Verified Translation of a Strongly Typed Functional Language with Variables to a Language of Indexed Gates

Rob Spoel

2019

Contents

List of Figures

List of Agda listings

1 Abstract

In this thesis, we present a translation from an embedded hardware description language with variable bindings (Λ_1) to an embedded hardware description language without variable bindings (Π-Ware). The host language for these embedded languages is Agda. We take a look at type theory and how it relates to higher order logic according to the Curry-Howard isomorphism, at different ways to implement variable binding and finally at the translation itself. Not only do we show how to translate to a nameless language, but we also present a correctness proof of said translation formalized in Agda itself.

Keywords— Agda, embedded languages, Curry-Howard isomorphism, compiler-correctness

2 Research goal

2.1 Preamble

Domain-specific languages (DSLs) are languages that are specialized for particular applications. They can provide benefits over their counterpart *general-purpose languages* (GPLs) by providing specific structures that let users express certain solutions to certain problems more clearly. A common approach when designing a DSL is to embed it in an existing *host* language (either inside another DSL or more commonly inside a GPL).

When embedding a DSL into a host language, there are multiple possible methods for handling variable binding. This work demonstrates a verified translation from an embedded hardware description DSL which represents variables using named De Bruijn bindings – which are practical for the internal representation of the DSL – to an embedded hardware description DSL which uses nameless bindings of indexed wires – which is practical for the compilation to actual hardware.

2.2 Research statement

Given a hardware description language with variable bindings, can we translate it to a hardware description language without bindings but with input/output indexed wires[1]? Given such a translation, can we prove that the translation is correct[2] for all possible programs that can be written in the source language? How well does a dependently typed context[3] lend itself to reason about hardware description languages?

2.3 Contribution

We start this document with some background in section 3 in the form of literature research on topics such as dependent types, higher-order logic, and embedded languages.

We demonstrate a prototype translation in the form of the *SKI transpiler*, showing that it is possible to translate from a language with bindings (simply-typed λ-calculus) to a nameless language (SKI combinators) in section 4. Furthermore, we demonstrate that we can use Agda's dependent type system to prove the translation's correctness, based on the Curry-Howard isomorphism.

In section 5, we introduce a new hardware description DSL – based on J. P. Pizani Flor's work – called Λ_1 (pronounced *lambda one*, named after the working name of $\lambda\pi$-Ware also by J. P. Pizani Flor). We demonstrate a complete translation from Λ_1 to Π-Ware in section 6 before highlighting some parts of the correctness proof in section 7.

[1] Section 5.2: Plugs versus named variables

[2] Section 3.4: Verified translation

[3] Section 3.1: Dependently typed programming: Agda

2.4 Scientific relevance

As hardware becomes more and more complex, the need for streamlined verification solutions becomes more and more pressing. Finding faults in circuits after their worldwide distribution is a scenario from which it is hard to recover, as the Intel FDIV bug in the mid 90's [Intel, 2004] demonstrates.

With the growing popularity of dependently typed programming languages such as Coq and Agda, there is an opportunity for a new hardware design language solution that can provide more mathematical soundness guarantees for the chips of the future. Verification and validation of hardware design play a considerable part in the cost of hardware design [Rekhi and Purasai, 2003]. Streamlining both of these steps into a single language could provide big savings as well as better scaling for big and small manufacturers alike.

For such a potential language to be attractive to developers, it is important to provide user-friendly abstractions of the underlying wires and plugs. For this reason, we believe it is of scientific significance to research the translating from higher-level user-space languages to lower-level machine-space languages, where a dependently typed context can provide strong mathematical soundness guarantees.

3 Background

3.1 Dependently typed programming: Agda

Agda is a dependently typed programming language based on Martin-Löf type theory [Bove et al., 2009, Martin-Löf, 1984]. It has a functional syntax which is close to that of Haskell. However, its type system is fundamentally different from Haskell's. The term *dependent* in the phrase *dependently typed* refers to the fact that *types* can depend on *values*. This means that, in Agda, types and values can be mixed freely. This is fundamentally different from the paradigm that most computer scientists are more closely familiar with, polymorphism. Polymorphism provides abstraction over *types* in *types*. A familiar example in the programming language Java is the abstract type `java.util.List`. The abstract type can be *specialized*, i.e. turned into a complete type, by passing in another type. A dependent type system provides abstraction over *values* in *types*. Any Agda expression can be used in a type's definition to provide such values. Notably, these expressions can also depend on other terms that came before it in the type definition. This lets programmers define arbitrary constraints on types.

This expressibility comes at a cost, however. All expressions in Agda must be total. For all possible inputs of an expression, the expression must not produce an output which is undefined nor cause an infinite recursion. The Agda type system needs to be able to determine whether two types are equal, and thus needs to be able to execute any Agda expression in finite time.

Agda is often written in an interactive workflow. Whilst all Agda functions must be total, developers may want to (partially) leave implementations open to revisit in the future. The Agda compiler therefore supports the concept of "holes". At any time, a developer can leave a hole open inside the program by typing a `?`. The next time that the program gets compiled, the compiler will attempt to type-check the hole and replace the question mark with a marker in the form of `{! !}`. If the developer is using an editor with interactive Agda support, such as Emacs[4] or Atom[5], the editor will display a list of open holes and their types. This workflow lets developers move on with their program while keeping track of lines of code that need to be revisited. If a type-checked hole's type depends on another value which still contains holes, the compiler will use temporary type-variables to indicate that it was not able to fully determine the constraints on this particular hole.

3.2 The Curry-Howard isomorphism

The Curry-Howard isomorphism [Sørensen and Urzyczyn, 2006] states that there exists a correspondence between formal logic systems and computational calculus, or in other words, between proof theory and type theory. According to the Curry-Howard isomorphism, dependent types correspond to higher order logic.

[4] `https://agda.readthedocs.io/en/latest/tools/emacs-mode.html`, accessed on 2019-05-01

[5] `https://atom.io/packages/agda-mode`, accessed on 2019-05-01

With this isomorphism in mind, Agda can be used as a proof system. Logical statements can be expressed in the form of Agda types, and a proof is given by constructing a value of that type. This is also called a constructive proof. Crucially, constructive proofs can not be created using proof strategies in the form $(\neg\neg p \Rightarrow p)$ and $(p \vee \neg p)$.

In classical mathematics, a common proof pattern is to try and show the absurdity of the negation of a proposition. By proving that the negative of a proposition (i.e. $\neg p$) is not valid, it must follow that the proposition itself is valid (i.e. $(\neg\neg p \Rightarrow p)$). For example, in a proposition of the form *"There is an x for which f(x) holds true"*, in classical mathematics, one might try to start a proof in terms of *"Imagine there is no such x..."*, and then end on *"Since this is impossible, the opposite must be true"*. In a constructive proof system, just showing the absence of the opposite is not enough to prove a proposition to be true. One has to actually provide (or *construct*) the x in question. Similarly, the axiom of excluded middle $(p \vee \neg p)$ places an unwanted restriction on possible values of p. It states that all p must be either true or false, but doesn't leave open room for propositions which may be undecidable or otherwise undefined. In Agda, we cannot rely on these classical axioms and must instead always construct constructive proofs.

Agda's type checker forms a system in which we can do logical reasoning using existing functional programming techniques as well as more advanced dependently typed programming techniques. We mentioned the importance of all Agda expressions needing to terminate in order for the type checker to finish its static analysis in finite time. In the context of logical reasoning, it's also important for all functions to not be infinitely looping or otherwise undefined, in order to maintain the soundness of the system.

3.3 Hardware design

Moore's law, which states that the number of transistors on a circuit doubles approximately every two years, is likely not to hold up forever. In fact, some posit that there is a hard limit on miniaturisation of circuitry given by Heisenberg's uncertainty principle, where quantum effects will cause unwanted interference if transistors get too close to one another [Powell, 2008]. Instead of improving the raw performance of general-purpose computing circuits by upping the number of transistors, another popular method to create high performance chips is to implement certain algorithms directly in hardware in the form of *ASIC*s: application-specific integrated circuits. One very recent example of ASICs being used in the wild is for Bitcoin mining. Where miners used to run their block hashing algorithms on general computing chips like CPUs and GPUs, the ongoing competition for Bitcoin rewards has pushed everyone towards the most efficient and fastest computing methods of these well-defined algorithms.

Since the 1980s, researchers have been researching functional programming languages to design and reason about hardware [Sheeran, 2005]. Functional programming and hardware design match up very nicely, not in the least because FP makes it easier to reason about program properties. Besides standalone functional hardware description languages made from scratch (e.g. μ-FP [Sheeran, 1984]), several *embedded domain specific languages* (EDSLs) have been created as well (e.g. Lava [Bjesse et al., 1998], ForSyDe

[Sander and Jantsch, 2004], HAWK [Matthews et al., 1998]).

Dependently typed programming is in many ways the logical next step after functional programming, a *successor* of sorts. A dependent type system can offer advantages over *simple* FP in the creation of domain specific languages [Oury and Swierstra, 2008]. This together with the demonstrable effectiveness of functional languages for hardware description makes a language such as Agda very well-suited as a host of a hardware EDSL.

3.4 Verified translation

The words «compiler» and «transpiler» do not actually hold any real semantic difference. Both terms refer to pieces of software that translate a program description from one language to another language. In the way that these terms are usually used, a compiler will translate a user-level language to a machine-level representation, whereas a transpiler will translate between user-level languages. However, the terms «user-level» and «machine-level» aren't well-defined classifications in the context of programming languages. For all intents and purposes of this work, «compiler correctness» is synonymous to «transpiler correctness».

The core idea behind verifying a compiler is to prove equality between evaluation of the source language and the target language. Dependent types let us formulate this equality as a type, for which we can write a constructive proof. There is a well-cited paper in the area of compiler correctness within dependently typed languages [Mckinna and Wright, 2006]. In this article, McKinna and Wright describe the working of a verified compiler for an embedded expression-based language to an embedded stack-machine language, similar to low-level machine code.

The relationship between source language and target language, and the verified translation from one to the other, can be expressed as a graph (See figure 3.1).

Given a `code` in a source language $\texttt{L}_\texttt{s}$, we can evaluate this code using the semantic evaluation rules of that language by passing in some `params` in the source language's value universe $\texttt{U}_\texttt{s}$. Alternatively, we can `translate` the `code` into a `program` of target language $\texttt{L}_\texttt{t}$.

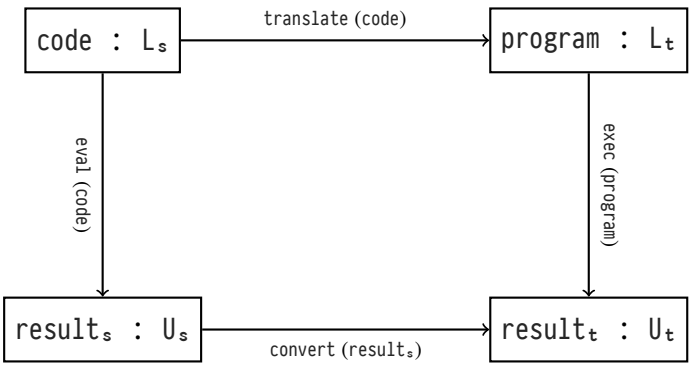

Figure 3.1: Graph demonstrating verified translation relation

By converting the params to the target language's value universe $\mathtt{U_t}$, we can execute the program passing in the converted input. The result of both evaluating the source language or executing the target language should land on the same result (after converting).

Note that even though we used different terms *evaluate* and *execute* to describe the running of either the source or the target language statements, they are conceptually the same operation, just on different inputs.

3.5 Embeddings and EDSLs

3.5.1 Variable binding in embedded domain specific languages

When talking about EDSLs, one has to differentiate between shallow and deep embeddings in the host language [Gibbons and Wu, 2014, Boulton et al., 1992]. In a deeply embedded DSL, syntactic structures are represented as data types inside the host language to allow quantifiable inspection. This is extra work for the developer of the library, but provides invaluable benefits when reasoning about the semantics of the embedded language, especially in a dependently typed context. A shallow embedding avoids such work, and only offers a mapping between the embedded language's syntax and the host's semantics. The deep embedding has the major benefit of splitting up the definition of *which* language constructs exist and *how* they have to be interpreted. This lets us inspect expressions of the embedded language without having to reflect on the definition of the shallow embedded semantics.

When developing a deeply embedded language, the library author has different choices for the representation of variables, each with their own caveats.

3.5.2 Nameless

In the context of variable binding representation techniques, nameless refers to a system where there are no variable bindings. Perhaps the most famous example of nameless binding is combinatorial logic, for example using the SKI basis [Smullyan, 1985, Curry et al., 1972].

In SKI combinator calculus, the combinators S, K and I form the three basic building blocks out of which programs can be constructed. Composing any two terms also forms a valid term, through application. The basic terms' semantics are defined as such:

$$\begin{aligned} I\,x &= x \\ K\,x\,y &= x \\ S\,x\,y\,z &= xz\,(yz) \end{aligned}$$

Even though the denotational semantics of these terms use variable identifiers x, y and z for their internal representation, the language of SKI combinator calculus only allows

the terms S, K and I, as well as their compositions through functional application. A legal expression in this language does not contain any variable bindings.

3.5.3 De Bruijn

De Bruijn is a system of variable binding commonly used when expressing λ-calculus. It is a notation which identifies a variable occurrence with the *distance* to the location of the binding λ [De Bruijn, 1994, Turing, 1937]. Traditionally, in λ-calculus, a new lambda abstraction would introduce a new named variable that can be referred to in the body of the abstraction. This naming scheme however comes with caveats. One has to consider how to handle non-uniqueness of names. This can be especially troublesome if a variable in an expression has to be replaced by a second expression that contains a free variable of the same name (β-reduction). Another difficulty with this naming scheme arises when one attempts to establish equivalence between two expressions where the only difference lies in the names of the bound variables.

The De Bruijn system solves these ambiguities, since each reference points directly to its binding location. The main disadvantage is that it is harder to find usage of a given variable throughout an expression for a human observer, since the identifier may change its value depending on the number of lambda abstractions in the expression at any given point. However, this representation makes it much easier to reason about expressions for a computer system, for example in a dependently typed proof assistant such as Agda.

3.5.4 HOAS/PHOAS

The approaches for variable binding discussed above can be implemented entirely inside the data type which represents the deep embedding. Another strategy can be to use variable binding of the host language. The host language usually has an advanced binding system which deals well with naming and shadowing and other cases where there are multiple definitions. Using the host language binding constructs is referred to as *higher-order abstract syntax*, or HOAS for short. A very simple example of lambda calculus using HOAS:

```
data L : Set where
  Lam : (L → L) → L

app : L → L → L
app (Lam f) x = f x
```

The `Lam` constructor takes a function of the type (`L → L`), which represents the body of the lambda abstraction. A user who wants to create something of this type would typically feed in an anonymous function (a lambda expression) in the host language, and use the host language's binding constructs to represent the embedded language's bindings, for example as such:

```
id = Lam (λ x → x)
```

Note how the binding of the named variable `x` is the host language's binding, but how it represents the binding in the embedded language.

The data type `L`, in the way it is presented above, is actually not legal Agda code. The `Lam` constructor is problematic. Agda only allows inductive appearances of the type in *strictly positive* positions. By looking at the `Lam` constructor as a function that takes a function as an argument, we can see that the first occurrence of `L` is contravariant in that position. This leads to a type system error in Agda. In order to see why this could be problematic, look at the following example:

```
ω = Lam (λ x → app x x)
Ω = app ω ω
```

If Agda were to try and evaluate `Ω`, it would construct a term that is the combinator equivalent of $(\lambda x.xx)(\lambda x.xx)$, of which the normal form does not terminate. This is unacceptable in the context of dependent types, since the type checker would run into an infinite loop if this were allowed. All valid expressions must be total in order for the static type checking phase to be guaranteed to terminate. Furthermore, we can't accept such undefined values in our types without making the logic they represent unsound. A solution to this problem is to use «parametrized higher-order abstract syntax», or PHOAS for short [Chlipala, 2008]. Let's add a parameter to the `L` data type:

```
data P (a : Set) where
  Lam : (a → P a) → P a
  Var : a → P a
```

By adding the type parameter to the data type `P`, the Agda type checker will catch ill-typed constructs such as `Ω` as defined above. The `Lam` constructor no longer has negative occurrences, and will be accepted by Agda. Lastly, we added the `Var` constructor to lift objects of type `a` to `P a`.

There is work by [Atkey et al., 2009] which converts expressions of untyped λ-calculus back and forth between HOAS and De Bruijn representation embedded in Haskell. Since the Haskell type checker is less restrictive than Agda's, this issue did not become a problem in their implementation. Only after they moved to simply typed λ-calculus did they run into what they refer to as *exotic* types. For their work, they also chose to add a type parameter to their higher-order abstract syntax in order to ensure well-typedness of their embedded language.

4 SKI transpiler

In this chapter, we investigate the feasibility of translating a language with typed variables to a language without such constructs, as well as proving correctness of said translation. It goes without saying that we use Agda as the platform for this work.

4.1 Simply-typed λ-calculus

The simply-typed λ-calculus is an extension of the λ-calculus which we already saw in 3.5.3 and 3.5.4. Its type system is simple, because it only consists of the unit type (ι) and the function constructor ($\rightarrow$). Simply-typed λ-calculus is a very interesting language to study the effects of type systems. However, this is not the focus of this chapter. Instead, we intend to translate the simply-typed λ-calculus into a nameless representation, in order to show that it is possible to go from a language which has variable bindings in the form of De Bruijn indices to a nameless language.

```
data U : Set where
  unit : U
  _⇒_  : U → U → U
```

```
⟦_⟧ : U → Set
⟦ unit ⟧ = ⊤
⟦ σ ⇒ τ ⟧ = ⟦ σ ⟧ → ⟦ τ ⟧
```

Agda listing 4.1: Simple type system

We model the simple type system in Agda listing 4.1. The type universe is represented with `U`. The translation which relates the type universe to the Agda type system is defined with `⟦_⟧`.

```
Ctx = List U
```

Agda listing 4.2: Type context

The term *context* is often used to mean very different things in different situations. In the frame of reference of this work, we use *context* to refer to a list of types (See Agda listing 4.2), which are used to capture the free variables in a given expression.

```
data Ref : Ctx → U → Set where
  top : ∀ {Γ τ}   → Ref (τ :: Γ) τ
  pop : ∀ {Γ σ τ} → Ref Γ τ → Ref (σ :: Γ) τ


data Term : Ctx → U → Set where
  app : ∀ {Γ σ τ} → Term Γ (σ ⇒ τ) → Term Γ σ → Term Γ τ
  lam : ∀ {Γ σ τ} → Term (σ :: Γ) τ → Term Γ (σ ⇒ τ)
  var : ∀ {Γ τ}   → Ref Γ τ → Term Γ τ
```

Agda listing 4.3: Simply typed lambda calculus language definition in Agda

The simply-typed λ-calculus is encoded in the `Term` data type (See Agda listing 4.3). Terms are indexed with a context `Ctx` which defines the types of free variables that appear in the term. Furthermore, terms are also indexed with a type `U`. Agda's type checker will only let us create valid terms that are type-safe. This is due to the *intrinsically typed* [Reynolds, 2000] nature of the object language `Term` in the host language Agda.

Pay special attention to the `var` constructor, which takes a `Ref` as a parameter. `Ref` encodes a specialized version of Peano numbers, where `top` represents *zero* and `pop` represents the *successor* operation. The specialization lies in the fact that `Ref` is indexed with a context and a type. Each call of `lam` pushes a new type onto the context stack of its argument term. This allows `var` calls to refer to that binding by encoding the depth of `lam` calls in its stack of `pop` and `top` calls.

The intrinsic design of our embedded simply-typed λ-calculus language also lets us define a meaningful evaluation function, which, given an expression in the object language together with a list of values for free occurring variables, will run the described program by mapping each term to the host language's semantics (See Agda listing 4.4).

```
data Env : Ctx → Set where
  nil  : Env []
  cons : ∀ {Γ τ} → ⟦ τ ⟧ → Env Γ → Env (τ :: Γ)


eval : ∀ {Γ τ} → Term Γ τ → Env Γ → ⟦ τ ⟧
eval (app t₁ t₂) env = eval t₁ env (eval t₂ env)
eval (lam t)     env = λ x → eval t (cons x env)
eval (var ref)   env = env ! ref
```

Agda listing 4.4: Simply typed lambda calculus evaluation semantics

4.2 SKI combinators

We touched upon SKI combinator calculus in section 3.5.2. To reiterate, they represent a Turing-complete language without any variable bindings.

```
data SKI : U → Set where
  S   : ∀ {a b c} → SKI ((a ⇒ b ⇒ c) ⇒ (a ⇒ b) ⇒ a ⇒ c)
  K   : ∀ {a b}   → SKI (a ⇒ b ⇒ a)
  I   : ∀ {a}     → SKI (a ⇒ a)
  _·_ : ∀ {α β}   → SKI (α ⇒ β) → SKI α → SKI β


apply : ∀ {τ} → SKI τ → ⟦ τ ⟧
apply S x y z   = x z (y z)
apply K x y     = x
apply I x       = x
apply (c₁ · c₂) = apply c₁ (apply c₂)
```

Agda listing 4.5: SKI combinators in Agda

We've directly encoded `S`, `K`, and `I` as Agda constructors in Agda listing 4.5 by using implicit type variables and recreating the expected shape for these combinators as their index of the type `SKI`. Similar to our definition of `Term`, this leads to an intrinsic embedding in Agda using the type universe `U`. We can combine these combinators using `_·_`. We've also included an unembedding function here called `apply`, which is the SKI equivalent of `eval` on the simply-typed λ-calculus side. The name `apply` was chosen to avoid confusion between these semantic evaluation methods.

4.3 Translation

Translating arbitrary `Term` expressions directly to `SKI` is not always possible. Terms that appear inside the body of a `lam` are *open*, i.e. they contain references inside them which are bound outside of the term. It will only be possible to translate *closed* terms to a SKI representation. To work around this restriction, we introduce an intermediate translation of possibly open terms to an intermediate representation `SKI'`, which supports bindings in Agda listing 4.6.

```
data SKI' : Ctx → U → Set where
  S   : ∀ {Γ a b c} → SKI' Γ ((a ⇒ b ⇒ c) ⇒ (a ⇒ b) ⇒ a ⇒ c)
  K   : ∀ {Γ a b}   → SKI' Γ (a ⇒ b ⇒ a)
  I   : ∀ {Γ a}     → SKI' Γ (a ⇒ a)
  _·_ : ∀ {Γ α β}   → SKI' Γ (α ⇒ β) → SKI' Γ α → SKI' Γ β
  ⟨_⟩ : ∀ {Γ τ}     → Ref Γ τ → SKI' Γ τ


cmp' : ∀ {Γ τ} → Term Γ τ → SKI' Γ τ
cmp' (app t₁ t₂) = cmp' t₁ · cmp' t₂
cmp' (lam t)     = lambda (cmp' t)
cmp' (var x)     = ⟨ x ⟩


lambda : ∀ {Γ σ τ} → SKI' (σ :: Γ) τ → SKI' Γ (σ ⇒ τ)
lambda S         = K · S
lambda K         = K · K
lambda I         = K · I
lambda (c₁ · c₂) = S · lambda c₁ · lambda c₂
lambda ⟨ top ⟩   = I
lambda ⟨ pop x ⟩ = K · ⟨ x ⟩
```

Agda listing 4.6: Translation of a simply-typed λ-calculus `Term` to an intermediate representation `SKI'`

The intermediate representation carries a context to capture the freely occuring variable bindings. Translating from `Term` to `SKI'` is trivial in the case of `app` and `var`, where we recursively compile each part of the application and store the variable reference, respectively.

The difficulty lies in translating the `lam` constructor. We introduce a helper function `lambda` to remove a variable binding from the `SKI'` expression and transform it into a new `SKI'` expression of an embedded function type. In essence, we are building a new expression in the form of a function which takes the previous variable binding as an input, thereby eliminating the binding.

This new function input will replace the value of the first reference within the context (of the type `σ` in the type signature above). Our function `lambda` has to define how to pass or discard this input value for each possible `SKI'` expression. In the case of simple occurrences of either `S`, `K`, or `I`, we can discard the input value. This is achieved by applying the combinator `K`, which does exactly that: it discards an argument. In the case of an application `_·_`, we need to recursively call `lambda` on each part of the application and also duplicate the input value for use in both parts. This is achieved by applying the combinator `S`, which applies an input to two expressions. Finally, in the case of a variable binding `⟨_⟩`, we need to inspect the De Bruijn index of that variable. If it is indeed the variable that is being replaced by our function input, we inject the input using the identity combinator `I`. For other variable bindings, we just reduce the De Bruijn index by one to make sure the reference points to the correct location in the context, which will also have been reduced by one element.

Since a closed `SKI'` expression, i.e. one with an empty context, contains no variable bindings, it can be converted to a `SKI` expression trivially. The `cmp'` function maintains the context of the input `Term`, so closed terms are translated to closed `SKI'` expressions which in turn can be translated to `SKI` expressions. This completes our translation of simply-typed λ-calculus to SKI:

```
closed : ∀ {τ} → SKI' [] τ → SKI τ
closed S = S
closed K = K
closed I = I
closed (c₁ · c₂) = closed c₁ · closed c₂
closed ⟨ () ⟩


cmp : ∀ {τ} → Term [] τ → SKI τ
cmp = closed ∘ cmp'
```

Agda listing 4.7: Final translation of simply-typed λ-calculus to `SKI`

4.4 Correctness

The compiler correctness property is expressed as follows in Agda:

```
correct : ∀ {τ} → (t : Term [] τ) → apply (cmp t) ≡ eval t nil
```

Agda listing 4.8: SKI transpiler correctness proposition

It states that, given a closed `Term` (i.e. no free variables, i.e. an empty context) of any type (`τ : U`), compiling the term and evaluating the resulting SKI combinators using `apply` is equivalent to evaluating the term directly.

The complete version of the correctness proof code is not included inside this document for conciseness' sake. We refer to the code accompanying this document. However, we can give a high-level description of what is going on. The type signature of `correct` states that given any closed term `t` of any embedded type `τ`, compiling the term to a SKI representation and applying it should give the same result as evaluating the term directly.

Agda can only check structural equality, so this requirement is stricter than just semantic equality. The strategy to prove this is to prove a similar correctness proof between term evaluation and the intermediate `SKI'` interpretation, which can include De Bruijn variables. Some notable lemmas that were required for this proof were: one to prove the correctness of the `lambda` function above, and one to prove that the environment is treated equally in the evaluation of terms and the application of the intermediate `SKI'` expressions.

5 Π-Ware and Λ_1

5.1 Π-Ware

Π-Ware is a deeply embedded domain specific language to describe hardware, which uses Agda as the host language [Flor et al., 2014]. It allows for the simulation, synthesis, and verification of hardware design. At the heart of Π-Ware lies the circuit data type ℂ (Agda listing 5.1). This data type defines how basic building blocks in the form of gates are interconnected in order to form a working circuit. It uses dependent types to guarantee the soundness of the number of connections between composited circuit elements.

```
data ℂ[_] (G : Gates) : {s : IsComb} → Ix → Ix → Set
  Gate       : ∀ {g# s}          → ℂ[ G ] {s} (|in| G g#) (|out| G g#)
  Plug       : ∀ {i o s}         → Vec (Fin i) o
                                 → ℂ[ G ] {s} i o
  _⟫_        : ∀ {i m o s}       → ℂ[ G ] {s} i m
                                 → ℂ[ G ] {s} m o
                                 → ℂ[ G ] {s} i o
  _‖_        : ∀ {i₁ o₁ i₂ o₂ s} → ℂ[ G ] {s} i₁ o₁
                                 → ℂ[ G ] {s} i₂ o₂
                                 → ℂ[ G ] {s} (i₁ + i₂) (o₁ + o₂)
  DelayLoop  : ∀ {i o l}         → ℂ[ G ] {σ} (i + l) (o + l)
                                 → ℂ[ G ] {ω} i o
```

Agda listing 5.1: Π-Ware circuit definition

The circuit data type is parametrized with a set of basic gates as a record of type `Gates`, the choice of which is up to the user. Two popular options are `BoolTrio` and `Nand`. The former contains logical negation, logical conjunction and logical disjunction. Each gate has a number of input and output wires. Notice how the `Gate` constructor above calls the `|in|` and `|out|` functions, which works on the `Gates` record taking an argument for the gate identifier `g#`, in order to specify the number of input and output wires for the given gate.

Furthermore, the circuit data type is indexed with an enumeration (`s : IsComb`) to indicate if the circuit contains loops (indicated by `ω`) or not (indicated by `σ`). The circuit is also indexed with two numbers `Ix` for input and output wires respectively. In order to get a better feel for the input and output wires, imagine that the entire circuit defines a function from a vector of the size of the number of input wires to a vector of the size of the number of output wires.

See figure 5.1 where we present an illustration of a simple example circuit consisting of a single `Gate` named `AND` from the `BoolTrio` set of `Gates`. It does not loop (indicated by

```
(Gate AND) : ℂ[ BoolTrio ] { σ } 2 1
```

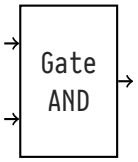

Figure 5.1: `AND` gate in Π-Ware

{ σ }). Finally, it takes 2 inputs and produces 1 output.

Circuits can be composited either in sequence (_⟫_) or in parallel (_‖_). By composing gates in parallel, the user creates a circuit that has the number of inputs and outputs of both gates added together. These gates can then be composited sequentially to create longer circuits which represent multiple chained logical steps.

By default, sequential composition will just map each output wire with index i to the input wire with index i. If this is not the desired effect, the user can employ the `Plug` constructor. Looking at the code of the `Plug` constructor, we can see it uses a vector of size `o`, where each element is a number in the range $[0, i-1]$. This lets users remap the wiring of outputs of circuits by composing the original circuit with a plug. Not only that, but by omitting or repeating certain indices in the vector, it also allows for the *forgetting* or the *duplication* of certain outputs respectively.

```
((Plug (0 :: 0 :: []) ⟫ Gate NAND) ‖ (Plug (0 :: 0 :: []) ⟫ Gate NAND)) ⟫ Gate NAND
```

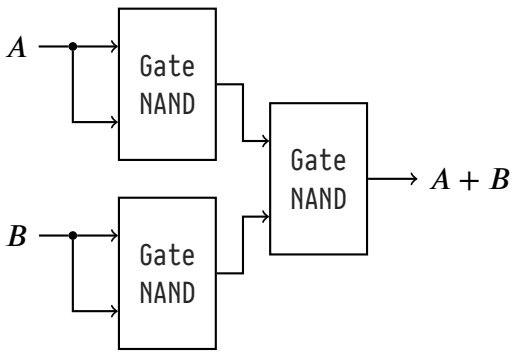

Figure 5.2: Implementation of $(A + B)$ in Π-Ware

See figure 5.2 where we present an illustration of how plugs, gates, and constructed circuitry can be composed in parallel and in sequence to produce more complex behavior. In this example, we first duplicate both the inputs (labeled A and B for convenience) using a `Plug` which outputs it's `0`th input twice before connecting them to their own `NAND` gate respectively. These operations are composed sequentially in order to generate the ”left

hand side" circuit which takes two inputs and produces two outputs. Finally, we apply sequential composition to connect the two "left hand side" outputs to another `NAND` gate, to produce our final output $A + B$.

In order to loop back wires from the output of a circuit back to its input, the user can use the `DelayLoop` constructor. Note how this is the only constructor which places a restriction on its argument's circuit, ensuring that its implicit index `{ s : IsComb }` must be equal to `σ`. It constructs a circuit with the combinational index set to `ω` to indicate the looping nature of the resulting circuit. All other constructors inside `ℂ` maintain the combinational index of their given input.

5.2 Plugs versus named variables

The circuit data structure `ℂ` uses indexed inputs and outputs. If a user wants to design a circuit with sequential composition, the Agda type checker will ensure that the number of outputs of the first circuit matches the number of inputs of the second circuit. However, the user has to pay attention themself that the wires are connected in a way that reflects the logical structure they are trying to build. Π-Ware's use of indexed gates and lack of variable bindings means that it is, similarly to SKI combinators, nameless (See section 3.5.2).

This representation is very close to the actual hardware representation of gates and wires, which is evident by the descriptions of the data structures given here. However, it requires the user to keep very precise track of outputs and inputs of circuits. This process is prone to human error when designing more complicated circuits.

Existing high-level programming languages have had support for named variables instead of indexed inputs and outputs for a long time. Using named variables creates self-documenting code, reducing the chance for human error during development. They also provide a user-friendly method to share computations across several parts of the program.

5.3 Λ_1

At the end of J. P. Pizani Flor's master thesis which introduces Π-Ware, there is mention of future work of a higher-level applicative interface language that would be nicer to use for circuit designers [Flor et al., 2014]. This follow-up work has since been published [Flor and Swierstra, 2018], presenting a new language called $\lambda\pi$-Ware. $\lambda\pi$-Ware comes in two flavors: `λB` and `λH`. These variations use De Bruijn variable bindings (See section 3.5.3) and HOAS style bindings (See section 3.5.4), respectively. We'll be focusing on the former, especially since a program of the latter can be unembedded into an equivalent program of the former.

The `λB` language inside $\lambda\pi$-Ware is indexed with a type universe and a type context, and also parametrized by a set of logical gates similar to Π-Ware. It offers several constructors, for example for referring to variables, for introducing sharing through *let*-binding, and for application.

During the development of the proofs for this thesis, $\lambda\pi$-Ware was still under active development. For this reason, we've made the decision to fork this language. Our fork offers many of the same features as $\lambda\pi$-Ware, with a few differences. Most notably, the absence of a *loop* constructor and a modification of the type universe.

We've named this fork Λ_1 (pronounced *lambda one*), after the working name that was used while J. P. Pizani Flor was developing the embedded language of $\lambda\pi$-Ware. See Agda listing 5.2 for the formal definition of this fork.

```
data Λ₁ : (Γ : Ctxt) → (Δ : List Uₚ) → (τ : Uₚ) → Set where
  ⟨_⟩              : ∀ {Γ}       → (g : Gate)
                                 → Λ₁ Γ (inputs g) (output g)
  #[_]             : ∀ {Γ τ}     → (r : Ref Γ τ)
                                 → Λ₁ Γ [] τ
  _$₁_             : ∀ {Γ Δ α β} → (f : Λ₁ Γ (α ∷ Δ) β)
                                 → (x : Λ₁ Γ [] α)
                                 → Λ₁ Γ Δ β
  letₓ_inₑ_        : ∀ {Γ Δ α τ} → (x : Λ₁ Γ [] α)
                                 → (e : Λ₁ (α ∷ Γ) Δ τ)
                                 → Λ₁ Γ Δ τ
  _,₁_             : ∀ {Γ α β}   → (x : Λ₁ Γ [] α)
                                 → (y : Λ₁ Γ [] β)
                                 → Λ₁ Γ [] (α ⊗ β)
  case⊗_of_        : ∀ {Γ Δ α β τ} → (xy : Λ₁ Γ [] (α ⊗ β))
                                 → (f : Λ₁ (α ∷ β ∷ Γ) Δ τ)
                                 → Λ₁ Γ Δ τ
  inl₁             : ∀ {Γ α β}   → (x : Λ₁ Γ [] α)
                                 → Λ₁ Γ [] (α ⊕ β)
  inr₁             : ∀ {Γ α β}   → (y : Λ₁ Γ [] β)
                                 → Λ₁ Γ [] (α ⊕ β)
  case⊕_either_or_ : ∀ {Γ Δ α β τ} → (xy : Λ₁ Γ [] (α ⊕ β))
                                 → (f : Λ₁ (α ∷ Γ) Δ τ)
                                 → (g : Λ₁ (β ∷ Γ) Δ τ)
                                 → Λ₁ Γ Δ τ
```

Agda listing 5.2: Λ_1 language definition

5.3.1 Type universe

So far, the type safety provided by the Π-Ware circuit data type $\mathbb{C}$ (after being fed with a parameter for the `Gates` to be used) consisted only of the input and output wire count. Using the input and output sizes of circuits as typing provided us with certain soundness guarantees, most notably the absence of short-circuits.

An alternative approach would be to index the circuits by the actual *type* of atomic data being transported over each input and output wire. We haven't touched on what can actually be transported along these wires, and just assumed that we were always talking about single bits. However, Π-Ware does not restrict us to transport only bits on wires. Since Π-Ware is a deep embedding rather than a shallow embedding (See section 3.5.1), the behavioral semantics of the language are defined separately from the language's structure. Π-Ware allows any data type that is finite and enumerable to be used as the so-called `Atom` to be transported over the wires.

Λ_1 uses a method of indexing input and output types separately. Instead of a type class that can be implemented for *atomic* data which can be transported along wires, it introduces a finite type universe similar to the Haskell reflective type universe, being a universe of products of sums. We present the definition of this type universe (named $\mathtt{U_p}$) in Agda listing 5.3.

```
data Uₚ : Set where
  𝟙   : Uₚ
  _⊠_ : Uₚ → Uₚ → Uₚ
  _⊞_ : Uₚ → Uₚ → Uₚ
```

Agda listing 5.3: Polytypes, the type universe for Λ_1

The *p* in $\mathtt{U_p}$ was chosen to refer to the term «polytype», since $\mathtt{U_p}$ can be used to represent any non-recursive generic type through induction over that type's structure. In Agda, all datatypes are defined as a list of constructors, where each constructor can have any number of arguments of arbitrary types themselves. The sum type represents an *alternation*. As such, the list of possible constructors for a datatype can be encoded as a sum over all possible constructors:

$$\mathtt{MyDataType_p} = \mathtt{Constructor^1} \boxplus (\mathtt{Constructor^2} \boxplus (\ldots \boxplus \mathtt{Constructor^n}))$$

The product type represents a *combination*. Each constructor can be encoded as a product of its arguments:

$$\mathtt{Constructor^i_p} = \mathtt{DataType^1} \boxtimes (\mathtt{DataType^2} \boxtimes (\ldots \boxtimes \mathtt{Datatype^m}))$$

It has been shown that a type universe such as this one is enumerable for non-recursive types [Altenkirch et al., 2007], which means we can atomize any non-recursive composition of `Uₚ` into `Atom`s to be transported over wires in Π-Ware. More about this in section 6.2.

The `Λ₁` data type has an index `τ` of type `Uₚ` to specify the *output* type of the circuit. The *input*s of a circuit are described by a list `Δ` of `Uₚ`. This is different from the published design of $\lambda\pi$-Ware, which only has a single index on `λB` for the type universe. `λB` gets away with just a single polytype index since it uses a type universe that includes function types through an arrow constructor. This lets it define inputs and outputs directly in that index.

By removing the function constructor from the definition of `Uₚ` and instead encoding the inputs and output of any `Λ₁` program explicitly in its type definition, we can forbid higher-order types. This means we can guarantee that, whenever we are given a circuit of `Λ₁`, it will not contain any contravariant occurrences of type variables.

We also provide a function `Tₚ` to map types in `Uₚ` to their corresponding Agda type in Agda listing 5.4. This allows us to create values in Agda that belong to an (un)embedded type (`Tₚ τ`).

```
Tₚ : Uₚ → Set
Tₚ 𝟙       = τ
Tₚ (σ ⊗ τ) = Tₚ σ × Tₚ τ
Tₚ (σ ⊕ τ) = Tₚ σ ⊎ Tₚ τ
```

Agda listing 5.4: Mapping of polytypes to Agda types

```
Bool : Set
Bool = Tₚ (𝟙 ⊕ 𝟙)


pattern false = inj₁ τ.tt
pattern true  = inj₂ τ.tt


_∧_ : Bool → Bool → Bool
false ∧ b = false
true  ∧ b = b
```

```
Maybe : Uₚ → Set
Maybe A = Tₚ (𝟙 ⊕ A)


pattern nothing = inj₁ τ.tt
pattern just x  = inj₂ x


is-just : ∀ {A} → Maybe A → Bool
is-just nothing  = false
is-just (just _) = true
```

Agda listing 5.5: Examples of common data types encoded as polytypes

Finally, in Agda listing 5.6, we also present a method to transform the list of input types together with a single output type as used in the indices of Λ_1 into Agda function types. We introduce a new datatype ΛSet which lets us store the tuple of Δ and τ in an alternative representation. Note how, even though we are technically reintroducing an arrow constructor, there is no way to create higher-order function types, because this arrow constructor strictly only allows adding of first-order (i.e. non-arrow) types U_p to the left-growing type term.

```
Λ⟦_⟧ : ΛSet → Set
Λ⟦ τ ⊩   ⟧ = T_p τ
Λ⟦ σ ⇁ τs ⟧ = T_p σ → Λ⟦ τs ⟧


data ΛSet : Set where
  _⊩  : U_p → ΛSet
  _⇁_ : U_p → ΛSet → ΛSet


_↠_ : (Δ : List U_p) → (τ : U_p) → ΛSet
ε      ↠ τ = τ ⊩
(x ∷ Δ) ↠ τ = x ⇁ (Δ ↠ τ)
```

Agda listing 5.6: How to transform Δ, τ to an Agda function type using Λ⟦ Δ ↠ τ ⟧

5.3.2 Variable bindings

Λ_1 uses De Bruijn indices to bind variable references. Since the language is defined recursively, any subterm has no direct knowledge of the terms that encompass it. Each term therefore carries with it a *context* Γ, which contains type information for the environment in which the term is being used. Γ works as a lookup table for type information, with the De Bruijn index of a variable being used as the index into the list.

5.3.3 Gates

We have removed the parametrisation of a gate library, instead hardcoding a set of gates. The translation of primitive gates is not of interest to this work. By choosing a fixed set of gates, the translation code is cleaner and easier to follow. It also allows us to depend on these basic gates when constructing building blocks to translate certain constructors, as we will see in section 6.

```
data Gate : Set where
  TRUE FALSE NOT AND OR : Gate
```

Agda listing 5.7: Gates used in Λ_1

6 Translation

6.1 Intermediate language

When we translated Simply Typed Lambda Calculus with typed variable bindings to SKI combinators without any bindings in section 4, we used a strategy that involved an intermediate language. The intermediate language was chosen to be close to the target language, which is supposed to be bindingless. However, we included a list of types as a context `Ctx` and allowed for an additional term constructor for references.

When translating from Λ_1 to Π-Ware, we choose a similar approach. First, we translate to an intermediate language (See Agda listing 6.1) which is almost identical to the target language Π-Ware, but also includes some type context for a term that represents a reference to a binding. It helps to visualise the references as holes in the completed circuit. The holes are always in the shape of missing circuitry on the *input*, since this is where the circuit expects the value of a specific variable. The context dictates how many output wires each placeholder has. In the second stage, when we translate the intermediate language to the target language, we need to connect the circuitry which represents the value of the variable binding to the outputs of the placeholder as we replace them. The goal is to replace every placeholder and end up with a circuit that does not need placeholders, and thus represents a valid Π-Ware circuit without variable bindings.

```
data IL[_] (G : Gates) : (Γ : Ctxt) → ℕ → ℕ → Set where
  G⟨_⟩ : ∀ {Γ}           → (g# : Gate# G)
                         → IL[ G ] Γ (#in G g#) (#out G g#)
  Grnd : ∀ {Γ o}         → IL[ G ] Γ 0 o
  Plug : ∀ {Γ i o}       → i ⊑ o
                         → IL[ G ] Γ i o
  _⋙_  : ∀ {Γ i j o}     → IL[ G ] Γ i j
                         → IL[ G ] Γ j o
                         → IL[ G ] Γ i o
  _‖_  : ∀ {Γ i₁ o₁ i₂ o₂} → IL[ G ] Γ i₁ o₁
                         → IL[ G ] Γ i₂ o₂
                         → IL[ G ] Γ (i₁ + i₂) (o₁ + o₂)
  Var  : ∀ {Γ τ}         → Ref Γ τ
                         → IL[ G ] Γ 0 (sz τ)
```

Agda listing 6.1: Intermediate language definition

Looking at the definition of our intermediate language, it should be immediately obvious that it is very close to Π-Ware. There are some differences, however. Most notably,

the addition of the (`Γ : Ctxt`) index on `IL[_]`'s type. The added constructor `Var` holds a variable in the form of a contextualized reference `Ref` (See Agda listing 6.2). Note that this reference only contains information about the *type* of the variable, not the *value*. We don't care about the value of the reference until we actually run the circuit, at which point we provide the evaluation function with a list of values, one for each item in the context. We will need evaluation semantics for this intermediate language later on in the correctness proof in section 7.4.

```
data Ref : Ctxt → Uₚ → Set where
  top : ∀ {Γ τ}   → Ref (τ :: Γ) τ
  pop : ∀ {Γ σ τ} → Ref Γ τ → Ref (σ :: Γ) τ
```

Agda listing 6.2: Definition of variable references used in the intermediate language

We already took a look at an equivalent `Ref` data type in section 4.1 The `Ref` datatype's implementation lets us refer to freely occuring variables in the context in a De Bruijn fashion (See section 3.5.3). It uses repeated calls of `pop` to encode the remaining iteration steps into the list of types (`Γ : Ctxt`). For example, `(top) : Ref (α :: β :: γ :: []) α` represents a reference to the first type, `α`. Next, `(pop top) : Ref (α :: β :: γ :: []) β` represents a reference to the second type, `β`. Note how the dependent type system is enforcing a sound reference chain into the context as we unzip the structure of `pop` calls.

While designing the intermediate language, there were two options to use as the type universe for the binding context. One option is to stay closer to the target language Π-Ware and to store the number of output wires for each reference. The alternative option is to stay closer to the source language Λ_1 and to store the type from that type universe (as shown in section 5.3.1). Even though both strategies should be manageable to bring to a working solution, we've chosen the latter option for our solution. When compared against numbers and arithmetic operations, the structured type data from the type universe of polytypes `Uₚ` is easier to manage in a dependent programming environment. The structured type data contains some information that tends to get lost when dealing with raw numbers.

Finally, the definition of `IL[_]` also contains a constructor for `Grnd`. This was added as an easy way to add null inputs inside the circuit, by essentially attaching the input wire to a *ground*. We technically don't need this constructor, but it eases the implementation effort and increases the readability for certain components that are needed later.

6.2 Atomization of polytypes

The domain of Λ_1 is different from that used in Π-Ware. Where Λ_1 circuits input and output polytypes $\mathsf{U_p}$, Π-Ware circuits work on vectors of `Atom` for their input and output. Since Π-Ware lets users use any `Atom` with the only restriction being that it is enumerable, it makes sense for us to use the simplest possible `Atom`, namely `Bool`. We refer to a vector of `Bool` as a *word*, or `W` in the code.

First, let us introduce a translation between polytyped values and words. We showed an alternative representation of input types and output type of Λ_1 using a datatype `ΛSet` in section 5.3.1. As we will see in the correctness proof in section 7, we will need the ability to *atomize* the unembedding of the Λ_1 circuit in order to be able to compare the behavioral equality of circuits translated to Π-Ware to programs in Λ_1. When we speak about atomizing the circuit, we mean the translation of inputs and outputs of the circuit from the space of polytypes to the space of words. This lets us feed Λ_1 programs with a word for the input and get a word as output. To achieve this, we require the translation of the circuit output from polytypes to words, but also the translation of the input word to polytypes. See Agda listing 6.3.

```
⇓ : ∀ {τ} → (v : Tₚ τ) → W (sz τ)
⇓ {𝟙}     _        = []
⇓ {σ ⊠ τ} (x , y)  = ⇓ x ++ ⇓ y
⇓ {σ ⊞ τ} (inj₁ x) = false ∷ pad₁ (sz τ) (⇓ x)
⇓ {σ ⊞ τ} (inj₂ y) = true ∷ pad₂ (sz σ) (⇓ y)


⇑ : ∀ {τ} → (w : W (sz τ)) → Tₚ τ
⇑ {𝟙}     _           = τ.tt
⇑ {σ ⊠ τ} w           = ⇑ (take (sz σ) w) , ⇑ (drop (sz σ) w)
⇑ {σ ⊞ τ} (false ∷ w) = inj₁ (⇑ (unpad₁ (sz σ) (sz τ) w))
⇑ {σ ⊞ τ} (true ∷ w)  = inj₂ (⇑ (unpad₂ (sz σ) (sz τ) w))
```

Agda listing 6.3: Definition of ⇓ and ⇑, to translate between polytypes and words

Remember that polytypes $\mathsf{U_p}$ can be used to describe any data type by performing induction over its generic structure. The function ⇓ lets us transform a polytyped value `v` into a word `w`. Conversely, the function ⇑ transforms a given word `w` back into a polytyped value `v`. Of course, the size of the word is dependent on how many bits we need to encode.

The unit type 𝟙 doesn't need any bits to represent its possible values, since there is only a single value possible. Product types `_⊠_` represent tuples. They require enough bits to encode both parts of the tuple. Hence, we translate product types into words by translating each part of the tuple into words and concatenating them. Finally, sum types `_⊞_` describe a choice between two polytypes. We need a single bit in order to encode which choice has

been made, and then we need to encode the polytype that was actually chosen as well.

However, there exists a caveat when encoding the chosen polytype in `σ ⊕ τ`. Our size function `sz` just returns a single size that would guarantee to fit the given polytype. Since the two possible polytypes `σ` and `τ` can potentially have different sizes, we need to choose the larger of the two sizes as the size for `σ ⊕ τ`. This in turn means that when encoding the smaller of the two polytypes, we need to pad the result with some dummy bits to meet the word-size requirement. See Agda listing 6.4.

Note that we are using our own custom *max* function for natural numbers (`_⊔₂_`). The Agda standard library does provide a max function, but it doesn't allow for easy inspection. We will go into more detail around (`_⊔₂_`) and the improvements it brings over the standard Agda one in section 7.3. For now, suffice it to say that type arguments which use (`_⊔₂_`) can be inspected by using `compare₂`, which tells us which of the two operands was greater (or less-or-equal respectively) and by how much.

We can use this property to implement the necessary padding of meaningless bits when required for the translation of a polytype into a word. When translating a polytype value (`inj₁ x`) of type `{ σ ⊕ τ }`, where `x` is of type `σ`, we can just translate `x` directly into a word of (`sz σ`) and pad it *up to* a size of (`sz τ`) using `pad₁`. Conversely, `pad₂` allows us to do the same when dealing with the other operand of `{ σ ⊕ τ }` (i.e. padding `y` from (`inj₂ y`) to up to `sz σ` bits).

In a similar fashion, when translating from words back to polytypes, we need to "unpad" the word. This throws away the meaningless bits from the word and allows us to translate the meaningful bits back into a polytyped value.

```
pad₁ : ∀ {m} n → W m → W (m ⊔₂ n)
pad₁ {m} n w with compare₂ m n
pad₁ {.m}          .(m + k) w
  | lesseq m k  = w ++ replicate false
pad₁ {.(m + suc k)} .m      w
  | greater m k = w


pad₂ : ∀ m {n} → W n → W (m ⊔₂ n)
pad₂ m {n} w with compare₂ m n
pad₂ .m          {.(m + k)} w
  | lesseq m k  = w
pad₂ .(m + suc k) {.m}      w
  | greater m k = w ++ replicate false
```

```
unpad₁ : ∀ m n → W (m ⊔₂ n) → W m
unpad₁ m n w with compare₂ m n
unpad₁ .m          .(m + k) w
  | lesseq m k  = take m w
unpad₁ .(m + suc k) .m      w
  | greater m k = w


unpad₂ : ∀ m n → W (m ⊔₂ n) → W n
unpad₂ m n w with compare₂ m n
unpad₂ .m          .(m + k) w
  | lesseq m k  = w
unpad₂ .(m + suc k) .m      w
  | greater m k = take m w
```

Agda listing 6.4: Definition of `pad` and `unpad`

Finally, we present a function `atomize` in Agda listing 6.5 which is able to take functions in the `ΛSet` space and let us run them in the `W→W` space. We achieved this by piecewise transforming chunks of the input word into polytyped values to partially apply to the `ΛSet`

for each input type inside Δ. Once all inputs are exhausted we can transform the output back to a word.

```
atomize : ∀ {Δ τ} → Λ⟦ Δ ↠ τ ⟧ → W→W (sz-list Δ) (sz τ)
atomize {[]}   l = const $ Ψ l
atomize {σ ∷ Δ} l = λ i → atomize {Δ} (l $ ⫛ {σ} (take (sz σ) i)) (drop (sz σ) i)
```

Agda listing 6.5: Definition of `atomize`

6.3 Stage 1

In our two-step translation approach, the first stage is by far the more complex one of the two. The first stage is all about translating Λ_1 terms to an intermediate language representation. We need to convert every possible constructor in Λ_1 into equivalent constructions made out of gates and plugs. The only thing we get to keep is variable bindings.

6.3.1 Translation

We present the definition of our first stage's `translate` function in Aga listing 6.6. As expected, primitive gates and variable bindings can be translated directly into our intermediate language. Tuples $(\_,_1\_)$ are simply representable by compositing each part of the tuple using parallel composition.

```
translate : ∀ {Γ Δ τ} → Λ₁ Γ Δ τ → IL[ ΛBoolTrio ] Γ (sz-list Δ) (sz τ)
translate ⟨ g ⟩                 = G⟨ g ⟩
translate #[ r ]                 = Var r
translate (f $₁ x)               = (translate x ‖ PlugId')
                                   ⋙ translate f
translate (letₓ x inₑ e)         = (translate x ‖ PlugId')
                                   ⋙ reduce-ctxt (translate e)
translate (x ,₁ y)               = translate x ‖ translate y
translate (case⊗ xy of f)        = (translate xy ‖ PlugId')
                                   ⋙ reduce-ctxt-twice (translate f)
translate (inl₁ {_} {α} {β} x) = G⟨ FALSE ⟩
                                   ‖ left-pad (sz α) (sz β) (translate x)
translate (inr₁ {_} {α} {β} y) = G⟨ TRUE ⟩
                                   ‖ right-pad (sz α) (sz β) (translate y)
translate (case⊕_either_or_ {α = α} xy f g)
                                 = branching-circuit {a = sz α}
                                   (translate xy)
                                   (reduce-ctxt (translate f))
                                   (reduce-ctxt (translate g))
```

Agda listing 6.6: Definition of `Stage1.translate`

For function application (`_$₁_`), we just attach the input (`translate x`) to the first set of wires of (`translate f`), using an identity plug (See Agda listing 6.7) for the remaining input wires. Our `PlugId'` function is smart enough to implicitly use the correct number of wires, which can also be zero. This means this definition works for both total and partial function application. See figure 6.1 for an illustration.

We achieve the implicit choice of the correct number of identity wires by letting Agda decide the size of the parameter `n` implicitly. The standard library function `allFin` provides us with a simple enumeration of numbers $(0, ..., n-1)$ which map each output to the corresponding input.

```
≤-id : ∀ n → n ≤ n
≤-id n = allFin n
```

```
PlugId' : ∀ {G Γ n} → IL[ G ] Γ n n
PlugId' {n = n} = Plug $ ≤-id n
```

Agda listing 6.7: Definition of the identity `Plug`

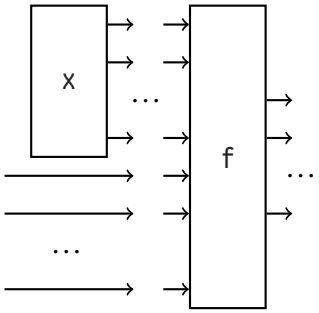

Figure 6.1: Partial function application of gates

Both translations for the sum-type constructors *in-left* `inl₁` and *in-right* `inr₁` are similar to each other in nature. They closely follow the logic described in section 6.2, where we encode a single bit to indicate the choice that the sum type represents. After this indicator bit, we encode the actual chosen circuit by recursively calling `translate` on the body of the sum-type constructor. We potentially need to pad the result of that translation with some dummy output wires in order to reach the required number of output wires as dictated by the maximum between the two possible sizes of the sum-type operands.

6.3.2 Let constructor

The `letₓ_inₑ_` constructor introduces a new variable binding. Looking back at the definition of this constructor, we can immediately see that the *let*-body is an expression that has an added element in its context:

```
(x : Λ₁ Γ [] α) → (e : Λ₁ (α ∷ Γ) Δ τ) → Λ₁ Γ Δ τ
```

However, our `translate` function only transforms from `Λ₁` to `IL` with identical contexts. Similarly, our circuit composition functions, `_⋙_` and `_∥_`, also only operate on `IL` circuits with identical contexts. This poses the question; how can we fit together the two inherently different `Λ₁` expressions `x` and `e`?

The answer is that we need to *reduce* the context. By "reducing the context", we mean that we remove the added element from the context, and instead add it as an element of the inputs. More on this in section 6.3.6.

Finally, now that we've transformed the circuit from a circuit with a variable in its context to one with an input to feed the variable's value, we can just feed our translations of the variable's value `x` into this reduced circuit the same way that we fed an input to our (partial) function application constructor `_$₁_`. The `reduce-ctxt` function will be called every time that we go under a variable binding, in order to map all occurrences of variable bindings to their appropriate inputs.

6.3.3 Case constructors

The translations for our two case switch constructors that let us operate on product types and sum types respectively are very close in nature to the let constructor. The product case constructor `case⊗_of_` is already mostly a glorified let constructor for all intents and purposes. Just like the evaluation for a let expression just adds the chosen value `x` to the evaluation environment list before evaluating the main body `e`, the evaluation for a product case expression just adds both components of `xy` to the evaluation environment list separately before evaluating the main body `f`. During the translation of the product case constructor, we need to reduce the context twice to remove both components of `xy` from the context. We use a separate function `reduce-ctxt-twice` for this rather than just calling the `reduce-ctxt` function twice. More about this choice in section 6.3.6.

Lastly, the sum case switch constructor actually presents the control flow with a branching path. The logic for this is outlined in section 6.3.7.

6.3.4 Vector coercion

Agda's dependent type system lets users reduce terms based on their structural equality. In the case of integer arithmetics, this means that Agda's type system will not evaluate the value of a given arithmetic expression to any sort of normal form. In fact, since arithmetic expressions can contain arbitrary bindings, a consistent normal form cannot necessarily be guaranteed by static analysis. Instead, the arithmetic expressions are compared syntactically. If a user wants to reduce two terms that are arithmetically equal but not syntactically equal, such as for example $(a+b)$ and $(b+a)$, the user can provide some `rewrite` clauses. By providing the type system with an equality lemma which states that `∀ { a b } → a + b ≡ b + a`, Agda can replace instances of $(a + b)$ with $(b + a)$, thereby achieving structural equality and the ability to reduce the term to `refl`, the Agda constructor for reflective equality.

However, function definitions that make extended use of `rewrite` are hard to examine. When writing proofs about such functions, the author of the proof will need to pay special attention to take the rewrites into consideration. This often leads to cryptic errors by the type-system when the author makes small mistakes. We would often run into such problems when writing the correctness proof as discussed in section 7, especially when dealing with vectors that represent inputs and outputs of circuitry.

Instead of using rewrites of integer arithmetic for vector length encoding in our translation implementation, we've opted to introduce the concept of *coercion*. A simple but powerful definition lets us coerce a circuit's input or output vectors from any integer arithmetic structure to any other equal structure. Since the caller needs to provide the equality relationship as an argument, we can use this argument when inspecting the function during our proofs later on. This has proven to be far easier to handle when compared to `rewrite` statements.

Note the simplicity of the definition. Since there is only one possible constructor `refl` for the argument `i ≡ i'`, we start our function definition on that case switch. Once `refl` has been filled in, Agda is able to structurally reduce `IL[ G ]⌜ i' o` to `IL[ G ]⌜ i o`, allowing us to just pass the input as the result. This is as expected, since we are not changing the

```
coerceᵢ : ∀ {G Γ i i' o} → i ≡ i' → IL[ G ] Γ i o → IL[ G ] Γ i' o
coerceᵢ refl e = e


coerceₒ : ∀ {G Γ i o o'} → o ≡ o' → IL[ G ] Γ i o → IL[ G ] Γ i o'
coerceₒ refl e = e
```

Agda listing 6.8: Definition of `coerce` for intermediate language

definition of the circuit. However, at the call-site of this coercion function, the caller can choose to transform the circuit's input (or output) to any equivalent arithmetic structure.

6.3.5 Combinator circuits

In section 3.5.2, we showed a computational system SKI that consists of three combinators S, K, and I, which can be combined to form more complex terms. Each of the combinators serves a different purpose. In this section, we show how to recreate the semantics of these combinators in our intermediate language circuitry.

Sequential and parallel combinators

$$Sxyz = xz(yz)$$

The S combinator is often called a *substitution* operator. It takes the output of (yz) and uses it as the second argument passed to x in the expression $(xz(yz))$. Another way to see this combinator is as a way to sequentially pass the same value into the argument list of two different functions. We've created our own version of such a combinator in our intermediate language as `S[_]·_·_` and called it the *sequential* combinator.

```
S-bypass : ∀ {G Γ} k i → IL[ G ] Γ (k + i) (k + (k + i))
S-bypass k i = coerceₒ (+-assoc k k i) $ PlugDup k ∥ PlugId i


S[_]·_·_ : ∀ {G Γ i j o} k
           → IL[ G ] Γ (k + i) j
           → IL[ G ] Γ (k + j) o
           → IL[ G ] Γ (k + i) o
S[_]·_·_ {i = i} k x y = S-bypass k i ⟫ PlugId k ∥ x ⟫ y
```

Agda listing 6.9: `S[_]·_·_` combinator circuitry

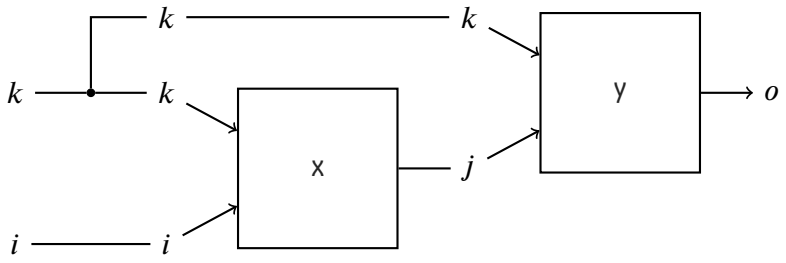

Figure 6.2: `S[_]·_·_` combinator circuitry

Since circuits work with wires, we need to provide the combinator with a number of wires `k` to indicate how many wires of input we want to duplicate. The wires get duplicated using a *bypass* construction, so that we can attach them as the first input of each argument circuit `x` and `y`.

At this point, we also introduce a new combinator circuit to supplement the sequential one, namely a *parallel* combinator `P[_]·_·_`. This combinator circuit provides an easy way to copy `k` input wires and partially apply them to both argument circuits `x` and `y` by attaching them as their first inputs.

```
P-insert : ∀ {G Γ} k i₁ i₂ → IL[ G ] Γ (k + (i₁ + i₂)) ((k + i₁) + (k + i₂))
P-insert k i₁ i₂ =
  coerceᵢₒ (+-assoc k i₁ i₂) (+-assoc (k + i₁) k i₂) $ PlugCopyK i₁ k ‖ PlugId i₂


P[_]·_·_ : ∀ {G Γ i₁ o₁ i₂ o₂} k
           → IL[ G ] Γ (k + i₁) o₁
           → IL[ G ] Γ (k + i₂) o₂
           → IL[ G ] Γ (k + (i₁ + i₂)) (o₁ + o₂)
P[_]·_·_ {i₁ = i₁} {i₂ = i₂} k x y = P-insert k i₁ i₂ ⟫ x ‖ y
```

Agda listing 6.10: `P[_]·_·_` combinator circuitry

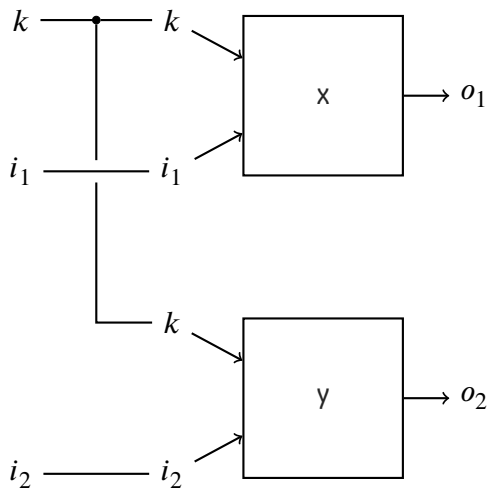

Figure 6.3: `P[_]·_·_` combinator circuitry

Kill combinator

$$Kxy = x$$

The K combinator is usually referred to as the *constant* function. When we demonstrated our SKI transpiler in section 4.3, we used the K combinator inside the function `lambda` to reduce the context of terms whenever we wanted to introduce a new dummy parameter, whose only purpose was to satisfy the requirement for an additional (unused) input parameter. We are achieving a similar feat with our combinator `K[_]·_` for circuits. It takes in a circuit `x` and adds `k` dummy input wires which are are not connected to anything. For this reason, in the context of circuitry, we've dubbed this the *kill* combinator.

```
K[_]·_ : ∀ {G Γ i o} k → IL[ G ] Γ i o → IL[ G ] Γ (k + i) o
K[ k ]· x = PlugNil k ‖ x
```

Agda listing 6.11: `K[_]·_` combinator circuitry

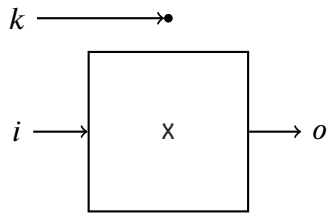

Figure 6.4: `K[_]·_` combinator circuitry

Identity combinator

$$Ix = x$$

The I combinator represents *identity*. We don't need to write any custom circuitry for this combinator, since we can just use an identity such as `PlugId'` (See Agda listing 6.7) that maps each output to the corresponding input.

6.3.6 Reducing context

The goal of context reduction is to move a binding from the head of the context list to the head of the list of inputs. One way to visualise this change is to think of it as adding a new input to the circuit, and attaching the wires of this input to every place inside the circuit where the variable binding was used. This lets us *share* a value for the binding – for example, in the form of the output of another circuit – among all the different places where that value is needed. This is similar to the bracket abstraction used in our SKI transpiler as shown in section 4.3. We present the context reduction logic for our intermediate language in Agda listing 6.12.

Our goal is to create a circuit with (`sz τ`) extra input connections, since `τ` is the type that we are removing from the context. In the case of the first three constructors, we don't actually need the value of the binding that we are removing. We can use our *kill* combinator to add the required wires without attaching them to the given circuit. Our *sequential* combinator lets us copy the new input wires to the start of two sequential circuits while keeping their sequential attachment structure intact. Similarly, the *parallel* combinator lets us do this while keeping the parallel attachment structure intact. In both cases, we

```
reduce-ctxt : ∀ {G τ Γ i o} → IL[ G ] (τ ∷ Γ) i o → IL[ G ] Γ (sz τ + i) o
reduce-ctxt {_} {τ} G⟨ g# ⟩       = K[ sz τ ]· G⟨ g# ⟩
reduce-ctxt {_} {τ} Grnd          = K[ sz τ ]· Grnd
reduce-ctxt {_} {τ} (Plug x)      = K[ sz τ ]· Plug x
reduce-ctxt {_} {τ} (x ⟫ y)       = S[ sz τ ]· reduce-ctxt x · reduce-ctxt y
reduce-ctxt {_} {τ} (x ∥ y)       = P[ sz τ ]· reduce-ctxt x · reduce-ctxt y
reduce-ctxt {_} {τ} (Var top)     = coerceᵢ (sym $ +-right-identity (sz τ)) $
                                    PlugId (sz τ)
reduce-ctxt {_} {τ} (Var (pop i)) = K[ sz τ ]· Var i
```

Agda listing 6.12: Definition of `reduce-ctxt`

recursively call `reduce-ctxt` to make sure that the new input wires are connected where they are required in the body of these circuits.

Finally, in the case of a variable reference, we need to inspect exactly which variable reference is in our hands. Remember from our variable constructor definition that the variable references are encoded in a De Bruijn fashion. In the case when the reference we encounter is not the one that we are currently trying to reduce from the context of bindings, we can safely kill the input wires, as there is no dependency between variable references. We just need to reduce the De Bruijn reference identifier by one, since, in the new reduced context, it refers to an earlier element of the list. In the case when we are actually dealing with the reference which we are removing from the context, we use an identity `Plug` to connect the new input wires into this location of the circuit.

At the end of section 6.3.3, we mentioned a special function for reducing the context twice. The reason we cannot simply call the `reduce-ctxt` twice can actually be inferred from its type signature. Given an intermediate language circuit with a context (`α ∷ β ∷ Γ`), calling `reduce-ctxt` would result in a remaining context (`β ∷ Γ`) with the initial context variable being moved to the input (`sz α + i`). A second call to `reduce-ctxt` will not result our desired output (`sz α + sz β + i`), but instead an input of (`sz β + (sz α + i)`).

For this reason, we're introducing a special function `reduce-ctxt-twice` to take two elements from the context list and add them to our input in the desired order.

6.3.7 Branching circuits

To translate the sum case switch constructor of Λ_1, we introduced a helper function – aptly named `branching-circuit` – that takes the two circuit bodies `f` and `g` and implements the branch in the control flow between the two, depending on the payload `xy`. As a rough outline, we implemented the circuit as a pipeline of sequentially arranged stages. The first bit of `xy` is extracted, since this is the decision-maker bit in the control flow. We feed

this bit together with the rest of `xy` into a *demultiplexer* to separate the values of `x` and `y` respectively. These values are then fed into the circuitry for `rdc-f` and `rdc-g`, context reduced versions of `f` and `g` respectively. Finally, the outputs of both these circuits are attached to a *multiplexer*, again using the decision bit to output the correct one of the two.

```
(xy     : IL[ ΛBoolTrio ] Γ 0 (1 + (a ⊔₂ b))) →
(rdc-f  : IL[ ΛBoolTrio ] Γ (a + i) o) →
(rdc-g  : IL[ ΛBoolTrio ] Γ (b + i) o) →
IL[ ΛBoolTrio ] Γ i o
```

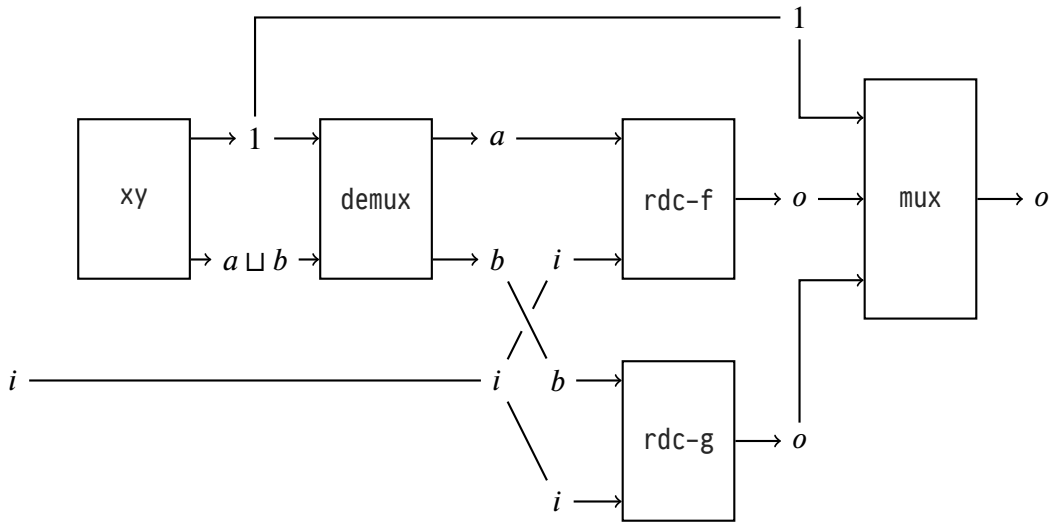

Figure 6.5: Branching circuit control flow

Looking at the circuit diagram in figure 6.5, we can see two new components that are not part of the argument circuitry, namely `demux` and `mux`. These represent a demultiplexer and a multiplexer, respectively.

The demultiplexer takes two inputs: one selector wire of a single bit, and one actual input of the maximum size between a and b, also referred to as size $(a \sqcup b)$. The selector wire will determine whether the first a wires of the actual input will get output to the first a output wires, or whether the the first b wires of the actual input will get output to the last b output wires.

The multiplexer takes three inputs: One selector wire of a single bit and two candidate inputs of identical size. The selector wire determines which of the two candidates gets passed to the output.

We refer to the code accompanying this document to see how we implemented the demultiplexer and multiplexer circuit fully in our intermediate language. The inner workings of multiplexers and demultiplexers are not controversial in terms of correctness and as such are not as interesting to the research goal at hand. We do note that the imple-

mentation depends on the use of AND, OR, and NOT gates, which is why the signature of branching-circuit shown below the circuit diagram is hard-coded to use the ΛBoolTrio gates.

6.4 Stage 2

In the second stage of our translation, we need to translate from our intermediate language IL to actual Π-Ware circuitry ℂ. We present the definition of out second stage's translate function in Agda listing 6.13. We only need to do this for IL circuits with empty contexts, since the total translation pipeline only accepts Λ_1 programs without open bindings. The only reason the first stage's translation function accepts arbitrary inputs with potential context is to let us use that function recursively on the bodies of constructors that introduce bindings.

```
grnd-circuit : ∀ {o} → ℂ[ ΛBoolTrio ] 0 o
grnd-circuit {zero}  = Plug (≒-nil zero)
grnd-circuit {suc o} = Gate FALSE ‖ grnd-circuit


translate : ∀ {i o} → IL[ ΛBoolTrio ] [] i o → ℂ[ ΛBoolTrio ] i o
translate G⟨ g# ⟩  = Gate g#
translate Grnd     = grnd-circuit
translate (Plug x) = Plug x
translate (x ⋙ y)  = translate x ⋙ translate y
translate (x ‖ y)  = translate x ‖ translate y
translate (Var ())
```

Agda listing 6.13: Definition of Stage2.translate

In the case of the Grnd constructor, we check how many null outputs were actually requested. If none, we substitute a null plug with zero inputs and zero output. In other words: no circuitry at all. If some dummy outputs were required, we just hook them up to some FALSE outputs which can represent the value 0 and be grounded. We only needed this constructor inside the implementation of our multiplexer, and it was only there to aid the readability of the multiplexer's implementation.

6.5 Final translation

In this chapter, we've shown how we can translate from Λ_1 to our intermediate language and how we can translate from our intermediate language to Π-Ware. All that remains is to combine these translation steps into a final translation definition, as seen in Agda listing 6.14.

```
open import ⋯ .Stage1.LambdaOne2IL using ()
  renaming (translate to Λ₁→IL)
open import ⋯ .Stage2.IL2PiWare using ()
  renaming (translate to IL→ΠW)


translate : ∀ {Δ τ} → (e : Λ₁ [] Δ τ) → ℂ[ ΛBoolTrio ] (sz-list Δ) (sz τ)
translate = IL→ΠW ∘ Λ₁→IL
```

Agda listing 6.14: Definition of `translate`, which translates from Λ_1 to Π-Ware

7 Correctness

In this chapter, we take a closer look at how we proved the correctness of our translation from Λ_1 to Π-Ware. To reiterate, our correctness requirement is that, for any closed expression `e` written in Λ_1, we expect the same end result regardless of whether we translate `e` to Π-Ware and run the circuit, or unembed `e` and translate that result into Π-Ware atoms. Note the call to `atomize` in the correctness proposition's signature below. In order to be able to compare the Π-Ware circuit that resulted from translating `e` to the circuit `e` itself, we need to bring both into the same input/output space. Since Π-Ware works on words and Λ_1 works on polytypes, we have the choice to bring either one into the space of the other. We already showed how `atomize` brings a function in the input/output space `ΛSet` of Λ_1 into the input/output space `W→W` of Π-Ware in section 6.2.

This chapter starts by introducing some concepts used for the correctness proof, after which we highlight some parts of the actual proof. We won't spell out the details of the entire proof in this document. The mathematics as described in code explain the proof more precisely and concisely than what we could achieve in written description in the English language. Please refer to the code accompanying this document for the full proof. The Agda type of the translation correctness proposition – i.e. the proposition that we intend to prove in this chapter – is as stated in Agda listing 7.1.

```
translate-correctness : ∀ {Δ τ}
  → (e : Λ₁ [] Δ τ)
  → ⟦ translate e ⟧[ ΛBoolTrioσ ] ≡ atomize {Δ} (unembed e ε)
```

Agda listing 7.1: Declaration of the translation correctness proposition

Since our translation works in two stages, we will split up the proof in two stages as well. In the code accompanying this document, there are two different modules that each define their own `translate-correctness` function. One of them is for *stage one correctness*, i.e. the correctness of translating Λ_1 to the intermediate language. The other is for *stage two correctness*, i.e. the correctness of translating the intermediate language to Π-Ware.

7.1 Equational reasoning in Agda

In Agda, we can define equality as follows:

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

Agda listing 7.2: Agda definition of equality (_≡_)

Given any `x` of type `A`, `x` is equal to itself. This property is more commonly known as propositional equality. Agda has built-in ways to handle this equality property. We briefly touched upon the interactive nature of writing Agda in section 3.1. A very common workflow heavily involves the use of equality.

Whenever we define a type signature for a function in Agda, for example a signature to express a certain property that we want to prove is true, we typically start off by writing the function body as a hole:

```
+-zero : (a : ℕ) → (a + 0) ≡ a
+-zero a = ?
```

Agda listing: Declaration of `+-zero`

We can instruct the interactive Agda editor to do a *case split* on the possible cases for a chosen identifier. For example, by splitting on `a`, we end up with the following definition:

```
+-zero : (a : ℕ) → (a + 0) ≡ a
+-zero zero    = ?
+-zero (suc a) = ?
```

Agda listing: Case switch on `+-zero`

The first case will be trivial to solve, as Agda will fill in the definitions of `zero` as well as `_+_` to normalize the goal to `zero ≡ zero`, which we can fulfill with the `refl` reflective equality.

For the second case, Agda will also normalize the goal, but end up stuck on the goal `suc (a + 0) ≡ suc a`. We can solve this goal by applying the concept of congruence, which states that if two values `x` and `y` are equal, for any given function `f` when applied (`f x`) and (`f y`) are also equal.

```
cong : ∀ {A B : Set} (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl

+-zero : (a : ℕ) → (a + 0) ≡ a
+-zero zero    = refl
+-zero (suc a) = cong suc (+-zero a)
```

Agda listing 7.3: Definition of `+-zero` using congruence `cong`

In our example of proving that `(a + 0)` is equal to `a`, we were able to prove the equality directly for each case of `a`. However, often we want to prove equality of two expressions by proving that both expressions are equal to a different intermediary expression. This concept is called transitivity:

```
trans : ∀ {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
trans refl refl = refl
```

Agda listing 7.4: Definition of transitivity `trans`

By providing two proofs, that `x` is equal to `y` and that `y` is equal to `z`, we effectively prove that `x` must be equal to `z`. This pattern of proving through intermediary steps is so common that there is an Agda module that greatly simplifies the execution of such proofs. This module is named *equational reasoning*: `≡-Reasoning`.

```
module ≡-Reasoning {A : Set} where
  infix  1 begin_
  infixr 2 _≡⟨_⟩_
  infix  3 _∎

  begin_ : ∀ {x y : A} → x ≡ y → x ≡ y
  begin x≡y = x≡y

  _≡⟨_⟩_ : ∀ (x : A) {y z : A} → x ≡ y → y ≡ z → x ≡ z
  x ≡⟨ x≡y ⟩ y≡z = trans x≡y y≡z

  _∎ : ∀ (x : A) → x ≡ x
  x ∎ = refl
```

Agda listing 7.5: Module for equational reasoning `≡-Reasoning`

This module provides a syntax that lets us write equality proofs in a very readable way. In equational reasoning, we write down the explicit values for each step as well as

proofs to show equality between each value. This works by chaining together a number of `_≡⟨_⟩_` operators. The first argument of this operator is the initial value `x` that we wish to prove equality about. The middle argument is an equality proof between `x` and a second value `y`. The last argument is (surprisingly) not the target value, but rather another proof of equality between `y` and `z`. The operator returns the transitive proof that `x` is equal to `z`. A caller can use the *QED* (Quod Erat Demonstrandum, that which was to be demonstrated) operator (`_∎`) to transform the final target value into a reflective equality proof. Thanks to the right associativity of the `_≡⟨_⟩_` operator, we can chain a number of steps together to create code that is very pleasant to read.

```
example : ∀ {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
example {_} {x} {y} {z} x≡y y≡z = begin
    x
  ≡⟨ x≡y ⟩
    y
  ≡⟨ y≡z ⟩
    z
  ∎
```

Agda listing 7.6: Simple example using `≡-Reasoning`

While writing proofs in the interactive Agda environment, this approach to equational reasoning also allows developers to use holes `?` to let the Agda type checker assist in finding expressions that fulfill the type requirements.

7.2 Functional extensionality in Agda

So far, we've seen Agda being able to deduce equality between values based on reflective equality. We've also seen a few lemmas that expand this equality through congruence and transitivity. Agda's standard library also provides us with lemmas for – among others – symmetry (`x ≡ y → y ≡ x`), substitution (`x ≡ y → P x → P y`) and congruence of application (`f ≡ g → f x ≡ g x`). This last one is especially interesting, since we haven't seen equality of functions yet.

On first glance, it would seem reasonable for Agda to provide a lemma in its standard library that states that if two functions have the same result for all possible inputs, the functions must be the same: `(∀ {x} → f x ≡ g x) → f ≡ g`. However, this lemma isn't available by default.

In order to clarify this ommission, we have to look at the difference between *intensional* equality and *extensional* equality [Univalent Foundations Program, 2013]. Intensional equality deals with equality through equal definition, whereas extensional equality distinguishes objects based on their observable behavior. Agda's type system is an intensional type system.

Agda's intensional type system uses β-reduction and η-reduction to normalize terms using their definitions in order to work out typing constraints. Extensional equality cannot be used for this, since extensional equality only equates things that behave the same. In the intensional type system of Agda, two functions are only equal if we can prove this using reflexivity.

Agda's standard library gives us a workaround for this problem. We can `postulate` functional extensionality, i.e. a lemma that two functions which, for each element of their domain map to identical elements of their codomain, are equal to each other. This postulate is known to be consistent. Using it will not compromise the soundness of our development.

This equivalence relation between functions explicitly states that we only care about values. Other side effects such as running time or memory usage are not of concern. This is good enough for our correctness proof, where we want to prove that the circuits provide the same output value regardless of whether we run the higher-level hardware description directly or translate it to gates and wires first, without regard for the runtime of our unembedding functions.

Functional extensionality is important for the correctness proof of our translation. The function signature for `translate-correctness` expresses an equality between two functions, namely between two `W→W` functions. The left hand side is the result of the translation of the Λ_1 expression and the right hand side is the atomized version of the Λ_1 expression. In order to prove the correctness of the translation, we want to prove that the evaluation of both these variants results in the same output for every possible input word. Given functional extensionality postulated under `fun-ext`, we can rephrase our correctness proposition for the first stage in the two-step translation pipeline as in Agda listing 7.7

```
translate-correctness : ∀ {Γ Δ τ} {env : Env Γ}
  → (e : Λ₁ Γ Δ τ)
  → eval[ gσ ] (translate e) env ≡ atomize {Δ} (unembed e env)
translate-correctness e = fun-ext λ w → translate-correctness-ext e w


translate-correctness-ext : ∀ {Γ Δ τ} {env : Env Γ}
  → (e : Λ₁ Γ Δ τ)
  → (w : W (sz-list Δ))
  → eval[ gσ ] (translate e) env w ≡ atomize {Δ} (unembed e env) w
```

Agda listing 7.7: Functional extensionality for `Stage1.translate-correctness`

7.3 Atomization correctness

In section 6.2, we showed how we can transform back and forth between polytypes and vectors of Π-Ware atoms (i.e. *words*). It seems reasonable to prove that, when we do a back-and-forth translation, the result should be unchanged. We introduce a proof for a proposition that specifies this in Agda listing 7.8. The proposition states that, for any value (x : Tₚ τ) of a polytype τ, translating it to a word and subsequently back to a polytype value, the value remains identical.

```
⇑∘⇓-identity : ∀ {τ} (x : Tₚ τ) → ⇑ (⇓ x) ≡ x
⇑∘⇓-identity {𝟙}     _       = refl
⇑∘⇓-identity {σ ⊗ τ} (x , y)  = begin
    ⇑ (take (sz σ) (⇓ x ++ ⇓ y)) , ⇑ (drop (sz σ) (⇓ x ++ ⇓ y))
  ≡⟨ cong₂ (λ p q → ⇑ p , ⇑ q) take-++-identity (drop-++-identity (⇓ x)) ⟩
    ⇑ (⇓ x) , ⇑ (⇓ y)
  ≡⟨ cong₂ (λ p q → p , q) (⇑∘⇓-identity x) (⇑∘⇓-identity y) ⟩
    x , y
  ∎
⇑∘⇓-identity {σ ⊕ τ} (inj₁ x) = begin
    inj₁ (⇑ (unpad₁ (sz σ) (sz τ) (pad₁ (sz τ) (⇓ x))))
  ≡⟨ cong (λ z → inj₁ (⇑ z)) (unpad₁∘pad₁-identity (sz τ)) ⟩
    inj₁ (⇑ (⇓ x))
  ≡⟨ cong inj₁ (⇑∘⇓-identity x) ⟩
    inj₁ x
  ∎
⇑∘⇓-identity {σ ⊕ τ} (inj₂ y) = begin
    inj₂ (⇑ (unpad₂ (sz σ) (sz τ) (pad₂ (sz σ) (⇓ y))))
  ≡⟨ cong (λ z → inj₂ (⇑ z)) (unpad₂∘pad₂-identity (sz σ)) ⟩
    inj₂ (⇑ (⇓ y))
  ≡⟨ cong inj₂ (⇑∘⇓-identity y) ⟩
    inj₂ y
  ∎
```

Agda listing 7.8: Correctness proof for translating back and forth between words and polytypes

Since Agda can't case switch on (x : Tₚ τ), as it does not have a way to know what results of (Tₚ τ) to expect, we do a case switch on the implicit parameter τ instead and let Agda fill in possible values for x from there.

In the case of tuples, we use a lemma of our own creation (See Agda listing 7.9) that proves that taking m items from an m + n vector that was built using the (_++_) operator results in the first operand. We use a similar one for dropping the first m items as well,

resulting in the second operand. This gives us the intermediate value (⩛ (⫢ x) , ⩛ (⫢ y)), on which we can recursively use the proposition that we are proving.

```
take-++-identity :
  ∀ {A : Set} {m n} {v₁ : Vec A m} {v₂ : Vec A n} → take m (v₁ ++ v₂) ≡ v₁
take-++-identity {m = zero} {v₁ = []}     = refl
take-++-identity {m = suc m} {v₁ = x ∷ v₁} =
  cong (λ z → x ∷ z) take-++-identity
```

Agda listing 7.9: Lemma `take-++-identity`

In the case of sums, we also use a lemma of our own creation (See Agda listing 7.10) that proves the nature of *unpadding* a *padded* word results in the original word. This lemma relies heavily on our own improved version (_⊔₂_) of a *max* function for natural numbers.

```
unpad₁∘pad₁-identity : ∀ {m} n {w : W m} → unpad₁ m n (pad₁ n w) ≡ w
unpad₁∘pad₁-identity {m} n with compare₂ m n
unpad₁∘pad₁-identity {.m}         .(m + k) | lesseq m k  = take-++-identity
unpad₁∘pad₁-identity {.(m + suc k)} .m     | greater m k = refl
```

Agda listing 7.10: Lemma `unpad₁∘pad₁-identity`

The Agda standard library version of the natural number max function works by rebuilding a result based on the arguments passed to it. We're introducing a version that lets Agda keep a reference to the actual maximum argument, including the difference between the arguments. We're providing this max function in two flavors. The first one distinguishes between *less*, *equal*, and *greater*. The second, which is incidentally the one that is used in the code for the transpiler, only distinguishes between *less or equal* and *greater*. Since the implementation of (_⊔₂_) is based on the result of `compare₂`, we can use the same comparison function to do case splits in our proofs.

```
_⊔_ : ℕ → ℕ → ℕ
zero ⊔ n      = n
suc m ⊔ zero  = suc m
suc m ⊔ suc n = suc (m ⊔ n)
```

Agda listing 7.11: Agda standard library version of *max*

```
data Ordering₂ : Rel ℕ Level.zero where
  lesseq  : ∀ m k → Ordering₂ m (m + k)
  greater : ∀ m k → Ordering₂ (m + suc k) m


compare₂ : ∀ m n → Ordering₂ m n
compare₂ zero    n = lesseq zero n
compare₂ (suc m) zero = greater zero m
compare₂ (suc m) (suc n) with compare₂ m n
compare₂ (suc m)           (suc .(m + k)) | lesseq .m k  = lesseq (suc m) k
compare₂ (suc .(n + suc k)) (suc n)       | greater .n k = greater (suc n) k


_⊔₂_ : ℕ → ℕ → ℕ
m ⊔₂ n with compare₂ m n
m          ⊔₂ .(m + k) | lesseq .m k  = m + k
.(n + suc k) ⊔₂ n      | greater .n k = n + suc k
```

Agda listing 7.12: Our improved version of *max*

7.4 Evaluation semantics

The correctness proposition as stated at the beginning of this chapter depends on the evaluation semantics of both Λ_1 and Π-Ware. Furthermore, our proof is split in two stages, just like the translation was. First, we want to prove a correctness proposition for translating from Λ_1 to our intermediate language. Second, we want to prove a correctness proposition for translating from our intermediate language to Π-Ware. This means that we also require an unembedding of our intermediate language.

7.4.1 Semantics of Π-Ware and intermediate language

The evaluation semantics for Π-Ware are based on input and output of *words*. An unembedded circuit is nothing more than a function which takes an input word and produces an output word. Since the intermediate language is designed to closely represent Π-Ware, the same applies there. The intermediate language also supports variables that exist in the polytype universe, for which we need to additionally unembed the value that we extract from the environment. Since the Π-Ware semantics are a subset of the Λ_1 semantics, we just demonstrate the latter in Agda listing 7.13.

```
plugσ : ∀ {i o} → i ⊑ o → W→W i o
plugσ p w = tabulate (flip lookup w ∘ flip lookup p)


seqσ : ∀ {i m o} → W→W i m → W→W m o → W→W i o
seqσ f₁ f₂ = f₂ ∘ f₁


parσ : ∀ {i₁ o₁ i₂ o₂} → W→W i₁ o₁ → W→W i₂ o₂ → W→W (i₁ + i₂) (o₁ + o₂)
parσ {i₁} f₁ f₂ w = f₁ (take i₁ w) ++ f₂ (drop i₁ w)


eval[_] : ∀ {G Γ i o} → Gateσ G → IL[ G ] Γ i o → Env Γ → W→W i o
eval[ gσ ] G⟨ g# ⟩ env = gσ g#
eval[ gσ ] Grnd    env = const (replicate false)
eval[ gσ ] (Plug x) env = plugσ x
eval[ gσ ] (x ⟫ y)  env = seqσ (eval[ gσ ] x env) (eval[ gσ ] y env)
eval[ gσ ] (x ∥ y)  env = parσ (eval[ gσ ] x env) (eval[ gσ ] y env)
eval[ gσ ] (Var x)  env = const $ ⇓ (env ! x)
```

Agda listing 7.13: Intermediate language semantics

The caller needs to supply the evaluation function with gate semantics (`gσ : Gateσ`) which define how each gate operates. Furthermore, since our intermediate language supports variables of polytypes, callers also need to provide the evaluation function with an environment of values that act as a lookup table when evaluating the `Var` constructor. Note how the evaluation semantics cast the value inside the environment from a polytype value into a word by calling the ⇓ function. Π-Ware defines the semantics for plugs as well as for sequential and parallel compositions. We are swapping out the Agda standard library's version of `take` and `drop` for our own versions (See Agda listing 7.15). Even though the standard library provides a functional one, our version makes it easier to prove some equality lemmas.

```
take : ∀ {A} m {n} → Vec A (m + n) → Vec A m
take m xs         with splitAt m xs
take m .(ys ++ zs) | (ys , zs , refl) = ys


drop : ∀ {A} m {n} → Vec A (m + n) → Vec A n
drop m xs         with splitAt m xs
drop m .(ys ++ zs) | (ys , zs , refl) = zs
```

Agda listing 7.14: Agda standard library version of `take` and `drop`

```
take : ∀ {A} m {n} → Vec A (m + n) → Vec A m
take zero    v       = []
take (suc m) (x ∷ v) = x ∷ take m v


drop : ∀ {A} m {n} → Vec A (m + n) → Vec A n
drop zero    v       = v
drop (suc m) (x ∷ v) = drop m v
```

Agda listing 7.15: Our improved version of `take` and `drop`

7.4.2 Semantics of Λ_1

The evaluation semantics of Λ_1 use unembedded polytypes for the inputs, output and environment of the unembedding function. In section 5.3.1, we already demonstrated the workings of `ΛSet` and how, together with `Λ⟦_⟧`, it can be used to transform a pair of inputs and output polytypes into an Agda function type with an arbitrary number of function parameters. This allows us to specify the unembedding in a very native Agda way as seen in Agda listing 7.16.

```
unembed : ∀ {Γ Δ τ} → (x : Λ₁ Γ Δ τ) → Env Γ → Λ⟦ Δ ↠ τ ⟧
unembed ⟨ g ⟩                  env = unembed-gate g
unembed #[ r ]                 env = env ! r
unembed (f $₁ x)               env = (unembed f env) (unembed x env)
unembed (letₓ x inₑ e)         env = unembed e ((unembed x env) ∷ env)
unembed (x ,₁ y)               env = (unembed x env) , (unembed y env)
unembed (case⊗ xy of f)        env = unembed f (
                                       (proj₁ $ unembed xy env) ∷
                                       (proj₂ $ unembed xy env) ∷
                                       env)
unembed (inl₁ x)               env = inj₁ (unembed x env)
unembed (inr₁ y)               env = inj₂ (unembed y env)
unembed (case⊕ xy either f or g) env with unembed xy env
... | inj₁ x                       = unembed f (x ∷ env)
... | inj₂ y                       = unembed g (y ∷ env)
```

Agda listing 7.16: Unembedding of Λ_1

7.5 Let correctness

In order to look at the correctness of the `letₓ_inₑ_` constructor's translation, we first need to take a closer look at the semantics of both the constructor itself as well as its translation. The evaluation semantics of the `letₓ_inₑ_` constructor dictate that the unembedding of `x` is added to the head of the variable environment before unembedding `f` itself. When atomized, this looks as follows:

```
atomize {Δ} (unembed e ((unembed x env) ∷ env)) w
```

Recalling from section 6.3.2, the translation of the $\texttt{let}_x\texttt{\_in}_e\texttt{\_}$ constructor into Π-Ware takes the translation of x and supplements it with a parallel identity plug in order to partially apply it to a reduced-context version of the translation of e:

```
(translate x ‖ PlugId') ⟫ reduce-ctxt (translate e)
```

When looking at the definition of the evaluation semantics of Π-Ware, the expression above normalizes to the following expression when evaluated. This is the expression for which we need to show that it is equal to the atomization of the unembedding of the $\texttt{let}_x\texttt{\_in}_e\texttt{\_}$ constructor:

```
(
  (eval[ gσ ] (reduce-ctxt (translate e)) env) ∘
  (parσ
    (eval[ gσ ] (translate x) env)
    (plugσ (≐-id (sz-list Δ)))
  )
) w
```

Using equational reasoning together with the concept of congruence cong, we can piecewise break down this complex evaluation with equality lemmas until we arrive at the desired atomization expression. First, let us massage this expression into a more readable form by replacing some calls with their actual definitions:

```
translate-correctness-ext {Δ = Δ} {env = env} (letₓ x inₑ e) w =
  let open ≡-Reasoning in ≡-Reasoning.begin
(
  (eval[ gσ ] (reduce-ctxt (translate e)) env) ∘
  (parσ
    (eval[ gσ ] (translate x) env)
    (plugσ (≐-id (sz-list Δ)))
  )
) w
  ≡⟨ refl ⟩
eval[ gσ ] (reduce-ctxt (translate e)) env
(
  (eval[ gσ ] (translate x) env (take 0 w)) ++
  (plugσ (≐-id (sz-list Δ)) (drop 0 w))
)
…
```

Agda listing 7.17: Correctness of $\texttt{let}_x\texttt{\_in}_e\texttt{\_}$ translation (1)

First, we've restructured the function composition call `_∘_` in order to pass `w` directly to `parσ`. Second, we've actually inserted the definition of `parσ` to explicitly pass the appropriate `take` and `drop` calls on `w` to both components of the parallel composition. We know that the variable `x` inside a *let* constructor cannot have any inputs; we can see that here since the expression reduces to one where we take zero atoms from the input word `w`. Thanks to our effort of making the expression more readable, we can immediately see that we can replace the call to `(take 0 w)` with an empty vector `[]` as well as the call to `(drop 0 w)` with the full word `w`. This is possible thanks to our design of custom `take` and `drop` functions earlier.

At this point, we can try to replace some of the parts of this expression by injecting lemmas through congruence:

```
⋯
  ≡⟨ cong (λ z → ⋯) plug-id-semantics-lemma ⟩
eval[ gσ ] (reduce-ctxt (translate e)) env
(
  (eval[ gσ ] (translate x) env []) ++
  w
)
  ≡⟨ cong (λ z → ⋯) (translate-correctness-ext x []) ⟩
eval[ gσ ] (reduce-ctxt (translate e)) env
(
  (atomize {[]} (unembed x env) []) ++
  w
)
  ≡⟨ refl ⟩
eval[ gσ ] (reduce-ctxt (translate e)) env (Ψ (unembed x env) ++ w)
⋯
```

Agda listing 7.18: Correctness of $\texttt{let}_{\texttt{x}}\texttt{\_in}_{\texttt{e}}\texttt{\_}$ translation (2)

The calls to `cong` inside these Agda listings have been shortened by not explicitly writing down the function body. This was done to keep this document readable. The contents of `⋯` repeat most of the expression, just replacing the sub-expression for which we want to replace it with an alternative (given the lemma) with `z`.

In the first step, we call our lemma `plug-id-semantics-lemma` which proves that identity plugs, when applied to a word, are equal to that word itself: `(plugσ (≐-id k) w ≡ w)`. We refer to the accompanying code for the implementation of this lemma.

In the second step, we recursively use the correctness proposition on `x`. Since we know that `x` doesn't have any inputs, `(atomize {[]} (unembed x env) [])` reduces to

(⟱ (unembed x env)) as per the definition of atomize.

At this point, we need a proposition that proves the correctness of reduce-ctxt. We know from section 6.3.6 that, when we call reduce-ctxt on circuitry expressions, we move a variable binding from the context into the list of inputs by transforming the underlying circuitry to share the input at all the necessary positions. We call this proposition reduce-ctxt-correctness and will take a close look at it in the next section.

```
…
  ≡⟨ reduce-ctxt-correctness (translate e) (unembed x env) ⟩
eval[ gσ ] (translate e) (unembed x env ∷ env) w
  ≡⟨ translate-correctness-ext e w ⟩
atomize {Δ} (unembed e ((unembed x env) ∷ env)) w
  ∎
```

Agda listing 7.19: Correctness of $\texttt{let}_{\texttt{x}}\texttt{\_in}_{\texttt{e}}\texttt{\_}$ translation (3)

Finally, we can make a recursive call to (translate-correctness-ext e w). It makes intuitive sense that, for the $\texttt{let}_{\texttt{x}}\texttt{\_in}_{\texttt{e}}\texttt{\_}$ constructor to be correct, both components x and e's correctness would be necessary too.

7.6 Reduce context correctness

The proof that our reduce-ctxt function works as intended is one of the main proofs within our correctness proof. To recap, context reduction was the method that we used to remove variable bindings by sharing their value to all the reference sites of that variable. This is the critical step when translating Λ_1 (which has variable bindings) to Π-Ware (which is nameless).

```
reduce-ctxt-correctness :
  ∀ {G τ Γ i o} {gσ : Gateσ G} {env : Env Γ} {w : W i}
  → (e : IL[ G ] (τ ∷ Γ) i o)
  → (x : Tₚ τ)
  → (eval[ gσ ] (reduce-ctxt e) env (⟱ x ++ w)) ≡ (eval[ gσ ] e (x ∷ env) w)
```

Agda listing 7.20: Correctness proposition for reduce-ctxt

The reduce-ctxt-correctness proposition (See Agda listing 7.20) states that, given an intermediate language expression e and a value x, evaluating e after reducing whilst giving it (⟱ x) as an additional input should output the same word as evaluating a non-reduced e

with x being part of the environment instead. Note that since our intermediate language uses polytypes on its environment, the type of x here is a polytyped value, which means that we need to cast it to a word when passing it as an input on the left hand side. In the previous section, when we made a call to reduce-ctxt-correctness, we actually passed (unembed x env) (where x refers to the Λ_1 expression of the *let* constructor) as the argument for our new identifier x here.

We need to prove the proposition for the reduce-ctxt call on every possible constructor in our intermediate language. Looking back at the actual definition of reduce-ctxt in section 6.3.6, we used combinator circuits of our own creation to implement the sharing of the new input. Let's take a look at a few examples to illustrate how the proof works for some of these combinators.

7.6.1 Reducing gates

```
reduce-ctxt {_} {τ} G⟨ g# ⟩ = K[ sz τ ]· G⟨ g# ⟩

K[ k ]· x = PlugNil k ‖ x
```

Recall that evaluating gates in Π-Ware and our intermediate language doesn't depend on the environment. This means that we expect that, during our equational reasoning, we will at some point manage to remove any reference to (x : T_p τ).

```
reduce-ctxt-correctness {τ = τ} {gσ = gσ} {w = w} G⟨ g# ⟩ x =
  let open ≡-Reasoning in ≡-Reasoning.begin
parσ (plugσ (≒-nil (sz τ))) (gσ g#) (⇓ x ++ w)
  ≡⟨ refl ⟩
plugσ (≒-nil (sz τ)) (take (sz τ) (⇓ x ++ w)) ++
(gσ g#) (drop (sz τ) (⇓ x ++ w))
  ≡⟨ refl ⟩
(gσ g#) (drop (sz τ) (⇓ x ++ w))
⋯
```

Agda listing 7.21: Correctness of reduce-ctxt for gates (1)

In fact, we can see that we are able to remove the entire empty part of our killer plug PlugNil without needing any additional lemmas. Once again, once we wrote out the definition of parσ explicitly with calls to our custom take and drop functions, we get a clear picture of how we can clean up both components of the parallel composition. In this case,

the first part dropped away completely. Agda isn't able to normalize (`drop (sz τ) (⇓ x ++ w)`) by itself, so we help it along with a lemma:

```
  …
    ≡⟨ cong (λ z → (gσ g#) z) (drop-++-identity (⇓ x)) ⟩
gσ g# w
    ∎
```

Agda listing 7.22: Correctness of `reduce-ctxt` for gates (2)

The `drop-++-identity` lemma works very similar to the `take-++-identity` lemma that we saw earlier. Both depend on our custom implementation of `drop` and `take`, respectively. Note how the proof here follows the concept behind the implementation of `reduce-ctxt` closely. We implemented the plug to kill the extra input wires that are given, since we don't need them for the implementation of this constructor once the variable has been removed from the context. The steps in the proof clearly show how we first remove the extra input and then massage the remainder with a lemma to bring it to the desired form. By splitting up the proofs into smaller (sometimes reusable) sub-proofs, we create a pleasant environment where we can write proofs that are not only sound, but also easy to follow step-by-step.

7.6.2 Reducing compositions

```
  reduce-ctxt {_} {τ} (x ⋙ y) = S[ sz τ ]· reduce-ctxt x · reduce-ctxt y
  reduce-ctxt {_} {τ} (x ∥ y) = P[ sz τ ]· reduce-ctxt x · reduce-ctxt y

    S-bypass k i = coerceₒ (+-assoc k k i) $ PlugDup k ∥ PlugId i
    S[_]·_·_ {i = i} k x y = S-bypass k i ⋙ PlugId k ∥ x ⋙ y

P-insert k i₁ i₂ =
  coerceᵢₒ (+-assoc k i₁ i₂) (+-assoc (k + i₁) k i₂) $ PlugCopyK i₁ k ∥ PlugId i₂
P[_]·_·_ {i₁ = i₁} {i₂ = i₂} k x y = P-insert k i₁ i₂ ⋙ x ∥ y
```

The sequential and parallel composition constructors use custom combinators `S[_]·_·_` and `P[_]·_·_` respectively. The actual proof of `reduce-ctxt-correctness` for these two cases is quite verbose, given the number of plugs and compositions that they use. For that reason, we refer to the accompanying code for the full evidence. Even though we're not explaining the entire proof for these cases in this document, we do want

to touch on the use of vector coercion, since we introduced it in section 6.3.4 specifically with the intent of making writing proofs easier.

We mentioned that we want to carry the transformations of vectors explicitly rather than relying on `rewrite` mechanisms. The proofs for these two cases of `reduce-ctxt-correctness` make use of the benefit of this explicitness. For example, let's take a look at the `S-bypass` construct. For the bypass to work, we want it to have an output of `(k + (k + i))` wires. Note the explicit placement of parentheses here. Even though we are all intuitively familiar with the associativity of `_+_` on natural numbers, Agda does not have this intuition built-in. Due to `_+_` being defined asymmetrically based on a case switch on it's first argument, concepts such as associativity or even commutativity are not given. Of course, Agda's standard library provides us with lemmas that prove these concepts, for example by giving us equality between `(k + (k + i))` and `((k + k) + i)` in the form of `(+-assoc k k i)`.

Vector sizes are specified as a natural number on their type index. The `coerce-vec` function lets us change the type of a vector by modifying its size index according to an equality lemma. Now, when we encounter such a coerced vector in our equational reasoning, we need a way to transform it back to the original. We were not able to provide a single catch-all lemma that uncoerces vectors in Agda, but we managed to write easy lemmas proving that coerced vectors behave as expected when we pass them as arguments to other functions. Without showing the full code of these lemmas (we refer to the accompanying code for that), one example to illustrate the use of coerced vectors can be found in Agda listing 7.23. This lemma proves that `take m` works on an associativity-coerced vector as expected. We can use lemmas like this one inside steps of equational reasoning to get rid of the calls to `coerce-vec`.

```
take-++-identity-c+a :
  ∀ {A} {m k₀ k₁} →
  (vₘ : Vec A m) → (v₀ : Vec A k₀) → (v₁ : Vec A k₁) →
  take m (coerce-vec (+-assoc m k₀ k₁) ((vₘ ++ v₀) ++ v₁)) ≡ vₘ
```

Agda listing 7.23: Example lemma using `coerce-vec`

7.7 Final correctness

In this chapter, we've shown how we can use equational reasoning and functional extensionality to prove the correctness proposition which we started with. We showed and explained some highlights of proofs inside the two-step translation pipeline. Finally, we'd like to state the correctness proof one more time, this time with the proof's body. For the full implementation, we – once again – refer to the code accompanying this document.

```
open import ⋯ .Stage1.LambdaOne2IL using () renaming (translate to Λ₁→IL)
open import ⋯ .Stage2.IL2PiWare using () renaming (translate to IL→ΠW)
open import ⋯ .Translation using (translate)


open import ⋯ .Stage1.Properties.Correctness using ()
  renaming (translate-correctness to stage1-correctness)
open import ⋯ .Stage2.Properties.Correctness using ()
  renaming (translate-correctness to stage2-correctness)


translate-correctness : ∀ {Δ τ}
  → (e : Λ₁ [] Δ τ)
  → ⟦ translate e ⟧[ ΛBoolTrioσ ] ≡ atomize {Δ} (unembed e ε)
translate-correctness {Δ} e =
  let open ≡-Reasoning in ≡-Reasoning.begin
    ⟦ translate e ⟧[ ΛBoolTrioσ ]
  ≡⟨ refl ⟩
    ⟦ IL→ΠW (Λ₁→IL e) ⟧[ ΛBoolTrioσ ]
  ≡⟨ stage2-correctness (Λ₁→IL e) ⟩
    eval[ ΛBoolTrioσ ] (Λ₁→IL e) ε
  ≡⟨ stage1-correctness e ⟩
    atomize {Δ} (unembed e ε)
  ∎
```

Agda listing 7.24: Final translation correctness proof

8 Conclusion

8.1 Research summary

Our initial research goal stated that we'd like to translate a hardware description language with variable bindings to one without variable bindings. We've presented a complete translation from Λ_1 (an embedded hardware description language in Agda with variable bindings) to Π-Ware (an embedded hardware description language in Agda without variable bindings). We also proven the translation to be correct, highlighting some parts of that correctness proof inside this document. We showed how to successfully and efficiently use the dependent type system of Agda to facilitate this correctness proof, while also showing some small potential improvements to the Agda standard library.

8.2 Future work

8.2.1 Remaining postulates and holes

This document does not provide a complete view of the translation and correctness proof code, but rather highlights some interesting parts of it. At multiple points, we've referred to the accompanying code to get a complete view. For the sake of transparency, we need to mention that there are still a few holes left open in the code that we sadly were not able to solve. Although the two-step translation is 100% complete, we've not managed to get the same completion ratio for the first step of the correctness proof. Agda allows for an option to interpret open holes as postulates, which lets us use the incomplete lemmas while treating them as proven true. This means we are able to show the high-level structure of the proof under the assumption that our lemmas are provable.

Although we've proven the correctness of the translation of the `case0_of_` constructor at a high level, the translation makes use of a lemma called `reduce-ctxt-twice-correctness`. We mentioned in section 6.3.6 that we need some reordering plugs before we can reduce the context twice. Given that the hard part of the correctness proof lies in proving the correctness of `reduce-correctness`, we are confident that we could prove the remaining hole that proves the correct behavior of these reordering plugs.

Although we spent a considerable amount of time implementing a custom multiplexer and demultiplexer in intermediate language circuitry, and are quite confident that they work as intended, we did not manage to formally prove this correctness. Both these components use a lot of composite circuitry. Although proving the correctness of these components would have been an interesting case-study into the scalability of our correctness proof approach using equational reasoning in Agda, we decided to focus our efforts elsewhere given that the workings of multiplexers and demultiplexers are not very controversial.

Lastly, we did not manage to prove the correctness of the translation of our *in-left* $\texttt{inl}_1$ and *in-right* $\texttt{inr}_1$ constructors. Although we firmly believe that our new comparison function for natural numbers is superior to the existing one inside Agda's standard library,

we were not able to convince the type system to use it in solving the lemma around padding.

8.2.2 Potential follow-up

Besides filling up the last remaining holes in the proof, this project offers other potential future follow-up research. For example, we made a conscious choice to remove looping constructors in order to focus the research on the translation to a nameless language. Reintroducing loops would be interesting since it would affect the evaluation semantics and thereby affect the correctness proof. Furthermore, it would be interesting to follow-up this work by allowing higher-order variables inside the source language.

We hope to see further research into hardware embedded languages that are embedded in dependent type systems in the future, as we believe there is still lots of untapped potential in this field.

Special thanks

Special thanks to:

Wouter Swierstra, my mentor and supervisor during this thesis. Thanks for your patience and for sharing your knowledge around everything Agda, type systems, and correctness proofs to help bring this project to a successful close.

Jurriaan Hage, for your detailed and extensive review of this thesis.

My parents **Corine & René Spoel**, who gave their unconditional love and support towards my studies from beginning to end.

References

[Altenkirch et al., 2007] Altenkirch, T., McBride, C., and Morris, P. (2007). Generic programming with dependent types. In *Datatype-Generic Programming*, pages 209–257. Springer.

[Atkey et al., 2009] Atkey, R., Lindley, S., and Yallop, J. (2009). Unembedding domain-specific languages. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM.

[Bjesse et al., 1998] Bjesse, P., Claessen, K., Sheeran, M., and Singh, S. (1998). Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM.

[Boulton et al., 1992] Boulton, R. J., Gordon, A. D., Gordon, M. J., Harrison, J., Herbert, J., and Van Tassel, J. (1992). Experience with embedding hardware description languages in hol. In *TPCD*, volume 10, pages 129–156. Citeseer.

[Bove et al., 2009] Bove, A., Dybjer, P., and Norell, U. (2009). A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer.

[Chlipala, 2008] Chlipala, A. (2008). Parametric higher-order abstract syntax for mechanized semantics. In *ACM Sigplan Notices*, volume 43, pages 143–156. ACM.

[Curry et al., 1972] Curry, H. B., Feys, R., Craig, W., and Craig, W. (1972). Combinatory logic.

[De Bruijn, 1994] De Bruijn, N. (1994). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Selected Papers on Automath*, 133:375–388.

[Flor and Swierstra, 2018] Flor, J. P. P. and Swierstra, W. (2018). Verified timing transformations in synchronous circuits with $\lambda\pi$-ware. In *International Conference on Interactive Theorem Proving*, pages 504–522. Springer.

[Flor et al., 2014] Flor, J. P. P., Swierstra, W., and Sijsling, Y. (2014). Piware: Hardware description and verification in agda.

[Gibbons and Wu, 2014] Gibbons, J. and Wu, N. (2014). Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *ACM SIGPLAN Notices*, volume 49, pages 339–347. ACM.

[Intel, 2004] Intel (2004). Statistical analysis of floating point flaw: Intel white paper.

[Martin-Löf, 1984] Martin-Löf, P. (1984). Intuitionistic type theory. *Naples: Bibliopolis*, 76.

[Matthews et al., 1998] Matthews, J., Cook, B., and Launchbury, J. (1998). Microprocessor specification in hawk. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 90–101. IEEE.

[Mckinna and Wright, 2006] Mckinna, J. and Wright, J. (2006). A type-correct, stack-safe, provably correct, expression compiler. In *in Epigram. Submitted to the Journal of Functional Programming*. Citeseer.

[Oury and Swierstra, 2008] Oury, N. and Swierstra, W. (2008). The power of pi. In *ACM Sigplan Notices*, volume 43, pages 39–50. ACM.

[Powell, 2008] Powell, J. R. (2008). The quantum limit to moore's law. *Proceedings of the IEEE*, 96(8):1247–1248.

[Rekhi and Purasai, 2003] Rekhi, S. and Purasai, R. (2003). The next level of abstraction: Evolution in the life of an asic design engineer. *Synopsys Users Group (SNUG), San Jose*.

[Reynolds, 2000] Reynolds, J. C. (2000). The meaning of types from intrinsic to extrinsic semantics.

[Sander and Jantsch, 2004] Sander, I. and Jantsch, A. (2004). System modeling and transformational design refinement in forsyde [formal system design]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32.

[Sheeran, 1984] Sheeran, M. (1984). mufp, a language for vlsi design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112. ACM.

[Sheeran, 2005] Sheeran, M. (2005). Hardware design and functional programming: a perfect match. *J. UCS*, 11(7):1135–1158.

[Smullyan, 1985] Smullyan, R. M. (1985). *To Mock a Mockingbird: and other logic puzzles including an amazing adventure in combinatory logic*. Oxford University Press, USA.

[Sørensen and Urzyczyn, 2006] Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier.

[Turing, 1937] Turing, A. M. (1937). Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163.

[Univalent Foundations Program, 2013] Univalent Foundations Program, T. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study.

Appendix

Code is available at `https://github.com/robrene/thesis`