MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Coinductive Formalization of SECD Machine in Agda

MASTER'S THESIS

Bc. Adam Krupička

Brno, Fall 2018

Masaryk University
Faculty of Informatics

Coinductive Formalization of SECD Machine in Agda

Master's Thesis

Bc. Adam Krupička

Brno, Fall 2018

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Adam Krupička

Advisor: RNDr. Martin Jonáš

Acknowledgements

I would like to thank my friends and family for supporting me throughout this work.

I would also like to thank many users of the Freenode IRC network for helpful discussions regarding various topics connected to matters contained in this thesis. This list includes, but is not limited to, Ahmad Salim Al-Sibahi, Guillaume Allais, Miëtek Bak, Paolo G. Giarrusso, and Andrea Vezzosi.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

SECD Agda formalization coinduction

Contents

1 Introduction **1**

2 Logic, Constructivism, Type Theory **3**
- 2.1 *Intuitionistic logic* . . . 3
- 2.2 *Type Theory* . . . 4
  - 2.2.1 Curry-Howard Correspondence . . . 5
  - 2.2.2 Dependent Types . . . 5
  - 2.2.3 Inhabitance . . . 6

3 Agda **7**
- 3.1 *Overview* . . . 7
  - 3.1.1 Trivial Types . . . 8
  - 3.1.2 Booleans . . . 9
  - 3.1.3 Products . . . 9
  - 3.1.4 Natural numbers . . . 10
- 3.2 *Propositional Equality* . . . 11
- 3.3 *Decidable Equality* . . . 14
- 3.4 *Coinduction* . . . 15
  - 3.4.1 Streams . . . 16
  - 3.4.2 The Delay Monad . . . 18

4 Formalizing Type Systems **21**
- 4.1 *De Bruijn Indices* . . . 21
  - 4.1.1 Implementation . . . 22
- 4.2 *Example: Simply Typed λ Calculus* . . . 23
  - 4.2.1 Syntax . . . 23
  - 4.2.2 Semantics by Embedding into Agda . . . 25

5 SECD Machine **29**
- 5.1 *Introduction* . . . 29
- 5.2 *Definition* . . . 30
- 5.3 *Execution* . . . 30
- 5.4 *Recursion* . . . 32
  - 5.4.1 Function dump . . . 33
  - 5.4.2 Tail call optimization . . . 33

6 Formalization 35
6.1 *Syntax* 35
6.1.1 Preliminaries 35
6.1.2 Machine types 36
6.1.3 Syntax 37
6.1.4 Derived instructions 40
6.1.5 Examples 41
6.2 *Semantics* 46
6.2.1 Types 46
6.2.2 Auxiliary functions 48
6.2.3 Execution 49
6.2.4 A note regarding the function dump 54
6.2.5 Tests 54
6.3 *Compilation from a high-level language* 57
6.3.1 Syntax 57
6.3.2 Compilation 58

7 Epilogue 61

Bibliography 63

1 Introduction

> *There are occasions when it pays better to fight and be beaten than not to fight at all.*
>
> — George Orwell, *Homage to Catalonia*

Ever since the dawn of functional programming languages, there was a need for an efficient and practical execution model which would serve as the compilation target for the high-level functional languages. The first of these was the SECD machine, introduced by Landin in 1964.

Statically typed languages have many advantages over those lacking in this area. They give higher assurances to the correctness of the programs therein written, and they allow for stronger optimizations by the compiler. Perhaps most of all, however, they give the programmer a solid framework in which to reason about the code being written.

Typed low-level assembly languages are an area that perhaps has not received much consideration: all the current mainstream assembly languages do not depend on a type system. The compiler generating this low-level code is trusted to only generate valid programs. This is something we wish to address in this work, by introducing a type system for SECD machine instructions.

Still another direction of interest is that of formalizing programming languages.

In Chapter 2, we give a rough overview of constructivism, Intuitionistic logic, and type theory.

In Chapter 3, the language Agda is introduced by way of example. We concentrate on the concepts used in the rest of this thesis.

In Chapter 4, we show how the syntax and semantics of typed systems can be formalized in Agda. We introduce some machinery and, as an example, perform a rudimentary formalization of the Simply Typed λ Calculus.

In Chapter 5, we present the formalism of SECD machines. We also introduce an extension of this formalism used in Chapter 6.

Chapter 6 presents the main matter of this thesis. We formalize typed SECD code and then proceed to give it semantics by way of an embedding into Agda. Lastly, we define a high-level λ language and give compilation from this into typed SECD code.

2 Logic, Constructivism, Type Theory

> *What I cannot create, I do not understand.*
>
> — Richard Feynman

This chapter presents a quick overview of several rather complex areas. However, the information here presented should be sufficient for understanding the rest of this thesis.

Previous knowledge of mathematical logic and computability theory would be helpful in order for the reader to be able to appreciate all of the below content, however it is not assumed.

2.1 Intuitionistic logic

Intuitionistic logic [5, 6] is a logic that, unlike most of current mathematics, only allows for constructive arguments. In practice, the main difference is that proof by contradiction is not allowed: in order to show that something is the case, it is not enough to show that the opposite is not the case. In theory, this is achieved by disallowing the law of the excluded middle (LEM), which states that for any proposition P, P either does or does not hold:

$$\forall P.\ P \vee \neg P$$

Certain other well-known classical tautologies, such as double negation elimination,

$$\forall P.\ \neg\neg P \to P$$

are equivalent to this principle. It is also the case that the axiom of choice, as formulated in set theory, implies the law of the excluded middle, a result by Diaconescu [15].

Intuitionistic logic began as an attempt by Brouwer to develop a base for all mathematics that would more closely follow the intuitions of the human mind. Futhermore, the Stanford Encyclopedia of Philosophy's entry on Intuitionism [33] states,

> (...) to Brouwer the general LEM was equivalent to the a priori assumption that every mathematical problem has a

> solution — an assumption he rejected, anticipating Gödel's incompleteness theorem by a quarter of a century.

In practice, there are considerations with regards to constructive approaches other than a purely philosophical one. Under the standard Brouwer-Heyting-Kolmogorov interpretation of the Intuitionistic logic [39], working in this setting means that every proposition proven amounts to a recipe, an algorithm, on how to transform the assumptions, or inputs, into the result, or output. For this reason, Intuitionistic logic should be of high interest especially to computer scientists.

As an instructive example, consider the normalization of proofs in some theory. It has been discovered that if one can establish soundness and completeness of this theory with regard to some suitable semantics, this naturally gives rise to a normalizer for this theory [10]. In a constructive setting, the proof of an implication consists of a function, and so proofs of soundness and completeness give us a way to convert between the syntactic and semantic world. Reflecting a proof into the semantical structure (soundness), and reifying from the semantical structure back into syntax (completeness), we obtain a normalized version of the original proof. This approach to normalization is commonly referred to as normalization by evaluation and has been used as early as 1975 by Martin-Löf in order to establish decidability of typechecking for his Intuitionistic Theory of (dependent) Types theory [31], albeit not under the moniker of normalization by evaluation [2].

2.2 Type Theory

Type theory was first introduced by Russell and Whitehead in 1910 in their transformational work Principia Mathematica [45] as a response to Russell's discovery of inconsistency of the naïve set theory [37] in 1901. In type theory, every expression has an associated type, and there are rules for the formation of values and types dependent on these. Compare this with set theory, where propositions such as $2 \in 3$ can be formulated[1].

The next breakthrough in type theory was the discovery of the Simply Typed λ Calculus [8] by Church in 1940. This, too, came as a way to avoid paradoxes present in the Untyped λ Calculus [9], which was found to be inconsistent by Kleene and Rosser [24]. The Untyped λ calculus was introduced as a universal model of computation, a point at which it succeeded, as it is equivalent in strength to Turing machines [40].

2.2.1 Curry-Howard Correspondence

It was later observed by Howard that the Simply Typed λ Calculus (STLC) could be viewed as a language for construction of proofs in Natural Deduction [21] (ND), an intuitionistic proof calculus introduced originally by Gentzen in 1934 [17] as an attempt at a more natural language for expressing proofs. This correspondence simply states that *propositions* of ND are isomorphic with *types* in STLC, *proofs* of ND with *terms* (or programs) of STLC, and *normalization* of proofs in ND with *conversion into normal form* of terms of STLC.

This leads to the realization that we can prove theorems by writing computer programs, and that subsequently we can have these proofs verified by a type checker. However, in order to be able to express more interesting properties, we need a type system stronger than STLC.

2.2.2 Dependent Types

In order to extend the expressivity to non-trivial propositions, dependent types were proposed first by de Bruijn [7] in 1967 in his project Automath, aiming at creating a language for encoding computer-verified mathematics. Later, in 1972, Martin-Löf formulated his Intuitionistic Theory of Types [31], in which dependent types play a central role. More recently, starting in the mid 2000's, Voevodsky introduced Univalent Foundations [42], which aim to give practical foundations for modern mathematics in a way that allows for computer-verified proofs.

---

[1] The above being, in fact, true, as per the standard construction of natural numbers in set theory due to von Neumann [43].

Dependent types are types which can depend on values. They correspond with quantifiers from predicate logic, thus allowing one to naturally express more involved propositions.

The type that corresponds to universal quantification $\forall$ is the type of dependent functions $\Pi(a : A).B(a)$, where the type of B can depend on the value a. A proof of such a proposition consists of a function which for any value a produces the proof of $B(a)$. For example, consider the statement

$$\Pi(n : \mathbb{N}).\mathit{even}(n) \vee \mathit{odd}(n).$$

A proof of this proposition would consist of a decision procedure which for any natural number n determines whether n is even or odd and returns a proof of this fact.

Corresponding with existential quantification $\exists$ is the type of dependent products $\Sigma(a : A).B(a)$. A proof would consist of a pair of some value a of type A and a proof of $B(a)$. As an example, consider the statement that there exists a prime number,

$$\Sigma(n : \mathbb{N}).\mathit{prime}(n).$$

One possibility of a proof would be the number 2 and a proof that 2 is prime, which would hopefully be self-evident.

2.2.3 Inhabitance

An important concept is that of inhabitance of some type. An inhabited type is a type that is non-empty, i.e., there are some values of this type. This is analogous to the proposition that this type corresponds to a provable proposition. Thus the values of some type are sometimes called witnesses to (the provability of) the corresponding proposition.

3 Agda

> *Agda: is it a dependently-typed programming language? Is it a proof-assistant based on intuitionistic type theory?*
> `¯\(°_0)/¯` *Dunno, lol.*
>
> — From the topic of the official Agda IRC channel

Agda [35] is a functional programming language with first-class support for dependent types. As per the Curry-Howard correspondence, well-typed programs in Agda can also be understood as proofs of inhabitance of their corresponding types; types being understood as propositions.

This section is meant as a crash-course in Agda syntax, not semantics. In other words, those not familiar with dependently typed programming languages and/or proof assistants would do better to follow one of the books published on this topic. See [16] for an introduction to dependent types as a whole, or [38] for an in-depth introduction to dependendly typed programming and theorem proving in Agda.

3.1 Overview

Due to the presence of dependent types, functions defined in Agda must be by default[1] provably terminating. Failure to do so would result in type-checking becoming undecidable. However, this does not cause the loss of Turing-completeness; indeed in section 3.4 we present how possibly non-terminating computations can still be expressed, with some help from the type system.

Agda has strong support for mixfix operators[2] and Unicode identifiers. This often allows for developing a notation close to what one has come to expect in mathematics. For example, the following is valid Agda syntax:

---

[1] This restriction can be lifted, however it is at the user's own risk.

[2] Operators that can have multiple name parts and are infix, prefix, postfix, or closed [12].

```
MP : ∀ {Γ α β} → α , Γ ⊢ β
                 ----------
               → Γ ⊢ α ⇒ β
```

As an aside, there is also some support for proof automation in Agda [25], however from the author's experience, the usability of this tool is limited to simple cases. In contrast with tools such as Coq [3], Isabelle [34], or ACL2 [23], Agda suffers from a lower degree of automation: there are no built-in tactics, though their implementation is possible through reflection [36].

3.1.1 Trivial Types

A type that is trivially inhabited by a single value is often refered to as *Top* or *Unit*. In Agda,

```
data ⊤ : Set where
  · : ⊤
```

declares the new data type ⊤ which is itself of type Set[3]. The second line declares a value constructor for this type, here called simply ·, which constructs a value of type ⊤[4].

The dual of ⊤ is the trivially uninhabited type, often called *Bottom* or *Empty*. Complete definition in Agda follows.

```
data ⊥ : Set where
```

Note that there are no constructors declared for this type. Due to the inner workings of Agda, this guarantees us an inhabited type.

The empty type also allows us to define the negation of a proposition,

---

[3] For the reader familiar with the Haskell type system, the Agda type *Set* is akin to the Haskell kind *Star*. Agda has a stratified hierarchy of universes, where *Set* itself is of the type Set_1, and so on.

[4] Again for the Haskell-able, note how the syntax here resembles that of Haskell with the extension `GADTs`.

```
¬_ : Set → Set
¬ P = P → ⊥
```

Here we also see for the first time the notation for mixfix operators. Note the underscore _ in the name declaration of this function: it symbolizes where the argument is to be expected.

3.1.2 Booleans

A step-up from the trivially inhabited type ⊤, the type of booleans is made up of two distinct values.

```
data Bool : Set where
  tt ff : Bool
```

Since both constructors have the same type signature, we take advantage of a feature in Agda that allows us to declare such constructors on one line, together with the shared type.

Now we can declare a function that will perform negation of Boolean values,

```
not : Bool → Bool
not tt = ff
not ff = tt
```

Here we utilize pattern matching to split on the argument and transform each boolean value into the opposite.

Another function we can define is the conjunction of two boolean values, using a similar approach,

```
_∧_ : Bool → Bool → Bool
tt ∧ b = b
ff ∧ _ = ff
```

3.1.3 Products

To define the product type, it is customary to use a record. This will give us implicit projection functions from the type.

```
record _×_ (A : Set) (B : Set) : Set where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
```

Here we declare a new record type, parametrized by two other types, A and B. These are the types of the values stored in the pair, which we construct with the operator `_,_`.

As an example, we can create a pair of two boolean values,

```
_ : Bool × Bool
_ = tt , ff
```

Here we see another use of the underscore: we can use it as a placeholder in the stead of an identifier. This is useful in situations where we wish to give some example we won't be using in the future.

To showcase the use of projections, we can define an uncurried version of `_∧_` as a function from products of two boolean values,

```
conj : Bool × Bool → Bool
conj r = proj₁ r ∧ proj₂ r
```

In practice, however, it is often less cumbersome to instead employ pattern matching together with the constructor syntax in order to destructure a record,

```
conj′ : Bool × Bool → Bool
conj′ (a , b) = a ∧ b
```

3.1.4 Natural numbers

To see a more interesting example of a type, let us consider the type of natural numbers. These can be implemented using Peano encoding, as shown below.

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

Here we have a nullary constructor for the value zero, and then a unary value constructor, which corresponds to the successor function. As an example, consider the number 3, which would be encoded as `suc(suc(suc zero))`.

As an example of a function on the naturals, let us define the addition function.

```
_+_ : N → N → N
zero + b  = b
suc a + b = suc (a + b)
```

We proceed by induction on the left argument: if that number is zero, the result is simply the right argument. If the left argument is a successor of some number a, we inductively perform addition of a to b, and then apply the successor function to the result.

3.2 Propositional Equality

In this section, we take a short look at one of the main features of intuitionistic type theory, namely, the identity type. This type allows us to state the proposition that two values of some data type are *equal*. The meaning of *equal* here is that both of the values are convertible to the same value through reductions. This is the concept of propositional equality. Compare this with definitional equality, which only allows us to express when two values have the same syntactic representation. For example, definitionally it holds that $2 = 2$, however $1 + 1 = 2$ only holds propositionally, because a reduction is required on the left-hand side.

We can define propositional equality in Agda as follows.

```
data _≡_ {A : Set} : A → A → Set where
  refl : {x : A} → x ≡ x
```

The curly braces denote an implicit argument, i.e., an argument that is to be inferred by the type-checker. The equality type is polymorphic in this underlying type, A.

The only way we have to construct values of this type is by the constructor `refl`, which says that each value is propositionally equal to itself. Propositional equality is thus an internalization of definitional equality as a proposition: we say that two values are propositionally equal if there is a chain of reductions which lead to establishing definitional equality between the two values.

Unlike in axiomatic treatments of equivalence, symmetry and transitivity of `_≡_` are theorems in Agda:

```
sym : {A : Set} {a b : A} → a ≡ b → b ≡ a
sym refl = refl

trans : {A : Set} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans refl refl = refl
```

By pattern-matching on the proofs of equality we force Agda to unify the variables a, b, and c. This is possible because there are no other conditions on the variables here. In more complex situations, Agda may fail to perform unification: in such a case we are required to explicitly de-structure the involved terms until unification can succeed. After all the variables are unified, we are *de facto* constructing a proof of $a \equiv a$, which we do with the `refl` constructor.

Finally, let us see the promised proof of $1 + 1 = 2$,

```
1+1≡2 : 1 + 1 ≡ 2
1+1≡2 = refl
```

The proof is trivial, as $1+1$ reduces directly to two. A more interesting proof would be that of associativity of addition,

```
+−assoc : ∀ {a b c} → a + (b + c) ≡ (a + b) + c
+−assoc {zero}    = refl
+−assoc {suc a}   = let a+[b+c]≡[a+b]+c = +−assoc {a}
                     in ≡−cong suc a+[b+c]≡[a+b]+c
  where ≡−cong : {A B : Set} {a b : A}
                  → (f : A → B) → a ≡ b → f a ≡ f b
        ≡−cong f refl = refl
```

Here we proceed by induction on the variable a, which is given as an implicit argument: hence in the definition we surround the argument by curly braces in order to be able to access it. We have no need for the arguments b and c, and as they are implicit as well, we simply do not write them in the left-hand side of the definition.

In the base case when $a = 0$ we are asked to prove that

$$0 + (b + c) \equiv (0 + b) + c.$$

By the definition of addition, the left side simplifies to $b + c$. The right side simplifies to $b + c$ as well, therefore we are permitted to close the case by refl.

In the general case we are to prove

$$suc\ a + (b + c) \equiv (suc\ a + b) + c.$$

First we observe how this simplifies according to the definition of addition: the left side simplifies to $suc\ (a + (b + c))$, whereas the right side first to $suc\ (a + b) + c$ and then to $suc\ ((a + b) + c)$. Therefore we are to prove that

$$suc\ (a + (b + c)) \equiv suc\ ((a + b) + c).$$

To this end we obtain the inductive assumption,

$$a + (b + c) \equiv (a + b) + c.$$

Now all we need is to insert the suc into this assumption, which we do by a call to the lemma ≡−cong, which proves that propositional equality is a congruence with respect to unary functions, such as suc.

The reader may also notice the use of the quantifier ∀ in the type of +−assoc. This is an instruction to Agda to infer the types of the variables in the type signature, in this case inferring a, b, and c to be of the type ℕ. It does *not* have the meaning of universal quantification, instead all function types are universally quantified by default, similarly to e.g. Haskell.

3.3 Decidable Equality

A strengthening of the concept of propositional equality is that of *decidable equality*. This is a form of equality that, unlike Propositional equality, can be decided programatically. We define this equality as a restriction of propositional equality to those comparisons that are decidable. Firstly, we need the definition of a decidable relation.

```
data Dec (R : Set) : Set where
  yes : R → Dec R
  no  : ¬ R → Dec R
```

This data type allows us to embed either a yes or a no answer as to whether R is inhabited. For example, we can state that the type ⊤ is inhabited by producing the witness ·,

```
_ : Dec ⊤
_ = yes ·
```

and that the type ⊥ is not,

```
_ : Dec ⊥
_ = no λ()
```

by discharging the absurd pattern by λ(). The constructor no takes a value of type ¬⊥, which stands for ⊥ → ⊥. Since the left-hand side is absurd, Agda allows us to conclude anything, even ⊥, by this syntax.

Now we can define what it means for a type to possess decidable equality,

```
Decidable : (A : Set) → Set
Decidable A = ∀ (a b : A) → Dec (a ≡ b)
```

Here we specify that for any two values of that type we must be able to produce an answer whether they are equal or not.

As an example, let us define decidable equality for the type of Naturals. We also use this as an excuse to introduce the keyword `with` which can be used to make a case-split on some expression,

```
_≟N_ : Decidable N
zero ≟N zero      = yes refl
(suc _) ≟N zero = no λ()
zero ≟N (suc _) = no λ()
(suc m) ≟N (suc n) with m ≟N n
... | yes refl      = yes refl
... | no ¬m≡n = no λ m≡n → ¬m≡n (suc−injective m≡n)
   where suc−injective : ∀ {m n} → suc m ≡ suc n → m ≡ n
         suc−injective refl = refl
```

Given a proof of equality of two values of a decidable type, we can forget all about the proof and simply ask whether the two values are equal or not,

```
⌊_⌋ : {A : Set} {a b : A} → Dec (a ≡ b) → Bool
⌊ yes p ⌋   = tt
⌊ no ¬p ⌋ = ff
```

3.4 Coinduction

Total languages, such as Agda, are sometimes wrongfully accused of lacking Turing-completeness. In reality, there are ways to model possibly non-terminating programs – given some time limit for their execution. One such way is to introduce a monad which captures the concept of a recursive call [32].

In this section we introduce the concept of coinduction on the example of streams and then proceed to define a monad which will be used later on in chapter 5 to give semantics to the execution of SECD machine code.

For a more in-depth overview of coinduction in Agda and especially the aforementioned monad, please refer to [1].

The concepts presented can be made specific in category theory, where given a functor F we can speak of $F-$coalgebras. Coinduction, then, is a way of proving properties of such systems. Morally, the distinction between induction and coinduction is that induction proceeds by breaking down a problem into some base case, whereas coinduction starts with a base case and iteratively extends to subsequent steps.

Well-known examples of $F-$coalgebras include streams and transition systems. The moral distinction here is that while elements of algebraic structures, or data, are constructed, elements of coalgebraic structures, or codata, are observed.

For a more in-depth introduction to coalgebra, please see [22].

3.4.1 Streams

Streams are infinite lists. For example, consider the succession of all natural numbers: it is clearly infinite. In some functional languages, such as Haskell, this can be expressed as a lazily constructed list. Agda, however, being total, does not allow for such a construction directly: an infinite data structure is clearly not inductively constructible. It is, however, observable: as with a regular list, we can peek at its head `hd`, and we can drop the head and look at the tail `tl` of the stream.

To capture this in Agda, we define a record with these projections and mark it as `coinductive`,

```
record Stream (A : Set) : Set where
  coinductive
  field
    hd : A
    tl  : Stream A
```

As an example, consider the aforementioned stream of natural numbers, starting from some n,

```
nats : N → Stream N
hd (nats n) = n
tl (nats n)  = nats (n + 1)
```

Here we employ a feature of Agda called copatterns. Recall that we are constructing a record: the above syntax says how the individual fields are to be realized. Note also that the argument to nats is allowed to be structurally enlarged before the recursive call, something that would be forbidden in an inductive definition.

Given such a stream, we may wish to observe it by peeking forward a finite number of times, thus producing a List,

```
takeˢ : ∀ {A} → N → Stream A → List A
takeˢ zero xs     = []
takeˢ (suc n) xs = hd xs :: takeˢ n (tl xs)
```

Now we can convince ourselves that the above implementation of nats is, indeed, correct,

```
_ : takeˢ 7 (nats 0) ≡ 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: []
_ = refl
```

For a more interesting example of a stream, consider the Hailstone sequence [11], with a slight modification to the single step function, given below:

```
step : N → N
step 1 = 0
step n with even? n
... | tt = ⌊ n /2⌋
... | ff = 3 ∗ n + 1
```

The sequence itself, then, can be given by the following definition,

```
collatz : ℕ → Stream ℕ
hd (collatz n) = n
tl (collatz n)  = collatz (step n)
```

For example, observe the sequence starting from the number 12,

```
_ : takeˢ 11 (collatz 12)
  ≡ 12 ∷ 6 ∷ 3 ∷ 10 ∷ 5 ∷ 16 ∷ 8 ∷ 4 ∷ 2 ∷ 1 ∷ 0 ∷ []
_ = refl
```

As an aside, using a dependent product, we can express the predicate that a stream will eventually reach some given value,

```
Reaches : ∀ {A} → Stream A → A → Set
Reaches xs a = Σ ℕ (λ n → atˢ n xs ≡ a)
```

Here the binary function `atˢ` is used, which returns the n-th element of the stream xs.

Hence, the Collatz conjecture can be stated as follows:

```
conjecture : ∀ n → Reaches (collatz n) 0
```

3.4.2 The Delay Monad

The Delay monad captures the concept of unbounded recursive calls. There are two ways to construct a value of this type: `now`, which says that execution has terminated and the result is available, and `later`, which means the result is delayed by some indirection and *might* be available later. In Agda, we define this as a mutual definition of an inductive and coinductive data-type as follows,

```
mutual
  data Delay (A : Set) (i : Size) : Set where
    now  : A → Delay A i
    later : ∞Delay A i → Delay A i

  record ∞Delay (A : Set) (i : Size) : Set where
    coinductive
    field
      force : {j : Size< i} → Delay A j
```

Here we use the built-in type Size which serves as a measure on the size of the delay. Note that the field force requires this to strictly decrease. This measure aids the Agda type-checker in verifying that a definition is *productive*, that is, some progress is made in each iteration of force. The type Size< i is the type of all sizes strictly smaller than i.

For any data-type we may define an infinitely delayed value,

```
never : ∀ {i A} → Delay A i
never {i} = later λ where .force {j} → never {j}
```

This can be used to signal an error in execution has occurred. The implicit size argument has been written explicitly for the reader's sake.

Here we also see for the first time the anonymous syntax for constructing records by copatterns. The above is synonymous with

```
mutual
  never′ : ∀ {i A} → Delay A i
  never′ = later ∞never′

  ∞never′ : ∀ {i A} → ∞Delay A i
  force ∞never′ = never′
```

In other words, anonymous records allow us to succinctly construct codata by use of copatterns, without the need of writing unwieldy mutual blocks.

Given a delayed value, we can attempt to retrieve it in a given finite number of steps,

```
runFor : ∀ {A} → N → Delay A ∞ → Maybe A
runFor zero (now x)       = just x
runFor zero (later _)     = nothing
runFor (suc _) (now x)  = just x
runFor (suc n) (later x) = runFor n (force x)
```

This idiom is useful for executing a computation that periodically offers its environment a chance to interrupt the computation, or proceed further on.

Delay is also a monad, with the unit operator[5] being now and the bind operator given below,

```
_>>=_ : ∀ {A B i} → Delay A i → (A → Delay B i) → Delay B i
now x >>= f  = f x
later x >>= f = later λ where .force → (force x) >>= f
```

This allows us to chain delayed computations where one depends on the result of another.

---

[5] In Haskell terminology, *return*.

4 Formalizing Type Systems

λ calculus isn't invented, it's discovered.

— Philip Wadler

In what follows, we take a look at how we can use Agda to formalize deductive systems and/or typed calculi. We concern ourselves with the simplest example there is, the Simply Typed λ Calculus.

Deductive systems are formal languages which allow the statement and proof of propositions in a manner that makes the conclusions indisputable, as long as one can agree on assumptions used in the proof and laws of reason encompassed by the system.

For a more in-depth treatment of the topic of formalizing programming languages and programming language theory in Agda, please refer to [44].

λ calculi are arguably the simplest model of computation. They almost invariably contain the basic concepts, which are variables, function formation, and function application. They come in many forms and can be adapted to model any specific requirements we may have, e.g. resource-conscious linear calculi [18], concurrency-oriented process calculi [4], or calculi modeling quantum computing [41].

For a brief history of λ calculi, please refer to chapter 2.

4.1 De Bruijn Indices

Firstly, we shall need some machinery to make our lives easier. We could use string literals as variable names in our system, however this would lead to certain difficulties further on, such as increased complexity of the formalization due to the need to handle string comparisons and such. Instead, we shall use the concept commonly referred to as De Bruijn indices [14]. In this formalism variable identifiers consist of natural numbers, where each number n refers to the variable bound by the binder n positions above the current scope in the syntax tree. Some examples of this naming scheme are shown in Figure 4.1. The immediately apparent advantage of using De Bruijn

String syntax	De Bruijn syntax
`λx.x`	`λ 0`
`λx.λy.x`	`λλ 1`
`λx.λy.λz.x z (y z)`	`λλλ 2 0 (1 0)`

Figure 4.1: Examples of λ terms using the standard naming scheme on the left and using De Bruijn indices on the right.

indices is that α-equivalence[1] of λ terms becomes trivially decidable by way of purely syntactic equality. Other advantages include easier formalization.

4.1.1 Implementation

To implement De Bruijn indices in Agda, we express what it means for a variable to be present in a context. Context is a collection of assumptions we are equipped with in a given situation. We shall assume that a context is a list of assumptions, as this is how contexts will be defined in the next subsection. We will express list membership as a new data type,

```
data _∈_ {A : Set} : A → List A → Set where
  here  : ∀ {x xs} → x ∈ (x :: xs)
  there : ∀ {x a xs} → x ∈ xs → x ∈ (a :: xs)
```

The first constructor says that an element is present in a list if that element is the head of the list. The second constructor says that if we already know that our element x is in a list, we can extend the list with some other element a and x will still be present in the new list.

As a few examples of elements of _∈_ consider the following shorthands that we use in examples further on.

---

[1] The problem of whether two λ terms represent the same function.

```
0 : ∀ {A} {x : A} {xs : List A} → x ∈ (x :: xs)
0 = here

1 : ∀ {A} {x y : A} {xs : List A} → x ∈ (y :: x :: xs)
1 = there here

2 : ∀ {A} {x y z : A} {xs : List A} → x ∈ (z :: y :: x :: xs)
2 = there (there here)
```

Now we can also define a function which, given a proof that an element is in a list, returns the aforementioned element,

```
lookup : ∀ {A x xs} → x ∈ xs → A
lookup {x = x} here = x
lookup (there w)    = lookup w
```

Now if during the construction of some λ term we find ourselves in a situation in which we wish to introduce a variable pointing to some assumption from the context, we can give a value such as **1** to mean the second assumption in the context, encoded as a list. Additionally, this **1** will also serve as a witness to the fact that this or that specific assumption is, indeed, in the context.

4.2 Example: Simply Typed λ Calculus

In this subsection, in preparation of the main matter of this thesis, we introduce the way typed deductive systems can be formalized in Agda. As promised, we approach the formalization the Simply Typed λ Calculus.

4.2.1 Syntax

First, we define the types for expressions in our system.

```
data ⋆ : Set where
  τ   : ⋆
  _⇒_ : ⋆ → ⋆ → ⋆
```

Here we defined an atomic type τ and a binary type constructor for function types. The meaning of τ is currently completely arbitrary: it will become concrete when giving semantics.

We proceed by defining context as a list of assumptions, where every assumption is encoded directly by its type.

```
Context : Set
Context = List ⋆
```

Now we are finally able to define the deductive rules that make up the calculus, using De Bruijn indices as explained above.

```
data _⊢_ : Context → ⋆ → Set where
  var : ∀ {Γ α} → α ∈ Γ → Γ ⊢ α
  λ_  : ∀ {Γ α β} → α :: Γ ⊢ β → Γ ⊢ α ⇒ β
  _$_ : ∀ {Γ α β} → Γ ⊢ α ⇒ β → Γ ⊢ α → Γ ⊢ β
```

The constructors above correspond exactly to the typing rules of the calculus. In the first rule we employed the data type `_∈_` implenting De Bruijn indices: if we can give a witness to the membership of assumption α in the context, we can derive it. In the second rule, which captures the concept of λ-abstraction, we say that if from a context extended with α we can derive β, then we can form the function $\alpha \Rightarrow \beta$. The last rule is that of function application: if from some context we can derive a function $\alpha \Rightarrow \beta$, and we can derive also α, we may use the function to obtain a β.

We can see some examples now, below we hive λ terms corresponding the S and K combinators. In standard notation, S is defined as λ`x.`λ`y.x` and K as λ`x.`λ`y.`λ`z.x z (y z).`

```
K : ∀ {Γ α β} → Γ ⊢ α ⇒ β ⇒ α
K = λ λ (var 1)

S : ∀ {Γ α β γ} → Γ ⊢ (α ⇒ β ⇒ γ) ⇒ (α ⇒ β) ⇒ α ⇒ γ
S = λ λ λ var 2 $ var 0 $ (var 1 $ var 0)
```

Note how we use Agda polymorphism to construct a polymorphic term of our calculus — there is no polymorphism in the calculus itself.

The advantage of this presentation is that only well-typed syntax is representable. Thus, whenever we work with a term of our calculus, it is guaranteed to be well-typed, which often simplifies things. We see an example of this in the next subsection.

4.2.2 Semantics by Embedding into Agda

Now that we have defined the syntax, the next step is to give it semantics. We do this in a straightforward manned by way of embedding our calculus into Agda.

First, we define the semantics of types, by assigning Agda types to types in our calculus.

```
⟦_⟧⋆ : ⋆ → Set
⟦ τ ⟧⋆       = N
⟦ α ⇒ β ⟧⋆ = ⟦ α ⟧⋆ → ⟦ β ⟧⋆
```

Here we choose to realize our atomic type as the type of Natural numbers. These are chosen for being a nontrivial type. The function type is realized as an Agda function type.

Next, we give semantics to contexts.

```
⟦_⟧C : Context → Set
⟦ [] ⟧C      = ⊤
⟦ x ∷ xs ⟧C = ⟦ x ⟧⋆ × ⟦ xs ⟧C
```

The empty context can be realized trivially by the unit type. A non-empty context is realized as the product of the realization of the first element and, inductively, a realization of the rest of the context.

Now we are ready to give semantics to terms. In order to be able to proceed by induction with regard to the structure of the term, we must operate on open terms.

```
⟦_⟧_ : ∀ {Γ α} → Γ ⊢ α → ⟦ Γ ⟧C → ⟦ α ⟧⋆
```

The second argument is a realization of the context in the term, which we shall need for variables,

```
⟦ var here ⟧ (x , _)        = x
⟦ var (there x) ⟧ (_ , xs) = ⟦ var x ⟧ xs
```

Here we case-split on the variable, in case it is zero we take the first element of the context, otherwise we recurse into the context until we hit zero. Note that the shape of the context Γ is guaranteed here to never be empty, because the argument to var is a proof of membership for Γ. Thus, Agda realizes that Γ can never be empty and we need not bother ourselves with a case-split for the empty context; indeed, we would be hard-pressed to give it an implementation. In other words, we are allowed to pattern-match on the semantics of *Γ*, which is guaranteed to be a product of realizations of the types therein contained. This is an advantage of the typed syntax with De Bruijn indices, as we can never encounter an index which would be out of bounds with respect to the context.

```
⟦ λ x ⟧ γ = λ α → ⟦ x ⟧ (α , γ)
```

The case for lambda abstraction constructs an Agda anonymous function that takes as the argument a value of the corresponding type and compute the semantics for the lambda's body, after extending the context with the argument.

```
⟦ f $ x ⟧ γ = (⟦ f ⟧ γ) (⟦ x ⟧ γ)
```

Finally, to give semantics to function application, we simply perform Agda function application on the subexpressions, after having computed their semantics in the current context.

Thanks to propositional equality, we can embed tests directly into Agda code and see whether the terms we defined above receive the expected semantics.

```
KN : N → N → N
KN x _ = x

_ : ⟦ K ⟧ · ≡ KN
_ = refl

SN : (N → N → N) → (N → N) → N → N
SN x y z = x z (y z)

_ : ⟦ S ⟧ · ≡ SN
_ = refl
```

Since this thesis can only be rendered if all the Agda code has successfully type-checked, the fact that the reader is currently reading this paragraph means the semantics functions as expected!

5 SECD Machine

> *Any language which by mere chance of the way it is written makes it extremely difficult to write compositions of functions and very easy to write sequences of commands will, of course, in an obvious psychological way, hinder people from using descriptive rather than imperative features. In the long run, I think the effect will delay our understanding of basic similarities, which underlie different sorts of programs and different ways of solving problems.*
>
> — Christopher Strachey, discussion following [28], 1966

5.1 Introduction

The original **S**tack, **E**nvironment, **C**ontrol, **D**ump machine is a stack-based, call-by-value abstract execution machine that was first outlined by Landin in [27]. It was regarded as an underlying model of execution for a family of languages, specifically, languages based on the abstract formalism of λ calculus.

Other machines modeling execution of functional languages have since been proposed, some derived from SECD, others not. Notable examples are the Krivine machine [26], which implements a call-by-name semantics, and the ZAM (Zinc abstract machine), which serves as a backend for the OCaml strict functional programming language [30].

For an overview of different kinds of SECD machines, including a modern presentation of the standard call-by-value, and also call-by-name and call-by-need versions of the machine, see [13].

There have also been hardware implementations of this formalism, e.g. [19, 20].

This chapter is meant as an intuitive overview of the formalism. We present the machine with the standard call-by-value semantics, following the original presentation by Landin [27]. At the end of this chapter we also present an extension of the machine with a *function* dump, which, to the best knowledge of the author, is a novel concept. This extension is crucial for the formalization of SECD with typed code in the next chapter.

5.2 Definition

The machine operates by executing instructions stored as a linked list, referred to as the control. Each instruction has the ability to change the machine state. The machine is notable for its first-class treatment of functions. As a natural target of compilation from the Untyped λ Calculus, the machine is a Turing-complete formalism.

Faithful to its name, the machine is made up of four components:

- Stack – stores values and functions operated on. Atomic operations, such as integer addition, are performed here;
- Environment – stores immutable assignments, such as function arguments and values bound by the *let* construct;
- Control – stores a list of instructions awaiting execution;
- Dump – serves as a store for pushing the current context when a function call is performed. The context is again retrieved when a function call returns.

The standard implementation sees all four of the above items as linked lists.

5.3 Execution

Execution of the machine consists of reading instructions from the Control and modifying the state of the machine as necessary. The basic instructions are:

- `ld x` – load the value bound to the identifier `x` from the environment and put it on the stack;
- `ldf f` – load the function — i.e., a sequence of instructions — `f` in the current environment, constructing a *closure*, and put it on the stack. A Closure is therefore a list of instructions together with an environment it can be executed in;
- `ap` – given that a closure and a value are present on the top of the stack, perform function application and afterwards put

the return value on the stack. Function application consists of popping the closure and value from the stack, dumping the current context onto the dump, emptying the stack, installing the closure's environment together with the argument, and finally loading the closure's code into the control register;

- `rtn` – return from a function, restoring the context from the function dump.

In addition, there are instructions for primitive operations, such as integer addition, list operations such as the head and tail operations, etc. All these only transform the stack, e.g. integer addition would consume two integers from the top of the stack and put back the result.

We use the notation $f[e]$ to mean the closure of function f in the environment e and $\emptyset$ to mean an empty stack, environment, control, or dump. The notation $e(x)$ refers to the value in environment e bound to the identifier x.

To see how the basic instructions and the addition instruction transform the machine state, please refer to Figure 5.1. It is usual to use

Before				After			
S	E	C	D	S′	E′	C′	D′
s	e	`ld x`,c	d	$e(x),s$	e	c	d
s	e	`ldf f`,c	d	$f[e],s$	e	c	d
$x,f[e'],s$	e	`ap`,c	d	$\emptyset$	x,e'	f	$(s,e,c),d$
y,s	e	`rtn`,c	$(s',e',c'),d$	y,s'	e'	c'	d
a,b,s	e	`add`,c	d	$a+b,s$	e	c	d

Figure 5.1: The above table presents the transition relation of the SECD Machine. On the left is the state of the machine before the execution of a single instruction. On the right is the newly mutated state. The components are read from left to right, i.e., the top is the leftmost value.

De Bruijn indices when referring to identifiers in the `ld` instruction. E.g. `ld 0` loads the topmost value in the environment and puts it on the stack. Hence, De Bruijn indices are used in the example in this chapter. They will also be used in the following chapter in the Agda formalization.

S	E	C	D
∅	∅	`ldf f, ldc 1, ap, …`	∅
f[∅]	∅	`ldc 1, ap, ldc 3, add`	∅
1,*f*[∅]	∅	`ap, ldc 3, add`	∅
∅	1	`ldc 1, ld 0, add, rtn`	(∅,∅,`ldc 3, add`)
1	1	`ld 0, add, rtn`	(∅,∅,`ldc 3, add`)
1,1	1	`add, rtn`	(∅,∅,`ldc 3, add`)
2	1	`rtn`	(∅,∅,`ldc 3, add`)
2	∅	`ldc 3, add`	∅
3,2	∅	`add`	∅
5	∅	∅	∅

Figure 5.2: Example execution from an empty initial state of the code `ldf f, ldc 1, ap, ldc 3, add` where `f` = `ldc 1, ld 0, add, rtn`.

To see an example of execution of the machine, please refer to Figure 5.2. This example loads a function, the number 1, and performs application of the loaded function, turned closure, to the number. The closure increments its argument by one. After the closure returns, 3 is added to the returned value. The final state has the number 5 on the stack.

5.4 Recursion

An attentive reader may notice that the above presentation does not give an obvious way of implementing recursive functions. The issue is that after execution of the instruction `ap`, the closure being applied is not retrievable. In practice, this may be resolved by storing the function on the stack that is then pushed onto the dump, and reconstructing the closure with the corresponding environment from the dump when needed.

The additional instructions introduced in this section are summarized in Figure 5.3.

5.4.1 Function dump

We propose a dedicated register for storing the closure being applied. Referred to as the *function dump*, the closure being applied is pushed onto this register during the instruction `ap`. With the instruction `rtn`, the topmost closure is then popped from the function dump and execution proceeds as normal. The advantages of this approach consist of: (1) better isolation of components of the machine, leading to a simpler description and formalization, and (2) the fact that the closure does not need to be reconstructed from the code of the function and the corresponding environment.

We also introduce the instruction `ldr` for loading closures from the function dump. It behaves analogously to the instruction `ld`, with the difference that `ld` serves for retrieving values from the environment, whereas `ldr` retrieves closures from the function dump.

Before					After				
S	E	C	F	D	S′	E′	C′	F′	D′
s	e	`ldr x`,c	f	d	$f(x),s$	e	c	f	d
$x,c'[e'],s$	e	`rap`,c	f	d	$\emptyset$	x,e'	c'	f	d

Figure 5.3: The above table presents the additional instructions from this section. The capital letter F stands for the function dump.

5.4.2 Tail call optimization

Another consideration is the elimination of tail calls. The standard approach [29] is to introduce a new instruction `rap` which behaves similarly to `ap`, with the modification that it does not bother pushing a return frame onto the dump.

6 Formalization

In this chapter, we approach the main topic of this thesis. We formalize a SECD machine in Agda, with typed syntax, and then proceed to define the semantics by way of coinduction. By typed syntax we mean an approach to SECD code which performs static verification of the code in question. For example, the add instruction is only allowed when two integers are provably on the top of stack, and a function can only access values from the environment if the function in question states so in advance in its type.

Finally, we define a typed λ calculus, corresponding to the capabilities of the SECD machine, and define a compilation procedure from this calculus to typed SECD programs.

6.1 Syntax

6.1.1 Preliminaries

Before we can proceed, we shall require certain machinery to aid us in formalizing the type system.

We define the data type Path, parametrized by a binary relation, whose values are finite sequences of values such that each value is in relation with the next one.

```
data Path {A : Set} (R : A → A → Set) : A → A → Set where
  ∅      : ∀ {a} → Path R a a
  _>>_ : ∀ {a b c} → R a b → Path R b c → Path R a c
```

The first constructor creates an empty path. The second takes an already existing path and prepends to it a value, given a proof that this value is in relation with the first element of the already-existing path. The reader may notice certain similarity to linked lists; indeed if for the relation we take the universal relation for A, we obtain a type that is isomorphic to linked lists.

We can view this type as the type of finite paths through a graph connected according to the binary relation.

We also define a shorthand for constructing the end of a path out of two edges. We use this in examples later on.

```
_>|_ : ∀ {A R} {a b c : A} → R a b → R b c → Path R a c
a >| b = a >> b >> ∅
```

Furthermore, we can also concatenate two paths, given that the end of the first path connects to the start of the second one. This is enforced by the type system of Agda.

```
_>+>_ : ∀ {A R} {a b c : A} → Path R a b → Path R b c → Path R a c
∅ >+> r          = r
(x >> l) >+> r = x >> (l >+> r)
```

6.1.2 Machine types

We start by defining the atomic constants our machine operates on. We limit ourselves to booleans and integers.

```
data Const : Set where
  bool : Bool → Const
  int   : Z → Const
```

Next, we define an Agda data type which captures the machine's types.

```
data Type : Set where
  intT boolT : Type
  pairT         : Type → Type → Type
  listT         : Type → Type
  _⇒_          : Type → Type → Type
```

Firstly, there are types corresponding to the constants we have already defined above. Then, we also introduce a product type and a list type. Finally, there is the function type, _⇒_, in infix notation.

Now, we define the type assignment of constants.

```
typeof : Const → Type
typeof (bool _) = boolT
typeof (int _)   = intT
```

Next, we define the typed stack, environment, and function dump.

```
Stack     = List Type
Env       = List Type
FunDump = List Type
```

For now, these only store the information regarding the types of the values in the machine. Later, when defining semantics, we will give realizations to these, similarly to how we handled contexts in the formalization of Simply Typed λ Calculus in 4.2.2.

Finally, we define the state as a record storing the stack, environment, and the function dump. Note that we do not formalize the dump register.

```
record State : Set where
  constructor _#_#_
  field
    s : Stack
    e : Env
    f : FunDump
```

Note that, unlike in the standard presentation of SECD Machines, which we saw in chapter 4, here the state does not include the code. This is because we are aiming for a version of SECD with typed assembly code: code will be defined in the next subsection as a binary relation on states.

6.1.3 Syntax

Since we aim to have typed code, we have to take a different approach to defining code. We define a binary relation that determines how a state of a certain *shape* is mutated following the execution of an instruction. By shape, we mean the types present in the separate components of the state. Using pattern matching, we are able to put certain restrictions on these, e.g. we can require that preceding a certain instruction, an integer must be on the top of the stack.

We have two versions of this relation: the first is the single-step relation, the second is the reflexive and transitive closure of the first using Path.

```
infix 5 ⊢_▷_
infix 5 ⊢_⇝_
```

Their definitions need to be mutually recursive, because certain instructions — defined in the single-step relation — need to refer to whole programs, a concept captured by the multi-step relation.

```
mutual
  ⊢_⇝_ : State → State → Set
  ⊢ s₁ ⇝ s₂ = Path ⊢_▷_ s₁ s₂
```

There is nothing surprising here, we use Path to define the multi-step relation.

Next, we define the single-step relation. As mentioned before, this relation captures how one state might change into another.

```
data ⊢_▷_ : State → State → Set where
```

Here we must define all the instructions our machine should handle. We start with the simpler ones.

```
ldc : ∀ {s e f}
  → (const : Const)
  → ⊢ s # e # f ▷ (typeof const :: s) # e # f
```

Instruction ldc loads a constant which is embedded in it. It poses no restrictions on the state of the machine and mutates the state by pushing the type of constant on the stack.

$$\begin{aligned}&\mathsf{ld} : \forall\ \{s\ e\ f\ a\}\\ &\quad\to (a \in e)\\ &\quad\to\ \vdash s \mathbin{\#} e \mathbin{\#} f \rhd (a :: s) \mathbin{\#} e \mathbin{\#} f\end{aligned}$$

Instruction ld loads a value of type a from the environment and puts it on the stack. It requires a proof that this value is, indeed, in the environment.

$$\begin{aligned}&\mathsf{ldf} : \forall\ \{s\ e\ f\ a\ b\}\\ &\quad\to (\vdash [\,] \mathbin{\#} (a :: e) \mathbin{\#} (a \Rightarrow b :: f) \rightsquigarrow [\ b\ ] \mathbin{\#} (a :: e) \mathbin{\#} (a \Rightarrow b :: f))\\ &\quad\to\ \vdash s \mathbin{\#} e \mathbin{\#} f \rhd (a \Rightarrow b :: s) \mathbin{\#} e \mathbin{\#} f\end{aligned}$$

The ldf instruction is considerably more involved. It loads a function of the type $a \Rightarrow b$, given as an argument, and puts it on the stack. In addition, the code we are loading also has to be of a certain shape to make it a function: it starts with the empty stack and finishes with a single value of type b (being returned) on the stack, the argument of type a it was called with must be put in the environment, and the function dump is to be extended with the type of the function itself to permit recursive calls in the function body. Note that we use the multi-step relation here to describe the type of the code.

Once a function is loaded, we may apply it,

$$\begin{aligned}&\mathsf{ap} : \forall\ \{s\ e\ f\ a\ b\}\\ &\quad\to\ \vdash (a :: a \Rightarrow b :: s) \mathbin{\#} e \mathbin{\#} f \rhd (b :: s) \mathbin{\#} e \mathbin{\#} f\end{aligned}$$

The instruction ap requires that a function and its argument are on the stack. After it has run, the returned value from the function will be put on the stack in their stead. The type of this instruction is fairly simple, the difficult part awaits us further on in the implementation.

$$\begin{aligned}&\mathsf{rtn} : \forall\ \{s\ e\ a\ b\ f\}\\ &\quad\to\ \vdash (b :: s) \mathbin{\#} e \mathbin{\#} (a \Rightarrow b :: f) \rhd [\ b\ ] \mathbin{\#} e \mathbin{\#} (a \Rightarrow b :: f)\end{aligned}$$

Return is an instruction we are to use at the end of a function in order to get the machine state into the one required by ldf. It throws away what is on the stack, with the exception of the return value.

Next, let us look at recursive calls.

```
ldr : ∀ {s e f a b}
    → (a ⇒ b ∈ f)
    → ⊢ s # e # f ▷ (a ⇒ b ∷ s) # e # f
```

The instruction ldr loads a function for a recursive application from the function dump. We can be many scopes deep in the function and we use a De Bruijn index here to count the scopes, same as we do with the environment. This is important, e.g., for curried functions where we want to be able to load the topmost function, not one that was already partially applied.

```
rap : ∀ {s e f a b}
    → ⊢ (a ∷ a ⇒ b ∷ s) # e # f ▷ [ b ] # e # f
```

This instruction looks exactly the same way as ap. The difference will be in implementation, as this one will attempt to perform tail call elimination.

```
if : ∀ {s s′ e f}
   → ⊢ s # e # f ↝ s′ # e # f
   → ⊢ s # e # f ↝ s′ # e # f
   → ⊢ (boolT ∷ s) # e # f ▷ s′ # e # f
```

The if instruction requires that a boolean value is present on the top of the stack. Based on this value, it decides which branch to execute. Here we hit on one limitation of the typed presentation: both branches must finish with a stack of the same shape, otherwise it would be unclear what the stack looks like after this instruction and static verification of the typed code could not proceed.

The remaining instructions are fairly simple in that they only manipulate the stack. Their types are outlined in Figure 6.1.

6.1.4 Derived instructions

For the sake of sanity we also define what amounts to simple programs, masquerading as instructions, for use in more complex programs later. The chief limitation here is that since these are members

Instruction	Stack before	Stack after
nil	*s*	listT *a* :: *s*
flp	*a* :: *b* :: *s*	*b* :: *a* :: *s*
cons	*a* :: listT *a* :: *s*	listT *a* :: *s*
head	listT *a* :: *s*	*a* :: *s*
tail	listT *a* :: *s*	listT *a* :: *s*
pair	*a* :: *b* :: *s*	pairT *a* *b* :: *s*
fst	pairT *a* *b* :: *s*	*a* :: *s*
snd	pairT *a* *b* :: *s*	*b* :: *s*
add	intT :: intT :: *s*	intT :: *s*
sub	intT :: intT :: *s*	intT :: *s*
mul	intT :: intT :: *s*	intT :: *s*
eq?	*a* :: *a* :: *s*	boolT :: *s*
not	boolT :: *s*	boolT :: *s*

Figure 6.1: Instructions implementing primitive operations and their associated types, i.e., their manipulations of the stack.

of the multi-step relation, we have to be mindful when using them and use concatenation of paths, _>+>_, as necessary.

```
nil? : ∀ {s e f a} → ⊢ (listT a :: s) # e # f ⇝ (boolT :: s) # e # f
nil? = nil >| eq?

loadList : ∀ {s e f} → List N → ⊢ s # e # f ⇝ (listT intT :: s) # e # f
loadList []        = nil >> ∅
loadList (x :: xs) = loadList xs >+> ldc (int (+ x)) >| cons
```

The first one is simply the check for an empty list. The second one is more interesting, it constructs a sequence of instructions which will load a list of natural numbers. Note that the constructor +_ is used to construct a positive Agda integer from a natural number.

6.1.5 Examples

In this section, we present some examples of SECD programs in our current formalism. Starting with trivial ones, we work our way up to using full capabilities of the machine.

The first example loads two constants and adds them.

```
2+3 : ⊢ [] # [] # [] ↝ [ intT ] # [] # []
2+3 =
     ldc (int (+ 2))
  >> ldc (int (+ 3))
  >| add
```

The second example constructs a function which expects an integer and increases it by one before returning it.

```
inc : ∀ {e f} → ⊢ [] # (intT ∷ e) # (intT ⇒ intT ∷ f)
                ↝ [ intT ] # (intT ∷ e) # (intT ⇒ intT ∷ f)
inc =
    ld 0
  >> ldc (int (+ 1))
  >> add
  >| rtn
```

Here, we can see the type of the expression getting more complicated. We use polymorphism to make sure we can load this function in any environment. In the type of the environment, we have to declare that an argument of type intT is expected, and the function dump has to be extended with the type of this function.

In the next example, we load the above function and apply it to the integer 2.

```
inc2 : ⊢ [] # [] # [] ↝ [ intT ] # [] # []
inc2 =
  ldf inc
  >> ldc (int (+ 2))
  >| ap
```

In the next example we test partial application.

```
λTest : ⊢ [] # [] # [] ⇝ [ intT ] # [] # []
λTest =
  ldf
    (ldf
      (ld 0 >> ld 1 >> add >| rtn) >| rtn)
  >> ldc (int (+ 1))
  >> ap
  >> ldc (int (+ 2))
  >| ap
```

First we construct a function which constructs a function which adds two topmost values from the environment. The types of these two values are inferred to be integers by Agda, as this is what the add instruction requires. Then, we load and apply the constant 1. This results in another function, partially applied. Lastly, we load 2 and apply.

In the example inc, we saw how we could define a function. In the next example we also construct a function. However, this time we embed the instruction ldf in our definition directly, as this simplifies the type considerably.

```
plus : ∀ {s e f} → ⊢ s # e # f ▷ ((intT ⇒ intT ⇒ intT) :: s) # e # f
plus = ldf (ldf (ld 0 >> ld 1 >> add >| rtn) >| rtn)
```

The only consideration is that when we wish to use this function in another program, rather than writing ldf plus, we must only write plus.

For an example of a recursive function, consider that of a mapping function,

```
map : ∀ {e f a b} →
      ⊢ [] # e # f ▷ [ (a ⇒ b) ⇒ listT a ⇒ listT b ] # e # f
map = ldf (ldf body >| rtn)
  where
  body =
        ld 0
        >> nil?
        >+> if (nil >| rtn)
              (ldr 0 >> ld 0 >> tail >> ap
              >> ld 1 >> ld 0 >> head >> ap
              >| cons)
        >> ∅
```

Here we first load the list we are mapping over. We check for emptiness, if the argument is empty we return the empty list. In the case that it is not, we need to make a recursive call. We use a trick: we load map already partially applied with the first argument, i.e., the mapping function, with the call ldr **0**. Then we load the list with ld **0**, obtain its head, and apply. Result is the rest of the list already mapped over.

Now to map over the first element, we load the mapping function with ld **1**. Then we retrieve the first element of the list — similarly as we did above, only this time we use head — and apply. Result is the newly mapped element on the top of the stack and the rest of the transformed list below it: we cons the two.

Lastly, a more involved example: that of a folding function. Here we test all capabilities of the machine.

```
foldl : ∀ {e f a b} →
        ⊢ [] # e # f ▷ [ ((b ⇒ a ⇒ b) ⇒ b ⇒ (listT a) ⇒ b) ] # e # f
foldl = ldf (ldf (ldf body >| rtn) >| rtn)
  where
    body =
          ld 0
      >> nil?
      >+> if (ld 1 >| rtn)
             (ld 2 >> ld 1 >> ap
        >> ld 0 >> head >> ap
        >> ldr 2 >> ld 2 >> ap
        >> flp >> ap
        >> ld 0 >> tail >| rap)
      >> ∅
```

To start, we load the list we are folding. We check whether it is empty: if so, the accumulator **1** is loaded and returned. On the other hand, if the list is not empty, we start with loading the folding function **2**. Next, we load the accumulator **1**. We perform partial application. Then, we load the list **0** and obtain its first element with head. We apply to the already partially-applied folding function, yielding the new accumulator on the top of the stack.

Now we need to make the recursive call: we load the function foldl itself with ldr **2**. Next we need to apply all three arguments: we start with loading the folding function **2** and applying it. We are now in a state where the partially applied foldl is on the top of the stack and the new accumulator is right below it; we flip[1] the two and apply. Lastly, we load the list **0**, drop the first element with tail and perform the recursive application with tail-call elimination.

---

[1] Note we could have reorganized the instructions in a manner so that this flip would not be necessary. Indeed, later we see that there is no need for this instruction in section 6.3

6.2 Semantics

Having defined the syntax, we can now turn to semantics. In this section, we give operational semantics to the SECD machine syntax defined in the previous section.

6.2.1 Types

We begin, similarly to how we handled the semantics in Section 4.2.2, by giving semantics to the types. Here we have to proceed by mutual induction, as in certain places we need to make references to the semantics of other types, and vice versa. The order of the following definitions is arbitrary from the point of view of correctness and was chosen purely for improving readability.

```
mutual
  ⟦_⟧ᵗ : Type → Set
  ⟦ intT ⟧ᵗ       = Z
  ⟦ boolT ⟧ᵗ      = Bool
  ⟦ pairT t₁ t₂ ⟧ᵗ = ⟦ t₁ ⟧ᵗ × ⟦ t₂ ⟧ᵗ
  ⟦ a ⇒ b ⟧ᵗ      = Closure a b
  ⟦ listT t ⟧ᵗ    = List ⟦ t ⟧ᵗ
```

Here we realized the machine types as the corresponding types in Agda. The exception is the type of functions, which we realize as a closure. The meaning of Closure will be defined at a later moment in the mutual block.

We proceed by giving semantics to the environment,

```
⟦_⟧ᵉ : Env → Set
⟦ [] ⟧ᵉ      = T
⟦ x :: xs ⟧ᵉ = ⟦ x ⟧ᵗ × ⟦ xs ⟧ᵉ
```

The semantics of an environment are fairly straightforward, we make a reference to the semantic function for types and inductively define the environment as a product of semantics of each type in it.

Next, we define the semantics of the function dump, which will be necessary in the definition in Closure,

```
⟦_⟧ᵈ : FunDump → Set
⟦ [] ⟧ᵈ              = ⊤
⟦ intT :: xs ⟧ᵈ       = ⊥
⟦ boolT :: xs ⟧ᵈ      = ⊥
⟦ pairT x x₁ :: xs ⟧ᵈ = ⊥
⟦ a ⇒ b :: xs ⟧ᵈ      = Closure a b × ⟦ xs ⟧ᵈ
⟦ listT x :: xs ⟧ᵈ    = ⊥
```

Since the type of the function dump technically permits also non-function types in it, we have to handle them here by simply saying that they may not be present in the function dump. There is, after all, no instruction that would allow putting a non-function type in the dump.

Now, finally for the definition of Closure. We define it as a record containing the code of the function, a realization of the starting environment, and finally a realization of the function dump. Recall that the function dump contains the closures introduced by ldf instructions higher in the syntax tree.

```
record Closure (a b : Type) : Set where
  inductive
  constructor ⟦_⟧ᶜ×⟦_⟧ᵉ×⟦_⟧ᵈ
  field
    {e}  : Env
    {f}  : FunDump
    ⟦c⟧ᶜ : ⊢ [] # (a :: e) # (a ⇒ b :: f)
             ⇝ [ b ] # (a :: e) # (a ⇒ b :: f)
    ⟦e⟧ᵉ : ⟦ e ⟧ᵉ
    ⟦f⟧ᵈ : ⟦ f ⟧ᵈ
```

This concludes the mutual block of definitions.

There is one more type we have not handled yet, Stack, which is not required to be in the mutual block above,

```
⟦_⟧ˢ : Stack → Set
⟦ [] ⟧ˢ      = ⊤
⟦ x :: xs ⟧ˢ = ⟦ x ⟧ᵗ × ⟦ xs ⟧ˢ
```

The stack is realized similarly to the environment, however the environment is referenced in the definition of `Closure`, making it necessary for it to be in the mutual definition block.

6.2.2 Auxiliary functions

In order to proceed with giving semantics to SECD execution, we first need a few auxiliary functions to lookup values from the environment and from the function dump.

As for the environment, the situation is fairly simple,

```
lookupᵉ : ∀ {x xs} → ⟦ xs ⟧ᵉ → x ∈ xs → ⟦ x ⟧ᵗ
lookupᵉ (x , _) here        = x
lookupᵉ (_ , xs) (there w) = lookupᵉ xs w
```

Looking up values from the function dump is slightly more involved, because Agda doesn't let us pattern-match on the first argument as we did here. This is because the definition of `⟦_⟧ᵈ` is more involved than that of `⟦_⟧ᵉ`. Specifically, there is the possibility of the dump being uninhabited. Instead, we must define an auxiliary function to drop the first element of the product,

```
tailᵈ : ∀ {x xs} → ⟦ x :: xs ⟧ᵈ → ⟦ xs ⟧ᵈ
tailᵈ {intT} ()
tailᵈ {boolT} ()
tailᵈ {pairT x x₁} ()
tailᵈ {a ⇒ b} (_ , xs) = xs
tailᵈ {listT x} ()
```

We pattern-match on the type of the value in the environment. This forces Agda to realize that only a closure may be in the function dump, at which point we can pattern-match on the product and drop the first element.

Now we can define the lookup operation for the function dump,

```
lookupᵈ : ∀ {a b f} → ⟦ f ⟧ᵈ → a ⇒ b ∈ f → Closure a b
lookupᵈ (x , _) here = x
lookupᵈ f (there w)  = lookupᵈ (tailᵈ f) w
```

dropping the elements as necessary with `tailᵈ` until we get to the desired closure.

Lastly, we define a function for comparing two values of the machine,

```
compare : {t : Type} → ⟦ t ⟧ᵗ → ⟦ t ⟧ᵗ → Bool
compare {intT} a b                     = ⌊ a ≟Z b ⌋
compare {boolT} a b                    = ⌊ a ≟B b ⌋
compare {pairT _ _} (a , b) (c , d)    = compare a c ∧ compare b d
compare {listT _} [] []                = tt
compare {listT _} (a :: as) (b :: bs) = compare a b ∧ compare as bs
compare {listT _} _ _                  = ff
compare {_ ⇒ _} _ _                    = ff
```

The above code implements standard comparison by structural induction. Functions `_≟Z_` and `_≟B_` are standard functions implementing decidable equality for the corresponding types. We refuse to perform any meaningful comparison of functions, instead treating any two functions as dissimilar.

6.2.3 Execution

Now we are finally ready to define the execution of instructions. Let us start with the type,

```
run : ∀ {s s′ e e′ f f′ i} → ⟦ s ⟧ˢ → ⟦ e ⟧ᵉ → ⟦ f ⟧ᵈ
                          → ⊢ s # e # f ↝ s′ # e′ # f′
                          → Delay ⟦ s′ ⟧ˢ i
```

Here we say that in order to execute the code

```
⊢ s # e # f ↝ s′ # e′ # f′
```

we require realizations of the stack s, environment e, and function dump f. We return the stack the code stops execution in, wrapped in the Delay monad in order to allow for non-structurally inductive and — possibly non-terminating — calls that will be necessary in some cases.

We proceed by structural induction on the last argument, i.e., the code. We start with the empty run,

```
run s e d ∅ = now s
```

In the case of an empty run, it holds that $s = s'$ and so we simply finish the execution, returning the current stack.

Next we consider all the cases when the run is not empty. We start with the instruction ldf,

```
run s e d (ldf code >> r) =
  run (⟦ code ⟧ᶜ×⟦ e ⟧ᵉ×⟦ d ⟧ᵈ , s) e d r
```

Recall that this instruction is supposed to load a function. Since the semantical meaning of a function is a closure, this is what we must construct. We do so out of the code, given as an argument to ldf, and the current environment and function dump. We use the constructor of Closure, `⟦_⟧ᶜ×⟦_⟧ᵉ×⟦_⟧ᵈ`. We put this closure on the stack and proceed with execution of the rest of the run.

```
run s e d (ld at >> r) = run (lookupᵉ e at , s) e d r
```

This instruction loads a value from the environment with the help of the auxiliary function `lookupᵉ` and puts it on the stack.

```
run s e d (ldc const >> r) = run (makeConst const , s) e d r
  where makeConst : (c : Const) → ⟦ typeof c ⟧ᵗ
        makeConst (bool x) = x
        makeConst (int x)  = x
```

In order to load a constant, we introduce an auxiliary conversion function for converting from an embedded constant to a semantical value. The constant is then put on the stack.

```
run s e d (ldr at >> r) =
  run (lookupᵈ d at , s) e d r
```

This instruction loads a closure from the function dump and puts it on the stack. Similarly to ld, we use an auxiliary function, in this case lookup^{d}.

Next, we handle the instruction for function application, employing do-syntax,

```
run (a , ⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , s) e d (ap >> r) =
  later
    λ where
      .force →
        do
          (b , _) ← run ·
                        (a , fE)
                        (⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , dump)
                        code
          run (b , s) e d r
```

Here we have to employ coinduction for the first time, as we need to perform a call to run that is not structurally recursive. This call is used to evaluate the closure, starting from the empty stack ·, in environment fE extended with the function argument a. The function dump also needs to be extended with the closure being evaluated in order to allow recursive calls. Once the call to run has returned, we grab the first item on the stack b and resume execution of the rest of the run r with b put on the stack.

We now handle recursive tail calls, i.e., the instruction rap. We need to make an additional case split here on the rest of the run r, as a tail call can really only occur if rap is the last instruction in the current run. However, there is no syntactic restriction that would prevent more instructions to follow a rap.

```
run (a , ⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , s) e d (rap >> ∅) =
  later
    λ where
      .force →
        run · (a , fE) (⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , dump) code
```

Above is the case when we can perform the tail call. In this case, the types align: recall that in the type signature of run we promised to return a stack of the type s'. Here, as rap is the last instruction, this means a stack containing a single item of some type β. This is exactly what the closure on the stack constructs. Hence, we can shift execution to the closure.

If there are more instructions after rap, we are not so lucky: here we don't know what s' is, and we have but one way to obtain a stack of this type: proceed with evaluating the rest of the run r. As such, we are unable to perform a tail call. Thus, we side-step the problem by converting rap to ap, performing the standard function application.

```
run (a , ⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , s) e d (rap >> r) =
  later
    λ where
      .force →
        run (a , ⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , ·) e d (ap >> r)
```

This approach also has the advantage of being able to use the instruction rap indiscriminately instead of ap in all situations, at the cost of delaying the execution slightly. However, as we discover in Section 6.3, this is hardly necessary.

As an aside, we could discard the instruction rap from the formalization altogether and implement tail call optimization directly in the ap instruction in the case when the rest of the run is empty. However, the author feels that the presented approach is more instructive to the real workings of the machine. Consider that in a hypothetical hardware implementation we may be unable to examine the following instruction in such a straightforward manner as above. Instead, rap would simply discard the remaining instructions in the control register.

Next, we have the rtn instruction, which simply drops all items from the stack but the topmost one. Once again, we have no guarantee that there are no more instructions after rtn, hence we make a recursive call to run. Under normal circumstances, r is the empty run ∅ and the execution returns the stack here constructed.

```
run (b , _) e d (rtn >> r) = run (b , ·) e d r
```

The if instruction follows,

```
run (test , s) e d (if c₁ c₂ >> r) with test
... | tt = later λ where .force → run s e d (c₁ >+> r)
... | ff = later λ where .force → run s e d (c₂ >+> r)
```

This instruction examines the boolean value on top of the stack and prepends the correct branch to r.

The instructions that remain are those implementing primitive operations that only manipulate the stack. We include them here for completeness' sake.

```
run s e d (nil >> r)              = run ([] , s) e d r
run (x , y , s) e d (flp >> r)    = run (y , x , s) e d r
run (x , xs , s) e d (cons >> r)  = run (x :: xs , s) e d r
run ([] , s) e d (head >> r)      = never
run (x :: _ , s) e d (head >> r)  = run (x , s) e d r
run ([] , s) e d (tail >> r)      = never
run (_ :: xs , s) e d (tail >> r) = run (xs , s) e d r
run (x , y , s) e d (pair >> r)   = run ((x , y) , s) e d r
run ((x , _) , s) e d (fst >> r)  = run (x , s) e d r
run ((_ , y) , s) e d (snd >> r)  = run (y , s) e d r
run (x , y , s) e d (add >> r)    = run (x + y , s) e d r
run (x , y , s) e d (sub >> r)    = run (x − y , s) e d r
run (x , y , s) e d (mul >> r)    = run (x ∗ y , s) e d r
run (a , b , s) e d (eq? >> r)    = run (compare a b , s) e d r
run (x , s) e d (nt >> r)         = run (not x , s) e d r
```

The only interesting cases here are head and tail when called on an empty list. In this case, we signal an error by terminating the execution, returning instead an infinitely delayed value with never.

6.2.4 A note regarding the function dump

There is one final remark to be said regarding the above implementation of the semantics. We have not captured the concept of the dump as introduced first in Chapter 5. Rather, we have exploited the Agda calls stack for the dump's purposes. The instruction in question is function application ap, where we use a do block in which we launch execution of the closure using a recursive call to run. We then use the result thus obtained to proceed with execution. In other words, Agda call stack serves as our function dump.

The reason for this approach is twofold. Firstly, this simplifies the formalization slightly, as we avoided having to formalize the dump register. The second reason is more serious: there is no simple approach to extending the above formalization with the dump register. This is due to the typed syntax of code. When writing some typed code, we must treat the ap instruction as having completed in one step, the next instruction after ap must already have access to the result of the function application in question; pretending that the function has already returned. However, when implementing the semantics, we need to perform all the instructions between the execution of ap and the return from the function with rtn. The types do not align here: in syntax, we only care about the type of the return value of ap, whereas in semantics we must also perform all the work of the function before returning.

In other words, it appears that there is no simple way to extend the above approach to also formalize the dump register. This is however an interesting problem that may be worthy of future considerations.

6.2.5 Tests

Being done with the trickiest part, we now define an auxiliary function for use in tests. It takes some code which starts from an empty initial state. In addition, there is a second argument, which signifies an upper bound on the number of indirections that may be encountered during execution. If this bound is exceeded, nothing is returned.

```
runN : ∀ {x s} → ⊢ [] # [] # [] ↝ (x ∷ s) # [] # []
                 → ℕ
                 → Maybe ⟦ x ⟧ᵗ
runN c n = runFor n
  do
    (x , _) ← run · · · c
    now x
```

Here we made use of runFor defined in Section 3.4.2.

Now for the promised tests, we evaluate the examples from 6.1.5.

```
_ : runN 2+3 0 ≡ just (+ 5)
_ = refl
```

```
_ : runN inc2 1 ≡ just (+ 3)
_ = refl
```

```
_ : runN λTest 2 ≡ just (+ 3)
_ = refl
```

So far, so good! The second argument to runN was chosen the lowest possible and denotes the number of indirections encountered in the form of later.

Now for something more complicated, we foldl the list $[1,2,3,4]$ with plus and the initial accumulator 0. Below we have the code to achieve this,

```
foldTest : ⊢ [] # [] # [] ⇝ [ intT ] # [] # []
foldTest =
  foldl
  >> plus
  >> ap
  >> ldc (int (+ 0))
  >> ap
  >> loadList (1 :: 2 :: 3 :: 4 :: [])
  >+> ap
  >> ∅
```

And indeed,

```
_ : runN foldTest 28 ≡ just (+ 10)
_ = refl
```

Let us also test the mapping function `map`, employed here to increment each element of the list $[1, 2, 3]$ by 1,

```
mapTest : ⊢ [] # [] # [] ⇝ [ listT intT ] # [] # []
mapTest =
  map
  >> plus
  >> ldc (int (+ 1))
  >> ap
  >> ap
  >> loadList (1 :: 2 :: 3 :: [])
  >+> ap
  >> ∅
```

```
_ : runN mapTest 13 ≡ just ((+ 2) :: (+ 3) :: (+ 4) :: [])
_ = refl
```

6.3 Compilation from a high-level language

As a final step, we define a typed λ calculus and implement compilation to typed SECD instructions defined in previous sections.

6.3.1 Syntax

We reuse the types defined in Section 6.1.2. This will not only make compilation cleaner, but also makes sense from a moral standpoint: we want our λ calculus to model the capabilities of our SECD machine. Hence, a context is a list of (SECD) types,

```
Ctx = List Type
```

As for the typing relation, we use a similar trick as with the SECD function dump to allow recursive calls. We keep two contexts, Γ for tracking variables, as in Section 4.2.1, and Ψ for tracking types of functions we can call recursively.

```
data _×_⊢_ : Ctx → Ctx → Type → Set where
  var : ∀ {Ψ Γ x} → x ∈ Γ → Ψ × Γ ⊢ x
  λ_  : ∀ {Ψ Γ α β} → (α ⇒ β :: Ψ) × α :: Γ ⊢ β → Ψ × Γ ⊢ α ⇒ β
  _$_ : ∀ {Ψ Γ α β} → Ψ × Γ ⊢ α ⇒ β → Ψ × Γ ⊢ α → Ψ × Γ ⊢ β
```

The first three typing rules resemble closely the ones from Section 4.2.1, with the addition of the function context Ψ.

Next, we have a variation of var for loading functions from Ψ,

```
rec : ∀ {Ψ Γ α β} → (α ⇒ β) ∈ Ψ → Ψ × Γ ⊢ α ⇒ β
```

We also have an if-then-else construct and a polymorphic comparison operator,

```
if_then_else_ : ∀ {Ψ Γ α} → Ψ × Γ ⊢ boolT
                          → Ψ × Γ ⊢ α → Ψ × Γ ⊢ α
                          → Ψ × Γ ⊢ α
_==_ : ∀ {Ψ Γ α} → Ψ × Γ ⊢ α → Ψ × Γ ⊢ α → Ψ × Γ ⊢ boolT
```

Finally, we have the integers and some primitive operations on them,

```
#_ : ∀ {Ψ Γ} → Z → Ψ × Γ ⊢ intT
_+_ _∗_ _-_ : ∀ {Ψ Γ} → Ψ × Γ ⊢ intT → Ψ × Γ ⊢ intT
                      → Ψ × Γ ⊢ intT
```

We also define the shorthand operator #⁺_ for embedding Agda natural numbers into #_,

```
#⁺_ : ∀ {Ψ Γ} → N → Ψ × Γ ⊢ intT
#⁺ n = # (+ n)
```

As an example, consider the factorial function in this formalism,

```
fac : [] × [] ⊢ (intT ⇒ intT)
fac = λ if (var 0 == #⁺ 1)
          then #⁺ 1
          else (var 0 ∗ (rec 0 $ (var 0 - #⁺ 1)))
```

6.3.2 Compilation

For the compilation, we use a scheme of two mutually recursive functions adapted from [29]. The first function, compileT, is used to compile expressions in the *tail position*, whereas compile is used for the other cases. Expression in a tail position is one which is final in a function. A recursive call in tail position can be optimized so that the call does not consume any additional stack space.

```
mutual
  compileT : ∀ {Ψ Γ α β} → (α ⇒ β :: Ψ) × (α :: Γ) ⊢ β
                         → ⊢ [] # (α :: Γ) # (α ⇒ β :: Ψ)
                           ↝ [ β ] # (α :: Γ) # (α ⇒ β :: Ψ)
  compileT (f $ x) =
    compile f >+> compile x >+> rap >> ∅
  compileT (if t then a else b) =
    compile t >+> if (compileT a) (compileT b) >> ∅
  compileT t = compile t >+> rtn >> ∅
```

The type of compileT reflects the fact that this function is called on the opened body of a lambda.

```
compile : ∀ {Ψ Γ α s} → Ψ × Γ ⊢ α
                      → ⊢ s # Γ # Ψ ↝ (α :: s) # Γ # Ψ
compile (var x)  = ld x >> ∅
compile (λ t)    = ldf (compileT t) >> ∅
compile (f $ x)  = compile f >+> compile x >+> ap >> ∅
compile (rec x)  = ldr x >> ∅
compile (if t then a else b) =
  compile t >+> if (compile a) (compile b) >> ∅
compile (a == b) = compile b >+> compile a >+> eq? >> ∅
compile (# x)      = ldc (int x) >> ∅
compile (a + b) = compile b >+> compile a >+> add >> ∅
compile (a * b)    = compile b >+> compile a >+> mul >> ∅
compile (a – b)    = compile b >+> compile a >+> sub >> ∅
```

The above is a fairly straighforward compilation process where we make a call to compileT in the compilation of a lambda. In the other cases we construct the corresponding sequence of SECD instructions.

The function compileT is to be used when an expression is in the tail position. Whether tail call optimization occurs depends on the exact nature of this expression. In case of function application, we may indeed perform the optimization. In the case of an if-then-else construct, the branches therein may also receive tail call optimization, and so we recurse. In all other cases, the expression in question loses the option of tail call optimization. We compile the expression normally and append the rtn instruction.

We now compile the above definition of fac. Below is the result, adjusted for readability:

```
_ : compile {s = []} fac ≡ ldf (
      ldc (int (+ 1)) >> ld 0 >> eq?
   >| if (ldc (int (+ 1)) >| rtn)
        (ldr 0
        >> ldc (int (+ 1))
        >> ld 0
        >> sub
        >> ap
        >> ld 0
        >> mul
        >| rtn)
   ) >> ∅
_ = refl
```

As a final test, we apply the function fac to the number 5, compile the expression, and evaluate it on the SECD,

```
_ : runN (compile (fac $ #⁺ 5)) 10 ≡ just (+ 120)
_ = refl
```

7 Epilogue

> *I walk slowly, like one who comes from so far away he doesn't expect to arrive.*
>
> — Jorge Luis Borges, *Selected poems*

We succeeded in formalizing syntax and semantics of a version of the SECD machine with typed code. We employed Agda as a tool for writing this formalization, using dependent types to make only well-typed SECD code representable, and using coinduction to handle the Turing-complete semantics.

As for the practical usability of this implementation, it would be possible to produce an executable binary file implementing the semantics, as Agda compiles to Haskell. In addition, there is also an experimental JavaScript compiler.

As a final remark, the author would like to note that the above implementation of semantics did not require any sort of de-bugging: as soon as the Agda type-checker was satisfied, all the tests succeeded without any further need of modification to the semantics. This suggests that dependent types could be a valuable tool in the implementation of safety-critical systems, a direction perhaps worthy of further pursuit.

Bibliography

[1] Andreas Abel and James Chapman. “Normalization by Evaluation in the Delay Monad: A Case Study for Coinduction via Copatterns and Sized Types”. In: (2014). DOI: `10.4204/EPTCS.153.4`. eprint: `arXiv:1406.2059`.

[2] Andreas Abel and Habilitation Talk. “Normalization by Evaluation: Dependent Types and Impredicativity”. In: *Unpublished. http://www. tcs. ifi. lmu. de/~ abel/habil. pdf* (2013).

[3] Bruno Barras et al. “The Coq proof assistant reference manual: Version 6.1”. PhD thesis. Inria, 1997.

[4] Gérard Boudol. “Towards a lambda-calculus for concurrent and communicating systems”. In: *Colloquium on Trees in Algebra and Programming*. Springer. 1989, pp. 149–161.

[5] Luitzen Egbertus Jan Brouwer. “On the foundations of mathematics”. In: *Collected works* 1 (1907), pp. 11–101.

[6] Luitzen Egbertus Jan Brouwer. “The unreliability of the logical principles”. In: *Brouwer* (1908).

[7] NG de Bruijn. “Description of the language Automath”. In: (1967).

[8] Alonzo Church. “A formulation of the simple theory of types”. In: *The journal of symbolic logic* 5.2 (1940), pp. 56–68.

[9] Alonzo Church. “A set of postulates for the foundation of logic”. In: *Annals of mathematics* (1932), pp. 346–366.

[10] Catarina Coquand. “A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions”. In: *Higher-Order and Symbolic Computation* 15.1 (2002), pp. 57–90.

[11] Richard E Crandall. “On the “3□+ 1” problem”. In: *Mathematics of computation* 32.144 (1978), pp. 1281–1292.

[12] Nils Anders Danielsson and Ulf Norell. “Parsing Mixfix Operators”. In: *Implementation and Application of Functional Languages*. Ed. by Sven-Bodo Scholz and Olaf Chitil. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 80–99. ISBN: 978-3-642-24452-0.

[13] Olivier Danvy. “A rational deconstruction of Landin’s SECD machine”. In: *Symposium on Implementation and Application of Functional Languages*. Springer. 2004, pp. 52–71.

[14] Nicolaas Govert De Bruijn. “Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem”. In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[15] Radu Diaconescu. “Axiom of choice and complementation”. In: *Proceedings of the American Mathematical Society* 51.1 (1975), pp. 176–178.

[16] Daniel P Friedman and David Thrane Christiansen. *The Little Typer*. MIT Press, 2018.

[17] Gerhard Gentzen. “Untersuchungen über das logische Schließen. I”. In: *Mathematische zeitschrift* 39.1 (1935), pp. 176–210.

[18] Jean-Yves Girard. “Linear logic”. In: *Theoretical computer science* 50.1 (1987), pp. 1–101.

[19] Brian Graham. “Secd: Design issues”. In: (1989).

[20] Brian Graham and Graham Birtwistle. “Formalising the design of an SECD chip”. In: *Hardware Specification, Verification and Synthesis: Mathematical Aspects*. Ed. by Miriam Leeser and Geoffrey Brown. New York, NY: Springer New York, 1990, pp. 40–66. ISBN: 978-0-387-34801-8.

[21] William A Howard. “The formulae-as-types notion of construction”. In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[22] Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016. DOI: `10.1017/CBO9781316823187`.

[23] Matt Kaufmann and J Strother Moore. “ACL2: An industrial strength version of Nqthm”. In: *Computer Assurance, 1996. COMPASS’96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*. IEEE. 1996, pp. 23–34.

[24] Stephen C Kleene and J Barkley Rosser. "The inconsistency of certain formal logics". In: *Annals of Mathematics* (1935), pp. 630–636.

[25] Pepijn Kokke and Wouter Swierstra. "Auto in Agda". In: *Mathematics of Program Construction*. Ed. by Ralf Hinze and Janis Voigtländer. Cham: Springer International Publishing, 2015, pp. 276–301. ISBN: 978-3-319-19797-5.

[26] Jean-Louis Krivine. "A call-by-name lambda-calculus machine". In: *Higher-order and symbolic computation* 20.3 (2007), pp. 199–207.

[27] Peter J Landin. "The mechanical evaluation of expressions". In: *The Computer Journal* 6.4 (1964), pp. 308–320.

[28] Peter J Landin. "The next 700 programming languages". In: *Communications of the ACM* 9.3 (1966), pp. 157–166.

[29] Xavier Leroy. "Functional programming languages, Part II: abstract machines". 2015–2016. URL: `https://xavierleroy.org/mpri/2-4/machines.pdf`.

[30] Xavier Leroy. "The ZINC experiment: an economical implementation of the ML language". PhD thesis. INRIA, 1990.

[31] Per Martin-Löf. "An intuitionistic theory of types: Predicative part". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, 1975, pp. 73–118.

[32] Conor McBride. "Turing-completeness totally free". In: *International Conference on Mathematics of Program Construction*. Springer. 2015, pp. 257–275.

[33] Joan Moschovakis. "Intuitionistic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2018. Metaphysics Research Lab, Stanford University, 2018.

[34] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[35] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, 2007.

[36] Ulf Norell et al. *Agda Language Reference*. Version 2.5.4.2. 2018. URL: `https://agda.readthedocs.io/en/v2.5.4.2/language/index.html`.

[37] Bertrand Russell. "Correspondence with Frege". In: (1982). Translated by Hans Kaal. Original from 1901.

[38] Aaron Stump. *Verified functional programming in Agda*. Morgan & Claypool, 2016.

[39] Anne Sjerp Troelstra. "History of constructivism in the 20th century". In: *Set Theory, Arithmetic, and Foundations of Mathematics* (2011), pp. 150–179.

[40] Alan M Turing. "Computability and λ-definability". In: *The Journal of Symbolic Logic* 2.4 (1937), pp. 153–163.

[41] André Van Tonder. "A lambda calculus for quantum computation". In: *SIAM Journal on Computing* 33.5 (2004), pp. 1109–1135.

[42] Vladimir Voevodsky. "Univalent foundations of mathematics". In: *International Workshop on Logic, Language, Information, and Computation*. Springer. 2011, pp. 4–4.

[43] John Von Neumann. "On the introduction of transfinite numbers". In: (2002). Translated by Jean van Heijenoort. Original from 1923., pp. 346–354.

[44] Philip Wadler. "Programming Language Foundations in Agda". In: *Brazilian Symposium on Formal Methods*. Springer. 2018, pp. 56–73.

[45] Alfred North Whitehead and Bertrand Russell. *Principia mathematica*. Vol. 1. University Press, 1910.