
      push -1
      add
      store i
      goto l1
    in
      push +1
      load i
      le
      branch l2 l0
```

Figure 6.7: Translated factorial program - schematic.

```
        push +1
        store r
l1:      push +1
        load i
        le
        branch l2 l0
l2:      load r
        load i
        mul
        store r
        load i
        push -1
        add
        store i
        goto l1
l0:      stop
```

Figure 6.8: Translated factorial program - assembly style.

6.2 Correctness

In this section we prove the correctness of the translator that we have defined in the last section. The correctness statement that we are going to prove is the following: If a program executed in a state σ results in a state σ' according to the semantics of the while language, then also the execution of the translated program in state σ results in state σ' according to the semantics of flow graphs.

The Agda formalization of this statement is as follows:

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

The logical statement $\text{EvalStmts } \sigma \ \text{prog } \sigma'$ expresses that the program prog executed in initial state σ terminates in state σ' . The translation of the program prog is given by translate prog . A simple function translateState is used to map the programming language states σ and σ' to environments which contain the values of all variables in the respective state³. Like described in the last section we start the execution of the translated programs with the empty stack (second tt in the above statement). By proving the above statement we show that the translated program also finishes with the empty stack (third tt). Like explained in Section 5.5 the overall flow graph resulted from the translation is defined relative to an empty list of labels. Hence the execution of the translation does not rely on any flow-graphs (first tt).

We will finally prove the above proposition in Figure 6.13. In order to do so we prove that the translations of expressions in Figure 6.1 and the translation of statements and statement lists in Figure 6.4 is correct. We do this by defining the functions $\text{translateExpr-correct}$ as well as $\text{translateStmt-correct}$ and $\text{translateStmts-correct}$ and we again use continuation passing style to define them.

The correctness proof for the translation of expressions is given by the Agda function $\text{translateExpr-correct}$. In its declaration in Figure 6.9 we express what we are going to prove about translateExpr : executing the translation of an expression expr , i.e. executing $(\text{translateExpr expr fg})$, in the environment $(\text{translateState } \sigma)$ and stack st results in an environment env and a stack st' if, first, the expression expr is evaluated according to the

³States and environments are structurally equal. We could have used the definition of state again instead of defining environments and could then avoid this function.

language semantics in state σ to a value `val` and second, the execution of the continuation `fg` results in the same environment `env` and stack `st'` when putting `val` on top of the stack `st` before the execution of the continuation `fg`⁴.

```

translateExpr-correct :
  ∀ {Γ t expr σ labels stack-types stack-types' fg env}
  {fgs : flowgraphs-for-labels Γ labels}
  {st : Stack stack-types} {st' : Stack stack-types'}
  {val : Val t} → EvalExpr σ expr val →
  EvalFlowGraph fgs (translateState σ) (val , st) fg env st' →
  EvalFlowGraph fgs (translateState σ) st (translateExpr expr fg) env st'

```

Figure 6.9: Statement of expression translation correctness.

The correctness of `translateExpr` is proven in Figure 6.10 by induction on the derivation of the evaluation of `expr`, i.e. the function `translateExpr-correct` is defined by pattern matching on its first explicit argument which has the Agda type `EvalExpr σ expr val`. Its second explicit argument `eval-fg` of Agda type `EvalFlowGraph fgs (translateState σ) (val , st) fg env st'` is its continuation. The proof follows the implementation of `translateExpr` in Figure 6.1. In the individual cases we show that the evaluation of the translated expressions yield the same values than the expressions themselves. For example for literals we show that the instruction `(push val)` yields the same value, namely `val`, than `(lit val)`.

Since we have implemented the translation of statements and statement lists by two mutual recursive functions `translateStmnt` and `translateStmnts`, we also give their correctness proofs by two mutual recursive Agda functions `translateStmnt-correct` and `translateStmnts-correct`. The declaration of `translateStmnt-correct` in Figure 6.11 expresses what we are going to prove about `translateStmnt-correct`: the execution of a translated statement `stmnt`, i.e. the execution of `(translateStmnt stmnt fg)`, in environment `(translateState σ)` results in an environment `env` if first, the execution of the statement `stmnt` in state σ results in a state σ' according to the semantics of the while language and second, the execution of the continuation `fg` in environment `(translateState σ')` results in environment `env`. Analogously the function `translateStmnts-correct` states the same for lists of statements.

⁴Since the execution of expressions does not change the environment we could prove the more strict statement where the result environment `env` is equal to the start environment `(translateState σ)`. But we need the given more generally formulated statement in the subsequent correctness proofs for the translation of statements and statement lists.

```

translateExpr-correct {σ = σ'} (eval-var x) eval-fg
  rewrite lookup-var-to-env-load x σ =
  eval-jf-instr (eval-load _ x) eval-fg

translateExpr-correct (eval-lit val) eval-fg =
  eval-jf-instr (eval-push _ val) eval-fg

translateExpr-correct (eval-add eval-e eval-e') eval-fg =
  translateExpr-correct eval-e
  (translateExpr-correct eval-e'
   (eval-jf-instr (eval-add _ _ _) eval-fg))

...

```

Figure 6.10: Proof of expression translation correctness.

Since we need the stack only for the execution of expressions but not for the execution of statements, we only use the empty stack `tt` in the declarations of `translateStmt-correct` and `translateStmts-correct`.

The correctness of `translateStmt` and `translateStmts-correct` is proven in Figure 6.12 by mutual induction on the derivation of the evaluation of `stmt` resp. `stmts`, i.e. the functions `translateStmt-correct` and `translateStmts-correct` are defined mutually recursive. `translateStmt-correct` is defined by pattern matching on its first explicit argument which has the Agda type `EvalStmt σ stmt σ'`. Its second explicit argument `eval-fg` of Agda type `EvalFlowGraph fgs (translateState σ') tt fg env tt` is its continuation. Analogous for `translateStmts-correct`.

In the individual cases we show that the execution of a translated statement results in the same state than the execution of the statement itself. For example for a conditional (`if cond then else`) we have one case (namely when the first argument is given by the constructor `eval-if-false`), where the condition `cond` evaluates to `false` and the result state of the execution of the if statement is given by the result state of the execution of the else block. We can use the before proven correctness of expression translation, i.e. the before defined Agda function `translateExpr-correct`, to conclude that the translated condition also evaluates to `false`. And hence the branching in the translation of the if statement

```

translateStmt-correct :
  ∀ {Γ stmt σ σ' env labels fg}
  {fgs : flowgraphs-for-labels Γ labels} →
  EvalStmt σ stmt σ' →
  EvalFlowGraph fgs (translateState σ') tt fg env tt →
  EvalFlowGraph fgs (translateState σ) tt (translateStmt stmt fg) env tt

translateStmts-correct :
  ∀ {Γ stmts σ σ' env labels fg}
  {fgs : flowgraphs-for-labels Γ labels} →
  EvalStmts σ stmts σ' →
  EvalFlowGraph fgs (translateState σ') tt fg env tt →
  EvalFlowGraph fgs (translateState σ) tt (translateStmts stmts fg) env tt

```

Figure 6.11: Statement of statement translation correctness.

jumps to the translated else block which by induction hypothesis, i.e. by a mutual recursive call to `translateStmts-correct`, terminates in the same state than the else block itself. This shows that the execution of the translation of the if statement results in the same state than the execution of the if statement itself.

Finally we use `translateStmts-correct` in the definition of `translate-correct` in Figure 6.13 to prove the overall correctness of the translator. Since we have defined the translator by translating programs with the empty flow graph `stop` as continuation in Figure 6.5, we use now the evaluation of the empty flow graph (`eval-stop _`) as continuation for `translate-correct`.



```
translateStmt-correct  $\{\sigma = \sigma\}$  (eval-assign {val = val} eval-expr x) eval-fg
  rewrite update-state-to-env-store x  $\sigma$  val =
  translateExpr-correct eval-expr (eval-jf-instr (eval-store _ x val) eval-fg)

translateStmt-correct (eval-if-false eval-cond-false eval-else _) eval-fg =
  eval-let (eval-let (eval-let
    (translateExpr-correct eval-cond-false (eval-branch-false
      (translateStmts-correct eval-else (eval-goto eval-fg))))))

translateStmt-correct (eval-if-true eval-cond-true eval-then _) eval-fg =
  eval-let (eval-let (eval-let
    (translateExpr-correct eval-cond-true (eval-branch-true
      (translateStmts-correct eval-then (eval-goto eval-fg))))))

translateStmt-correct (eval-while-false eval-cond-false _) eval-fg =
  eval-let (eval-fix (eval-let
    (translateExpr-correct eval-cond-false (eval-branch-false eval-fg))))

translateStmt-correct (eval-while-true eval-cond-true eval-body eval-stmt) eval-fg
  with translateStmt-correct eval-stmt eval-fg
... | eval-let eval-let-body =
  eval-let (eval-fix (eval-let
    (translateExpr-correct eval-cond-true (eval-branch-true
      (translateStmts-correct eval-body (eval-goto eval-let-body))))))

translateStmts-correct eval-skip eval-fg = eval-fg

translateStmts-correct (eval-seq eval-stmt eval-stmts) eval-fg =
  translateStmt-correct eval-stmt (translateStmts-correct eval-stmts eval-fg)
```

Figure 6.12: Proof of statement translation correctness.

```

translate-correct :  $\forall \{\Gamma\} \{\text{prog} : \text{List } (\text{Stmt } \Gamma)\} \{\sigma \ \sigma'\} \rightarrow$ 
  EvalStmts  $\sigma$  prog  $\sigma' \rightarrow$ 
  EvalFlowGraph tt (translateState  $\sigma$ ) tt (translate prog) (translateState  $\sigma'$ ) tt
translate-correct eval-prog = translateStmts-correct eval-prog (eval-stop _)

```

Figure 6.13: Translator correctness.

7 Related Work

Compiler verification is an established field of research in computer science starting with a pen and paper proof of the correctness of a compiler for arithmetic expressions by McCarthy and Painter[32] in 1967 intended as a prototype for the correctness proofs of more complete compilers with the aim to computer check those proofs.

Compiler verification is not the only approach to ensure the trustworthy execution of machine code generated by a compiler, two others are translation validation and certifying compilers¹. In the translation validation[47] approach one does not verify the compiler, rather there is an additional component, the analyzer, that checks for a given source program whether the compiler output behaves the same than the source program. The analyzer does not need to be complete, i.e. it may fail to validate some correct translated programs, but ought to reject compiler outputs that do not behave like the source program or even provide a counterexample, e.g. an input for which the executions of source and of the translated program differ.

A certifying compiler on the other hand assures that the translated programs satisfy some functional property. The certifying compiler does not only compile a source program but also delivers additional data that is used by another component, the certifier, to construct a proof that the compiler output fulfills the desired property. Necula and Lee[38] present a certifying compiler that translates a type-safe subset of the C programming language into DEC Alpha assembly code while assuring type safety and memory safety of the output assembly code. Certifying compilers are also used in the proof-carrying code[37] technique where code from possibly untrusted sources is accompanied by some proof that it can be executed safely.

Recent projects in the field of compiler verification are CompCert[12] and CakeML[10]². CompCert[30, 29] is a research project led by Xavier Leroy that achieved to develop the

¹Leroy[29](Section 2) relates the different approaches.

²A general overview about compiler verification projects and related topics is given by Leinenbach[28].

verified optimizing compiler CompCert in the proof assistant Coq[54, 5]. The source language of the CompCert compiler, Clight[8], is a fairly large subset of the programming language C, supporting e.g. pointer arithmetic. Its target language, PCC, is a subset of PowerPC assembly language. The verified part of the CompCert compiler consists of 14 compiler passes that make use of 8 different intermediate representations, for each of them a formal semantics is given. Semantic preserving is proven for the single compiler passes in isolation, the overall correctness result is gained by the composition of the single proofs. Leroy justifies the use of a high number of intermediate representations by the ease of the verification. One of the intermediate representations is a register transfer language (RTL) which is used to perform constant propagation and common subexpression elimination optimizations based on data-flow analyses and as input format for register allocation. The speed of compiled programs is measured in a benchmark as being competitive to that of GCC 4.0.1 with optimization levels -O1 and -O2. Not verified are the parser, assembler and linker. To gain an executable of the compiler one needs to use Coq's extraction mechanism[31] to generate OCaml[42] code from the Coq development and compile this OCaml code with an OCaml compiler.

Kumar, Myreen et al.[24] describe the verification of a read-eval-print loop (REPL) of the programming language CakeML in the theorem prover HOL4[18]. CakeML is a subset of the strongly typed, impure³ functional programming language Standard ML[33]. Part of this REPL verification is a compiler verification which goes beyond the verification effort of CompCert in two concerns. First also parsing is verified and second it further reduces the trusted computing base by not relying on some unverified code extraction mechanism or additional compiler to execute the CakeML compiler. Instead the compiler is bootstrapped in the following way: Kumar, Myreen et al. use a technique called proof-producing synthesis[36] that gives for a given HOL4 function a CakeML program together with a proof that the execution of the CakeML program according to its semantics has the same result than the HOL4 function itself. They apply this technique to the CakeML compiler, which is implemented and verified in HOL4, to get the source code of the Compiler in CakeML and then apply the HOL4 variant of the compiler to the CakeML variant of it to yield a verified bytecode⁴ implementation of the compiler. This bytecode is further translated to x86-64 machine code and verified according to the small-step operational semantics of a subset of x86 machine code as defined in by Myreen [35](Section 6) which extends the definition of Susmit Sarkar et al.[49].

Another compiler verification was performed in the Verisoft[3] project which aimed at


³Standard ML functions can have side effects.

⁴an assembly language for a virtual stack machine

the pervasive formal verification of a computer system, i.e. the verification of an entire computer system stack from the gate level hardware to the application software level. The involved compiler translates C0, a subset of the C programming language, to an assembly language for the DLX[45, 34] instruction set architecture of the verified architecture microprocessor (VAMP)[23, 7, 6]. The microprocessor has a pipelined RISC architecture and was originally specified and verified to implement the DLX instruction set architecture in the interactive theorem prover PVS[43] and was ported to Isabelle/HOL[19, 41] in the Verisoft project. Following Chirica and Martin[11] as well as Goerigk and Hoffmann[16] the verification of the C0 compiler[27] is divided in two part: the verification of the compiler specification and the verification of the implementation. Leinenbach[28] used Isabelle/HOL to perform the first task. He gave a specification of the C0 compile in form of an executable HOL function, gave a small-step semantics of C0 and proved a “small-step simulation theorem” which expresses that any step in the execution of a source program according to the small-step C0 semantics is simulated by some steps of the compiled program according to the semantics of the Vamp assembly. The implementation verification was done by Petrova[46]. She implemented the C0 compiler in C0 itself and proved that this compiler produces the same code than the one in Isabelle/HOL. For this purpose she used a verification environment for sequential imperative programs integrated in Isabelle/HOL, developed by Schirmer[50]. This verification environment formalizes Hoare logics[17] for both partial and total correctness of programs in a general imperative programming language model which are both proven sound and relative complete in Isabelle/HOL. A verification condition generator, which is integrated in Isabelle/HOL in form of an Isabelle tactic, computes the weakest precondition for a Hoare triple by automatic backwards application of the Hoare rules. Thus by applying this verification condition generator tactic it remains a purely logical statement to prove in Isabelle/HOL.

Different to the described works in compiler verification, this thesis presents just the verification of a single compiler pass. It also differs from the described works in that it makes use of dependent types by representing the syntax by intrinsically typed syntax. Bach Poulsen et al.[4] investigate the use of intrinsic typing for interpreters of languages with mutable state and more general binding structures. Compared to our work they do not start from an imperative programming language but start from a simply-typed lambda calculus and extend it with ML-style references to obtain mutable state. They also contribute an Agda library which provides an intrinsically-typed syntax for scope graphs[39] and frames[48] which provide a uniform way to handle more general binding structures such as non-lexical binding.

The work of Pardo et al.[44] is similar to ours in that they also make use of dependent types to define a correct-by-construction compiler from an imperative language to a stack



based intermediate language. They do not only use dependent types to express the type system of the programming language but also its semantics.

Another approach is taken by Steinhöfel and Hähnle[51]. They aim for modularity and a high degree of automation in the verification process and introduce a framework where programming language semantics can be specified in terms of symbolic execution. They instantiate their framework for Java[20] and a subset of LLVM IR as source resp. target language.

8 Conclusion

In this thesis we introduced a representation of control-flow graphs that uses de Bruijn indices to represent labels and we formalized an intrinsically typed syntax and a big-step semantics for this control-flow graph representation. Furthermore we defined a translation of while programs into this control-flow graphs and proved the correctness of this translation. The representation of controls flow graphs in Section 5.5 differs from the original representation of the thesis supervisor in three ways. First in his representation the types of the environments are not fixed for a program but new variables can be declared in a program. Consequently labels do not only comprise the stack types there but also the environment types. Second he considered a programming language with function definitions. Hence there is a `return` statement instead of a `stop` statement. And third the thesis author further developed the original flow graph representation in the following way: in the original representation branchings saved two sub-flow graphs. Those two sub-flow graphs are replaced by two labels. This leads to more uniformity since branchings are now represented in the same way as unconditional jumps, namely by labels instead of sub-flow graphs. Additionally it makes definitions and proofs shorter.

Bibliography

- [1] *Agda's documentation*. URL: <https://agda.readthedocs.io/en/latest/> (visited on 04/11/2020).
- [2] Wolfgang Ahrendt et al., eds. *Deductive Software Verification - The KeY Book - From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6. URL: <https://doi.org/10.1007/978-3-319-49812-6>.
- [3] Eyad Alkassar et al. "The Verisoft Approach to Systems Verification". In: *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*. Ed. by Natarajan Shankar and Jim Woodcock. Vol. 5295. LNCS. Springer, 2008, pp. 209–224.
- [4] Casper Bach Poulsen et al. "Intrinsically-Typed Definitional Interpreters for Imperative Languages". In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158104. URL: <https://doi.org/10.1145/3158104>.
- [5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642058809.
- [6] S. Beyer et al. "Putting it all together - Formal Verification of the VAMP". In: (2006).
- [7] Sven Beyer. "Putting it all together - Formal Verification of the VAMP". PhD thesis. Saarland University, Saarbrücken, 2005.
- [8] Sandrine Blazy and Xavier Leroy. "Mechanized semantics for the Clight subset of the C language". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 263–288. DOI: 10.1007/s10817-009-9148-3.
- [9] François Bobot et al. "Why3: Shepherd Your Herd of Provers". In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. <https://hal.inria.fr/hal-00790310>. Wrocław, Poland, Aug. 2011, pp. 53–64.

-
- [10] *CakeML. A Verified Implementation of ML*. URL: <https://cakeml.org/> (visited on 04/11/2020).
- [11] Laurian M. Chirica and David F. Martin. “Toward Compiler Implementation Correctness Proofs”. In: *ACM Trans. Program. Lang. Syst.* 8.2 (Apr. 1986), pp. 185–214. ISSN: 0164-0925. DOI: 10.1145/5397.30847. URL: <http://doi.acm.org/10.1145/5397.30847>.
- [12] *CompCert*. URL: <http://compcert.inria.fr/> (visited on 04/11/2020).
- [13] Thierry Coquand and Christine Paulin. “Inductively defined types”. In: *COLOG-88*. Ed. by Per Martin-Löf and Grigori Mints. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 50–66. ISBN: 978-3-540-46963-6.
- [14] *CSmith*. URL: <https://embed.cs.utah.edu/csmith/> (visited on 04/11/2020).
- [15] *GCC. the GNU Compiler Collection*. URL: <https://gcc.gnu.org/> (visited on 04/11/2020).
- [16] Wolfgang Goerigk and Ulrich Hoffmann. “Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct”. In: *Applied Formal Methods — FM-Trends 98*. Ed. by Dieter Hutter et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 122–136. ISBN: 978-3-540-48257-4.
- [17] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: <https://doi.org/10.1145/363235.363259>.
- [18] *HOL. Interactive Theorem Prover*. URL: <https://hol-theorem-prover.org/> (visited on 04/11/2020).
- [19] *Isabelle*. URL: <https://isabelle.in.tum.de/> (visited on 04/11/2020).
- [20] *Java Language and Virtual Machine Specifications*. URL: <https://docs.oracle.com/javase/specs/> (visited on 04/11/2020).
- [21] G. Kahn. “Natural semantics”. In: *STACS 87*. Ed. by Franz J. Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 22–39. ISBN: 978-3-540-47419-7.
- [22] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.
- [23] Daniel Kröning. *Formal verification of pipelined microprocessors*. 2001. DOI: <http://dx.doi.org/10.22028/D291-25709>.

-
- [24] Ramana Kumar et al. “CakeML: A Verified Implementation of ML”. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’14. San Diego, California, USA: ACM, 2014, pp. 179–191. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535841. URL: <http://doi.acm.org/10.1145/2535838.2535841>.
- [25] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*. Palo Alto, California, Mar. 2004.
- [26] Vu Le, Mehrdad Afshari, and Zhendong Su. “Compiler Validation via Equivalence Modulo Inputs”. In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’14. Edinburgh, United Kingdom: ACM, 2014, pp. 216–226. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594334. URL: <http://doi.acm.org/10.1145/2594291.2594334>.
- [27] D. Leinenbach, W. Paul, and E. Petrova. “Towards the Formal Verification of a CO Compiler: Code Generation and Implementation Correctness”. In: *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005)*. 2005.
- [28] Dirk Leinenbach. “Compiler Verification in the Context of Pervasive System Verification”. PhD thesis. Saarland University, Saarbrücken, 2008.
- [29] Xavier Leroy. “Formal certification of a compiler back-end, or: programming a compiler with a proof assistant”. In: *POPL 2006: 33rd symposium Principles of Programming Languages*. ACM, 2006, pp. 42–54. DOI: 10.1145/1111037.1111042.
- [30] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.
- [31] Pierre Letouzey. “Extraction in Coq: An Overview”. In: *Logic and Theory of Algorithms*. Ed. by Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 359–369. ISBN: 978-3-540-69407-6.
- [32] John McCarthy and James Painter. “Correctness of a compiler for arithmetic expressions”. In: American Mathematical Society, 1967, pp. 33–41.
- [33] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0262631814.
- [34] S.M. Müller and W.J. Paul. *Computer Architecture, Complexity and Correctness*. Springer, 2000.

-
- [35] Magnus O. Myreen. “Verified just-in-time compiler on x86”. In: *Principles of Programming Languages (POPL)*. Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 107–118.
- [36] Magnus O. Myreen and Scott Owens. “Proof-producing Synthesis of ML from Higher-order Logic”. In: *SIGPLAN Not.* 47.9 (Sept. 2012), pp. 115–126. ISSN: 0362-1340. DOI: 10.1145/2398856.2364545. URL: <http://doi.acm.org/10.1145/2398856.2364545>.
- [37] George C. Necula. “Proof-Carrying Code”. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’97. Paris, France: Association for Computing Machinery, 1997, pp. 106–119. ISBN: 0897918533. DOI: 10.1145/263699.263712. URL: <https://doi.org/10.1145/263699.263712>.
- [38] George C. Necula and Peter Lee. “The Design and Implementation of a Certifying Compiler”. In: *SIGPLAN Not.* 33.5 (May 1998), pp. 333–344. ISSN: 0362-1340. DOI: 10.1145/277652.277752. URL: <https://doi.org/10.1145/277652.277752>.
- [39] Pierre Neron et al. “A Theory of Name Resolution”. In: *Programming Languages and Systems*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 205–231. ISBN: 978-3-662-46669-8.
- [40] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007. ISBN: 978-1-84628-691-9. DOI: 10.1007/978-1-84628-692-6. URL: <https://doi.org/10.1007/978-1-84628-692-6>.
- [41] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Vol. 2283. LNCS. Springer, 2002.
- [42] OCaml. URL: <https://ocaml.org/> (visited on 04/11/2020).
- [43] S. Owre, J. M. Rushby, and N. Shankar. “PVS: A prototype verification system”. In: *Automated Deduction—CADE-11*. Ed. by Deepak Kapur. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 748–752. ISBN: 978-3-540-47252-0.
- [44] Alberto Pardo et al. “An Internalist Approach to Correct-by-Construction Compilers”. In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP ’18. Frankfurt am Main, Germany: Association for Computing Machinery, 2018. ISBN: 9781450364416. DOI: 10.1145/3236950.3236965. URL: <https://doi.org/10.1145/3236950.3236965>.

-
- [45] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990. ISBN: 1-55880-069-8.
- [46] Elena Petrova. “Verification of the C0 Compiler Implementation on the Source Code Level”. PhD thesis. Saarland University, Saarbrücken, 2007.
- [47] A. Pnueli, M. Siegel, and E. Singerman. “Translation validation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 151–166. ISBN: 978-3-540-69753-4.
- [48] Casper Bach Poulsen et al. “Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics”. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Ed. by Shriram Krishnamurthi and Benjamin S. Lerner. Vol. 56. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 20:1–20:26. ISBN: 978-3-95977-014-9. DOI: 10.4230/LIPIcs.ECOOP.2016.20. URL: <http://drops.dagstuhl.de/opus/volltexte/2016/6114>.
- [49] Susmit Sarkar et al. “The semantics of x86-CC multiprocessor machine code”. In: *Principles of Programming Languages (POPL)*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 379–391.
- [50] Norbert Schirmer. “Verification of Sequential Imperative Programs in Isabelle/HOL”. PhD thesis. Technische Universität München, 2006.
- [51] Dominic Steinhöfel and Reiner Hähnle. “Modular, Correct Compilation with Automatic Soundness Proofs”. In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2018, pp. 424–447. ISBN: 978-3-030-03418-4.
- [52] *The Agda standard library*. URL: <https://github.com/agda/agda-stdlib> (visited on 04/11/2020).
- [53] *The Agda Wiki*. URL: <https://wiki.portal.chalmers.se/agda/pmwiki.php> (visited on 04/26/2020).
- [54] *The Coq Proof Assistant*. URL: <https://coq.inria.fr/> (visited on 04/11/2020).
- [55] *The KeY Project*. URL: <https://www.key-project.org/> (visited on 04/11/2020).
- [56] *The LLVM Compiler Infrastructure*. URL: <http://www.llvm.org/> (visited on 04/11/2020).

-
- [57] *Why3. Where Programs Meet Provers*. URL: <http://why3.lri.fr/> (visited on 04/11/2020).
- [58] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993. ISBN: 978-0-262-23169-5.
- [59] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532. URL: <http://doi.acm.org/10.1145/1993498.1993532>.