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

These are the acknowledgements for my thesis, which can span multiple paragraphs.

Abstract

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

Keywords

SECD Agda formalization coinduction

Contents

List of Tables

1 Introduction

2 Intuitionistic logic

What I cannot create, I do not understand.

— R. Feynman

2.1 History

2.2 Curry-Howard correspondence

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[14] 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 correspoinding 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 [6] for an introduction to dependent types as a whole, or [16] for an in-depth introduction to dependendly typed programming and theorem proving in Agda.

3.1 Overview

Due to the presence of dependent types, all functions defined must be provably terminating. Failure to do so would result in type-checking becoming undecidable. However, this does not mean the loss of Turing-completeness; indeed we will see in section 3.5 how possibly non-terminating computations can still be expressed, with some help from the type system.

Agda has strong support for mixfix operators[1] and Unicode identifiers. This often allows for developing a notation close to what one has come to expect in mathematics. However, with great power comes great responsibility and one should be careful to not abuse the notation too much, a problem exacerbated by the fact that operator overloading, as used excessively in mathematics, is not directly possible.

1. Operators which can have multiple name parts and are infix, prefix, postfix, or closed[3].

As an aside, there is also some support for proof automation in Agda[9], however from the author's experience the usability of this tool is limited to simple cases. In contrast with tools such as Coq[2], Agda suffers from lower degree of automation: there are no built-in tactics, though their implementation is possible through reflection[15].

3.1.1 Trivial Types

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

```
data ⊤ : Set where
  · : ⊤
```

declares the new data type ⊤ which is itself of type Set[2]. The second line declared a constructor for this type, here called simply ·, which constructs a value of type ⊤[3].

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

```
data ⊥ : Set where
```

Note how there are no constructors declared for this type, therefore it is clearly uninhabited.

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

```
¬_ : Set → Set
¬ P = P → ⊥
```

3.1.2 Booleans

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

2. 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.
3. Again for the Haskell-able, note how the syntax here resembles that of Haskell with the extension `GADTs`.

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

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

We can also declare our first function now, one that will perform negation of Boolean values.

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

Here we utilized pattern matching to split on the argument and flipped one into the other. Note the underscore _ in the name declaration of this function: it symbolizes where the argument is to be expected and declares it as a mixfix operator.

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
infixr 4 _,_
```

Here we declared 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 _,_. We also declare the fixity of this operator to be right-associative.

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 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 to some number a, we inductively perform addition of a to b, and then apply the successor function.

3.2 Propositional Equality

In this section, we will 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, definitionaly it holds that $2 = 2$, however, $1 + 1 = 2$ only holds propositionaly, 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 propositionaly equal to itself. 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 case splitting on the arguments we force Agda to unify the variables a, b, and c. Afterwards, we can construct the required proof with the refl constructor. This is a feature of the underlying type theory of Agda.

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
```

TODO: If this is to be kept here, explain.

3.3 Decidable Equality

A different concept of 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 which are decidable. Firstly, we will 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. 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,

```
_≟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 Formalizing Type Systems

In what follows, we will take a look at how we can use Agda to formalize deductive systems. We will take the simplest example there is, the Simply Typed λ Calculus. Some surface-level knowledge of this calculus is assumed.

3.4.1 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. Instead, we shall use the concept commonly referred to as De Bruijn indices[5]. These replace variable names with natural numbers, where each number n refers to the variable bound by the binder n positions above the current scope in the syntactical tree. Some examples of this naming scheme are shown in Figure 3.1. The immediately apparent advantage of us-

Literal 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)`
`λf.(λx.f(x x)) (λx.f(x x))`	`λ(λ 1 (0 0)) (λ 1 (0 0))`

Figure 3.1: Examples of λ terms using standard naming scheme on the left and using De Bruijn indices on the right.

ing De Bruijn indices is that α-equivalence of λ terms becomes trivially decidable by way of purely syntactic equality. Other advantages include easier formalization.

Implementation

To implement De Bruijn indices in Agda, we will express what it means for a variable to be present in a context. We shall assume that a context is a list of types, 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)
infix 10 _∈_
```

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.

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
```

We will also define shorthands to construct often-used elements of _∈_ for use in examples later on.

```
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)
```

3.4.2 Example: Simply Typed λ Calculus

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

Syntax

First, we define the types in our system.

```
data ⋆ : Set where
  ι : ⋆
  _⇒_ : ⋆ → ⋆ → ⋆
infixr 20 _⇒_
```

Here we defined some atomic type ι and a binary type constructor for function types. We proceed by defining context as a list of types.

```
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
  _$_ : ∀ {Γ α β} → Γ ⊢ α ⇒ β → Γ ⊢ α → Γ ⊢ β
infix 4 _⊢_
infixr 5 λ_
infixl 10 _$_
```

The constructors above should be fairly self-explanatory: they correspond exactly to the typing rules of the calculus. In the first rule we employed the data type _∈_ implenting De Bruijn indices. Second rule captures the concept of λ-abstraction, and the last rule is function application.

We can see some examples now,

```
I : ∀ {Γ α} → Γ ⊢ α ⇒ α
I = λ (var 0)

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 polymorhism 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 will see an example of this in what follows.

Semantics by Embedding into Agda

Now that we have defined the syntax, the next step is to give it semantics. We will 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.

$$
\begin{aligned}
&[\![\_]\!]\star : \star \to \mathsf{Set}\\
&[\![\ \iota\ ]\!]\star = \mathbb{N}\\
&[\![\ \alpha \Rightarrow \beta\ ]\!]\star = [\![\ \alpha\ ]\!]\star \to [\![\ \beta\ ]\!]\star
\end{aligned}
$$

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 inductively as an Agda function type.

Next, we give semantics to contexts.

$$
\begin{aligned}
&[\![\_]\!]\mathsf{C} : \mathsf{Context} \to \mathsf{Set}\\
&[\![\ [\,]\ ]\!]\mathsf{C} = \top\\
&[\![\ x :: xs\ ]\!]\mathsf{C} = [\![\ x\ ]\!]\star \times [\![\ xs\ ]\!]\mathsf{C}
\end{aligned}
$$

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.

$$
[\![\_]\!] : \forall\ \{\Gamma\ \alpha\} \to \Gamma \vdash \alpha \to [\![\ \Gamma\ ]\!]\mathsf{C} \to [\![\ \alpha\ ]\!]\star
$$

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

$$
\begin{aligned}
&[\![\ \mathsf{var\ here}\ ]\!]\ (x\ ,\ \_) = x\\
&[\![\ \mathsf{var}\ (\mathsf{there}\ x)\ ]\!]\ (\_\ ,\ xs) = [\![\ \mathsf{var}\ x\ ]\!]\ xs
\end{aligned}
$$

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.

$$[\![\ \lambda\ x\ ]\!]\ \gamma = \lambda\ [\![\alpha]\!] \to [\![\ x\ ]\!]\ ([\![\alpha]\!]\ ,\ \gamma)$$

The case for lambda abstraction constructs an Agda function which will take 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.

$$[\![\ f\ \$\ x\ ]\!]\ \gamma = ([\![\ f\ ]\!]\ \gamma)\ ([\![\ x\ ]\!]\ \gamma)$$

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.

```
IN : N → N
IN x = x

_ : ⟦ I ⟧ · ≡ IN
_ = 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 function as expected!

3.5 Coinduction

[1]

3.5.1 The Delay Monad

```
open import Size
open import Data.Maybe

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
open ∞Delay public

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

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

_>>=_ : ∀ {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
```

4 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 [12], 1966

4.1 Introduction

The **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 [11]. 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 have since been proposed, some derived from SECD, others not. Notable are the Krivine machine[10], 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 [13].

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 version of the machine, and a more modern version of the machine which foregoes the dump in favour of using the stack for the purposes of the dump, see [4].

There have also been hardware implementations of this formalism, e.g. [7, 8], though it is unclear to the author whether the issue with verifying the garbage collector mentioned in the latter work was ever fully addressed.

This chapter is meant as an intuitive overview of the formalism. We will present the machine with the standard call-by-value semantics.

4.2 Definition

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

- Stack – stores values operated on. Atomic operations, such as integer addition, are performed here;
- Environment – stores immutable assignments, such as function arguments and values bound with the *let* construct;
- Control – stores a list of instructions awaiting execution;
- Dump – serves as a baggage place for storing the current context when a function call is performed.

Regarding the memory model, all four items defined here are meant to be realized as linked lists.

4.3 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;
- `ap` — given that a closure and a value are present on the top of the stack, perform function application and put the return value on the stack;
- `rtn` — return from a function, restoring control to the caller.

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 under the identifier x.

To see how the basic instructions and the addition instruction transform the machine state, please refer to Figure 4.1.

To see an example of execution of the machine, please refer to Figure 4.2.

It is usual to use 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.

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 4.1: 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.

S	E	C	D
$\emptyset$	$\emptyset$	`ldf f, ldc 1, ap, ldc 3, add`	$\emptyset$
$f[\emptyset]$	$\emptyset$	`ldc 1, ap, ldc 3, add`	$\emptyset$
$1, f[\emptyset]$	$\emptyset$	`ap, ldc 3, add`	$\emptyset$
$\emptyset$	1	`ldc 1, ld 0, add, rtn`	($\emptyset$,$\emptyset$,`ldc 3, add`)
1	1	`ld 0, add, rtn`	($\emptyset$,$\emptyset$,`ldc 3, add`)
$1, 1$	1	`add, rtn`	($\emptyset$,$\emptyset$,`ldc 3, add`)
2	1	`rtn`	($\emptyset$,$\emptyset$,`ldc 3, add`)
2	$\emptyset$	`ldc 3, add`	$\emptyset$
$3, 2$	$\emptyset$	`add`	$\emptyset$
5	$\emptyset$	$\emptyset$	$\emptyset$

Figure 4.2: 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`.

5 Formalization

In this chapter, we approach the main topic of this thesis. We will formalize a SECD machine in Agda, with typed syntax, and then proceed to define the semantics by way of coinduction. Finally, we will define a typed λ calculus, corresponding exactly to the capabilities of the SECD machine, and define a compilation procedure from this calculus to typed SECD programs.

5.1 Syntax

5.1.1 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.

```
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
infixr 5 _>>_
```

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 a certain similarity to linked lists; indeed if for the relation we take the universal one for our data type A, we stand to obtain a type that's 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 will 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 matches the start of the second one.

```
_>+>_ : ∀ {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)
infixr 4 _>+>_
```

5.1.2 Machine types

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

```
data Const : Set where
  true false : Const
  int : ℤ → Const
```

Next, we define which types our machine recognizes.

```
data Type : Set where
  intT boolT : Type
  pairT : Type → Type → Type
  listT : Type → Type
  _⇒_ : Type → Type → Type
infixr 15 _⇒_
```

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 true  = boolT
typeof false = boolT
typeof (int x) = 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 ?.

Finally, we define the state as a record storing the stack, environment, and the function dump.

```
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 ?, here the state does not include the code. This is because we are aiming for a version of SECD with typed assembly code. We will define code next

5.1.3 Typing relation

Since we aim to have typed assembly, we have to take a different approach to defining code. We will define a binary relation which will determine how a state of a certain shape is mutated following the execution of an instruction.

We will have two versions of this relation: first one is the single-step relation, the second one is the transitive closure of the first one 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
```

Here there is nothing surprising, 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 will 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 constant on the stack.

```
ld : ∀ {s e f a}
   → (a ∈ e)
   → ⊢ s # e # f ▷ (a :: s) # e # f
```

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.

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

The ldf instruction is considerably more involved. It loads a function of the type $a \Rightarrow b$ and puts it on the stack. Note how we use the multi-step relation here. In addition, the code we are loading also has to be of a certain shape to make it a function: the argument it was called with must be put in the environment, and the function dump is to be extended with the type of the function to permit recursive calls to itself.

Once a function is loaded, we may apply it,

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

ap requires that a function and its argument are on the stack. After it has run, the returning 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 implementation.

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

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.

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

ldr loads a function for 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.

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

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.

$$\begin{aligned}&\mathsf{if} : \forall\ \{s\ s'\ e\ f\}\\&\quad \to \vdash s\ \#\ e\ \#\ f \rightsquigarrow s'\ \#\ e\ \#\ f\\&\quad \to \vdash s\ \#\ e\ \#\ f \rightsquigarrow s'\ \#\ e\ \#\ f\\&\quad \to \vdash (\mathsf{boolT} :: s)\ \#\ e\ \#\ f \rhd s'\ \#\ e\ \#\ f\end{aligned}$$

The if instruction requires that a boolean value be present on the stack. Based on this, 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.

The remaining instructions are fairly simple in that they only manipulate the stack. Maybe we will show you only a few of them and hide the rest later.

$$
\begin{aligned}
&\mathsf{lett} : \forall\ \{s\ e\ f\ x\} \\
&\quad \to \vdash (x :: s)\ \#\ e\ \#\ f \rhd s\ \#\ (x :: e)\ \#\ f \\
&\mathsf{nil} \quad : \forall\ \{s\ e\ f\ a\} \\
&\quad \to \vdash s\ \#\ e\ \#\ f \rhd (\mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \\
&\mathsf{flp} \quad : \forall\ \{s\ e\ f\ a\ b\} \\
&\quad \to \vdash (a :: b :: s)\ \#\ e\ \#\ f \rhd (b :: a :: s)\ \#\ e\ \#\ f \\
&\mathsf{cons} : \forall\ \{s\ e\ f\ a\} \\
&\quad \to \vdash (a :: \mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \rhd (\mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \\
&\mathsf{head} : \forall\ \{s\ e\ f\ a\} \\
&\quad \to \vdash (\mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \rhd (a :: s)\ \#\ e\ \#\ f \\
&\mathsf{tail} : \forall\ \{s\ e\ f\ a\} \\
&\quad \to \vdash (\mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \rhd (\mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \\
&\mathsf{pair} : \forall\ \{s\ e\ f\ a\ b\} \\
&\quad \to \vdash (a :: b :: s)\ \#\ e\ \#\ f \rhd (\mathsf{pairT}\ a\ b :: s)\ \#\ e\ \#\ f \\
&\mathsf{fst} \quad : \forall\ \{s\ e\ f\ a\ b\} \\
&\quad \to \vdash (\mathsf{pairT}\ a\ b :: s)\ \#\ e\ \#\ f \rhd (a :: s)\ \#\ e\ \#\ f \\
&\mathsf{snd} \quad : \forall\ \{s\ e\ f\ a\ b\} \\
&\quad \to \vdash (\mathsf{pairT}\ a\ b :: s)\ \#\ e\ \#\ f \rhd (b :: s)\ \#\ e\ \#\ f \\
&\mathsf{add} \quad : \forall\ \{s\ e\ f\} \\
&\quad \to \vdash (\mathsf{intT} :: \mathsf{intT} :: s)\ \#\ e\ \#\ f \rhd (\mathsf{intT} :: s)\ \#\ e\ \#\ f \\
&\mathsf{sub} \quad : \forall\ \{s\ e\ f\} \\
&\quad \to \vdash (\mathsf{intT} :: \mathsf{intT} :: s)\ \#\ e\ \#\ f \rhd (\mathsf{intT} :: s)\ \#\ e\ \#\ f \\
&\mathsf{mul} \quad : \forall\ \{s\ e\ f\} \\
&\quad \to \vdash (\mathsf{intT} :: \mathsf{intT} :: s)\ \#\ e\ \#\ f \rhd (\mathsf{intT} :: s)\ \#\ e\ \#\ f \\
&\mathsf{eq?} \quad : \forall\ \{s\ e\ f\ a\} \\
&\quad \to \vdash (a :: a :: s)\ \#\ e\ \#\ f \rhd (\mathsf{boolT} :: s)\ \#\ e\ \#\ f \\
&\mathsf{nt} \quad : \forall\ \{s\ e\ f\} \\
&\quad \to \vdash (\mathsf{boolT} :: s)\ \#\ e\ \#\ f \rhd (\mathsf{boolT} :: s)\ \#\ e\ \#\ f
\end{aligned}
$$

Derived instructions

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

$$\begin{aligned}&\mathsf{nil?} : \forall\ \{s\ e\ f\ a\} \to\ \vdash (\mathsf{listT}\ a :: s)\ \#\ e\ \#\ f \leadsto (\mathsf{boolT} :: s)\ \#\ e\ \#\ f\\&\mathsf{nil?} = \mathsf{nil} >|\ \mathsf{eq?}\end{aligned}$$

$$\begin{aligned}&\mathsf{loadList} : \forall\ \{s\ e\ f\} \to \mathsf{List}\ \mathbb{N} \to\ \vdash s\ \#\ e\ \#\ f \leadsto (\mathsf{listT}\ \mathsf{intT} :: s)\ \#\ e\ \#\ f\\&\mathsf{loadList}\ [\,] = \mathsf{nil} >> \emptyset\\&\mathsf{loadList}\ (x :: xs) = (\mathsf{loadList}\ xs) >+> (\mathsf{ldc}\ (\mathsf{int}\ (+\ x)) >|\ \mathsf{cons})\end{aligned}$$

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.

5.1.4 Examples

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

The first example loads two constants and adds them.

$$\begin{aligned}&\mathsf{2{+}3} :\ \vdash [\,]\ \#\ [\,]\ \#\ [\,] \leadsto [\ \mathsf{intT}\ ]\ \#\ [\,]\ \#\ [\,]\\&\mathsf{2{+}3} =\\&\qquad \mathsf{ldc}\ (\mathsf{int}\ (+\ 2))\\&\quad >> \mathsf{ldc}\ (\mathsf{int}\ (+\ 3))\\&\quad >|\ \mathsf{add}\end{aligned}$$

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

$$\begin{aligned}&\mathsf{inc} : \forall\ \{e\ f\} \to\ \vdash [\,]\ \#\ (\mathsf{intT} :: e)\ \#\ (\mathsf{intT} \Rightarrow \mathsf{intT} :: f)\\&\qquad\qquad\qquad \leadsto [\ \mathsf{intT}\ ]\ \#\ (\mathsf{intT} :: e)\ \#\ (\mathsf{intT} \Rightarrow \mathsf{intT} :: f)\\&\mathsf{inc} =\\&\qquad \mathsf{ld}\ \mathbf{0}\\&\quad >> \mathsf{ldc}\ (\mathsf{int}\ (+\ 1))\end{aligned}$$

```
    >> add
    >| rtn
```

Here we can see the type of the expression getting more complicated: we use polymorphism to make make sure we can load this function in any environment, in the environment we have to declare that an argument of type intT is expected, and lastly the function dump has to be expanded 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 the two values in the environment. The types of these two are inferred to be integers by Agda, as this is what the add instruction requires. Then, we load an 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 load this function in another program, rather than writing ldf plus we must only write plus.

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)
      >> ∅
```

Here is what's going on: 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 it list is not empty, we start with loading the folding function **2**. Next, we load the accumulator **1**. We perform partial application. Next, we load the list **0** and obtain it's first element with head. We apply to the already partially-applied folding function, yielding a new accumulator on the stack.

Now we need to make the recursive call: we load ourselves 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, drop the first element with tail and perform 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 we will see that there is no need for this instruction in section ?

5.2 Semantics

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

  ⟦_⟧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 = ⊥

  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

  ⟦_⟧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

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

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

```
lookupᵉ (_ , xs) (there w) = lookupᵉ xs w

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

--lookupᵈ : ∀ {x xs} → ⟦ xs ⟧ᵈ → x ∈ xs → ⟦ x ⟧ᶜˡ
--lookupᵈ {mkClosureT _ _ _} (x , _) ? = x
--lookupᵈ {mkClosureT _ _ _} list (t? at) = lookupᵈ (tailᵈ list

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

run : ∀ {s s′ e e′ f f′ i} → ⟦ s ⟧ˢ → ⟦ e ⟧ᵉ → ⟦ f ⟧ᵈ → ⊢ s # e # f ⇝ s′ # e′ # f′
                                → Delay ⟦ s′ ⟧ˢ i
run s e d ∅ = now s
run s e d (ldf code >> r) = run (⟦ code ⟧ᶜ×⟦ e ⟧ᵉ×⟦ d ⟧ᵈ , s) e d r
run s e d (ldr at >> r) = run (lookupᵈ d at , s) e d r
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) cod
                              run (b , s) e d r
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
run (a , ⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , s) e d (rap >> x >> r) =
  later λ where .force → run (a , ⟦ code ⟧ᶜ×⟦ fE ⟧ᵉ×⟦ dump ⟧ᵈ , ·) e d (ap >> x >> r)
run (b , _) e d (rtn >> r) = run (b , ·) e d r
run (x , s) e d (lett >> r)       = run s (x , e) d r
run s e d (nil >> r)              = run ([] , s) e d r
run s e d (ldc const >> r)        = run (makeConst const , s) e d r
  where makeConst : (c : Const) → ⟦ typeof c ⟧ᵗ
        makeConst true  = tt
        makeConst false = ff
        makeConst (int x) = x
```

```
run s e d (ld at >> r)           = run (lookupᵉ e at , 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 (x :: 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
  where compare : {t₁ t₂ : Type} → ⟦ t₁ ⟧ᵗ → ⟦ t₂ ⟧ᵗ → ⟦ boolT ⟧ᵗ
        compare {intT} {intT} a b = ⌊ a ≟Z b ⌋
        compare {boolT} {boolT} a b = ⌊ a ≟B b ⌋
        compare {pairT _ _} {pairT _ _} (a₁ , a₂) (b₁ , b₂) = (compare a₁ b₁) ∧ (compare a₂ b₂)
        compare {listT xs} {listT ys} a b = ⌊ length a ≟N length b ⌋ -- BDO
        compare {_} {_} _ _ = ff
run (x , s) e d (nt >> r) = run (not x , s) e d r
run (bool , s) e d (if c₁ c₂ >> r) with bool
... | tt = later λ where .force → run s e d (c₁ >+> r)
... | ff = later λ where .force → run s e d (c₂ >+> r)

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


_ : runN 2+3 1 ≡ just (+ 5)
_ = refl

_ : runN inc2 2 ≡ just (+ 3)
_ = refl
```

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

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

_ : runN foldTest 29 ≡ just (+ 10)
_ = refl
```

5.3 Compilation from a higher-level language

```
Ctx = List Type

infix 2 _×_⊢_
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
  _$_ : ∀ {Ψ Γ α β} → Ψ × Γ ⊢ α ⇒ β → Ψ × Γ ⊢ α → Ψ × Γ ⊢ β
  rec : ∀ {Ψ Γ α β} → (α ⇒ β) ∈ Ψ → Ψ × Γ ⊢ α ⇒ β
  if_then_else_ : ∀ {Ψ Γ α} → Ψ × Γ ⊢ boolT → Ψ × Γ ⊢ α → Ψ × Γ ⊢ α → Ψ × Γ ⊢ α
  _==_ : ∀ {Ψ Γ} → Ψ × Γ ⊢ intT → Ψ × Γ ⊢ intT → Ψ × Γ ⊢ boolT
  #_ : ∀ {Ψ Γ} → Z → Ψ × Γ ⊢ intT
  #⁺_ : ∀ {Ψ Γ} → N → Ψ × Γ ⊢ intT
  mul : ∀ {Ψ Γ} → Ψ × Γ ⊢ intT ⇒ intT ⇒ intT
  sub : ∀ {Ψ Γ} → Ψ × Γ ⊢ intT ⇒ intT ⇒ intT
infixr 2 λ_
infixl 3 _$_
infix 5 _==_
```

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

mutual
  compileT : ∀ {Ψ Γ α β} → (α ⇒ β :: Ψ) × (α :: Γ) ⊢ β → ⊢ [] # (α :: Γ) # (α ⇒ β :: Ψ) ⇝ [ β
  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 >> ∅

  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 (#⁺ x) = ldc (int (+ x)) >> ∅
  compile mul = ldf (ldf (ld 0 >> ld 1 >| mul) >| rtn) >> ∅
  compile sub = ldf (ldf (ld 0 >> ld 1 >| sub) >| rtn) >> ∅

_ : runN (compile (fac $ #⁺ 5)) 27 ≡ just (+ 120)
_ = refl
```

6 Epilogue

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

[3] 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.

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

[5] 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.

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

[7] Brian Graham. "Secd: Design issues". In: (1989).

[8] 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.

[9] 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.

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

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

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

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

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

[15] 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`.

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