Π-Ware: An Embedded Hardware Description Language using Dependent Types

João Paulo Pizani Flor
Department of Information and Computing Sciences,
Utrecht University - The Netherlands
e-mail: j.p.pizaniflor@students.uu.nl

Monday 4th August, 2014

Chapter 1

Introduction

Several factors have been causing an increasing demand for hardware acceleration of algorithms. On the one hand, Moore's law still holds for the near future [11], which means bigger circuits in the same wafer area. On the other hand, advances in Instruction-Level Paralellism (ILP) and microarchitecture are starting to have diminishing returns [9].

Also, there is pressure to reduce the duration and cost of the circuit design process. These trends are at odds with the techniques and tooling used in the hardware design process, which have not experienced the same evolution as the ones for software development.

The design of an Application-Specific Integrated Circuit (ASIC) imposes strong requirements, specially with regards to correctness (functional and otherwise). Design mistakes which make their way into the manufacturing process can be very expensive, and errors that reach the consumer cost even more, as there's no such thing as "updating" a chip. One infamous example of such a bug reaching the consumer was the *FDIV* bug in Intel's Pentium chip, which might have costed the company over 400 million dollars [19].

In the software industry, specially in application domains requiring high correctness assurance (such as information security, machine control, etc.), functional programming techniques have long been used to improve productivity and reduce the need for extensive testing and debugging. These claims have been confirmed both in industrial settings [24] and in experiments [13].

Another advantage usually attributed to functional programming is an increased capacity to *reason* about your programs, specially to perform what is called *equational reasoning*. Equational reasoning can be used even as an optimization technique. For example, some libraries for array programming in Haskell take advantage of the following law involving `map` and function composition. The proof is derived by structural induction on the array or list, and by using equational reasoning with the definitions of the functions involved:

```
map f . map g == map (f . g)
```

Besides the growing usage of functional programming languages in several domain areas, functional techniques and constructs keep "penetrating" imperative languages with each new release. Some examples are:

- Apple's recently released Swift™ language, which features immutable data structures, first-class functions and type inference.

- Java's adoption of *generics* and -- more recently -- of lambda expressions[1].
- The concept of *nullable type*[2] in C#, equivalent to Haskell's `Maybe`.
- Python's *generator expressions*, inspired by list comprehensions and lazy evaluation.

In a certain way, we can compare the power of the tools and techniques used nowadays in hardware design to the early days of software development. Of course there are inherent and fundamental differences between the two activities, but this comparison leads us to ask whether recent ideas from programming language research, specially those related to functional programming, can be used to improve hardware design.

Research trying to answer this broad question started already in the 1980s, with the work of Prof. Mary Sheeran and others [22], developing *functional* hardware description languages, such as *μFP* [21]. Later, a trend emerged of developing Embedded Domain-Specific Languages (EDSLs) for hardware description, *hosted* in purely functional languages, such as Haskell [16]. Prominent examples of this trend are the Lava [4] family, as well as ForSyDe [20].

A circuit description written in Lava -- an EDSL hosted in Haskell -- can look like the following:

```
toggle :: Signal Bool
toggle = let output = inv (latch output) in output
```

Even though pure functional languages (especially Haskell) are well-suited for implementing EDSLs, there is still room for improvement in the type-safety of the embedded languages. This can be done by hosting the EDSLs in language supporting *dependent types*, as exemplified in the paper "The Power of Pi" [17].

A type system with *dependent types* can express strong properties about the programs written in it. Systems with dependent types can be seen as regular programming languages, but because of the expressive power of dependent types, we can also see them as *interactive theorem provers*.

Using dependent types, one can more easily being together *specification* and *implementation*. The type signature of a function can give many more *guarantees* about its behaviour. A classical example when introducing Dependently-Typed Programming (DTP) is the type of sized vectors, along with the safe head function, depicted in Listing 1.

```
data Vect (α : Set) : ℕ → Set where
  ε       : Vect α zero
  _::_    : ∀ {n} → α → Vect α n → Vect α (suc n)

head : ∀ {α n} → Vect α (suc n) → α
head (x :: xs) = x
```

Listing 1: Type of sized vectors and a safe head function.

In this example, the parameter to head cannot be empty -- its size will be at least 1 (suc zero). This is expressed in the parameter's type: Vect α (suc n). When checking the

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
[2] http://msdn.microsoft.com/en-us/library/1t3y8s4s.aspx

totality of head (whether all cases are covered by pattern matching), the type checker will notice that the only constructor of Vect able to produce an element of type Vect α (suc n) is _ :: _.

In the safe head example, we made the specification more precise by constraining the type of an argument. We can also use dependent types to make the return type of a function more precise. The group function for sized vectors in *Agda*'s standard library has the following type:

```
group : ∀ {α : Set} n k  → (xs : Vec α (n * k))
                         → ∃ λ (xss : Vec (Vec α k) n) → xs ≡ concat xss
```

It receives as parameters, besides the vector to be "sliced", the number of slices desired (n) and the size of each slice (k). Notice that the size of the passed vector needs to match ($n * k$). The return type of this function may be read as:

> A vector (xss) of size n (whose elements are vectors of size k), such that $xs \equiv$ concat xss.

Therefore, it returns a collection of groups "sliced" from the original vector, each with the requested size. Additionally, it returns a proof that concatenating all groups results in the original vector. This type serves as a *complete specification of correctness* for the function. Any function with this type is, by definition, a *correct "grouping" function*.

In a deep-embedded DSL for hardware there is usually a *type* representing circuits. One can imagine that it would be useful to design this type in such a way that as few of its elements as possible are *malformed*. Therefore, we want to make the typing of the circuits as strong as possible, to eliminate as many *classes of design mistakes* as possible.

Depedent type systems allow for easy expression of these *well-formedness* rules for circuits. One simple criterion of circuit *well-formedness* is, for example, that it contains no "floating" signals. That is, all of a circuit's internal components must have all their ports connected. Figure 1.1 shows an example of circuit violating this rule.

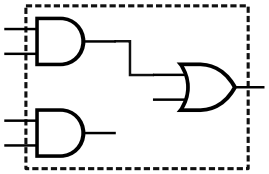

Figure 1.1: Malformed circuit with a "floating" signal.

In this M.Sc thesis we developed a hardware EDSL -- Π-Ware -- which enforces this and other well-formedness rules using dependent types. Π-Ware is a *deep-embedded* Domain-Specific Language (DSL) hosted in the Agda programming language, supporting simulation for combinational and synchronous sequential circuits. Furthermore, the user of Π-Ware can prove *expressive* properties about circuits in the logic of Agda itself (intuitionistic first-order logic).

In Chapter 2 we will establish the questions, concepts, languages and tools from which we took inspiration to create Π-Ware. We will discuss the process of hardware design, functional languages for hardware, and how dependent types can enter the picture.

In Chapter 3 we dive into a detailed study of the Π-Ware EDSL itself. We provide a detailed account of how Π-Ware works currently, what design decisions were involved in its development, and how to use it. Specifically, we give examples of circuits modelled in Π-Ware and proofs of properties involving these circuits.

Finally, in Chapter 4, we discuss what has been achieved by Π-Ware as it stands by comparing it with other hardware EDSLs, recent and past. We also clarify what desirable features reamin to be implemented as future work, and on which conditions (if any) these implementation efforts depend.

Chapter 2

Background

2.1 Hardware Description

The hardware development process can be well understood by analyzing its similarities and differences to software development. Both hardware and software development usually "begin" with a high-level *specification* of the algorithm to be implemented. Also, both proceed by a series of translations, increasingly adding more details to the description.

However, the final targets of both hardware and software development differ: while in software the final artifact is machine code (sequence of instructions) for some architecture, in hardware the target is usually a *floorplan*, a spacially placed graph of logic gates and wires.

Also, the transformation steps in the software and hardware chains are different, as depicted in Figure 2.1. In this figure, the rectangular boxes represent transformations steps towards a more detailed description, while the ellipsoid shapes represent artifacts that are consumed/produced by each step.

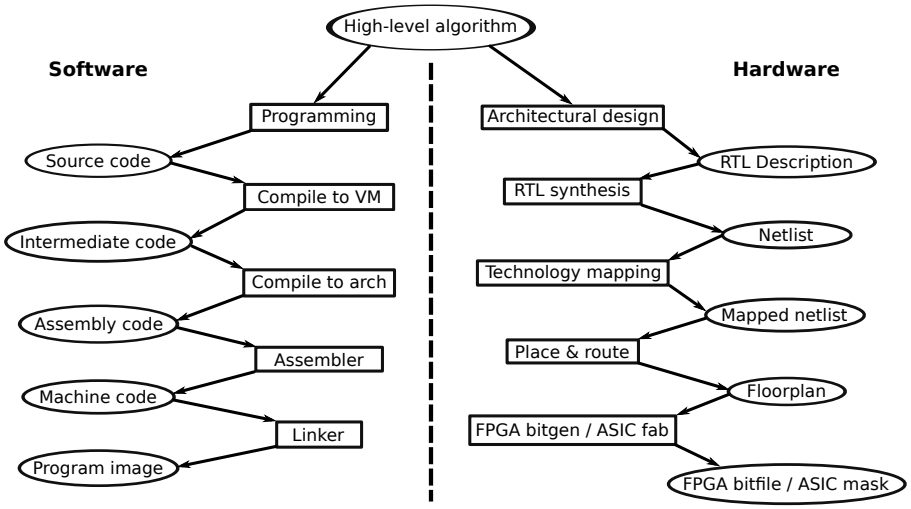

Figure 2.1: Software and Hardware refinement chains.

The first (highest) two levels in the hardware implementation flow are usually described using so-called Hardware Description Languages (HDLs), usually Verilog or VHDL. Nowadays they are used by hardware engineers to write both behavioural specifications of circuits (topmost ellipsis in Figure 2.1) as well as Register-Transfer Level (RTL) descriptions. However, these languages were originally designed for *simulation* purposes, and there are several problems that arise when using them to model hardware *architecture* and behaviour.

First of all, only a *subset* of these languages can be used for *synthesis* (actually deriving a netlist and floorplan). Although there is a standard [1] defining a *synthesizable subset* of VHDL, tools differ greatly in the level of support.

To further complicate the matter, this synthesizable subset is not *syntactically segregated*. One example of complex requirement for synthesizability of VHDL is: "in a process, every output must be assigned a value for every possible combination of input values". In Listing 2 we have an example process violating this requirement.

```
process(sel, a, b)
begin
    if (sel = '0') then
        y <= a;
    end if;
end process;
```

Listing 2: Unsynthesizable VHDL process (no `else` branch).

Another significant difference between hardware and software development is the level of *automatization* in the chains (Figure 2.1). Usually, all transformation steps in the software chain are automatic. On the hardware chain, the *crucial* "Architectural design" step is mostly manual.

The complex task of making explicit the space-time trade-offs lays in the hands of the hardware designer: they must decide how much parallelism to use, how to pipeline the processing steps of the algorithm, taking care to not generate data hazards, etc.

Recently, tools have been developed for this first step -- called High-Level Synthesis (HLS). They usually take *behavioural VHDL*, C or even C++ as input, and produce RTL code. However, current HLS-based hardware implementation chains still face two main problems:

- The input languages (VHDL, C) are not expressive enough.
- Specification, verification and synthesis are all done with different tools and different languages.
  - Even behavioural VHDL can be considered practically a different language than Register-Transfer Level VHDL.

Functional programming languages have been touted as a solution to both of these problems. A functional program is a more abstract and expressive specification of an algorithm than a C function or VHDL entity. Also, functional languages could be used to *both* write the specification of a circuit's behaviour, as well as its Register-Transfer Level description.

2.2 Functional Hardware Description

In the beginning of the 1980s, many researchers were trying to use functional programming languages to design and reason about hardware circuits. These developments were happening at the same time when the VHSIC Hardware Description Language (VHDL) was being designed and standardized. Even though VHDL and Verilog ended up "winning" and becoming the *de facto* industry standards, it is still useful to take a look at the ideas behind these early HDLs, as some of them inspired current approaches.

The idea was then to come up with new functional languages, *specialized* to do hardware description. One of the prominent early examples in this set is µFP [21], which was in turn inspired by John Backus's FP [3]. In contrast to VHDL, which was later adapted to synthesis in an ad-hoc way, µFP was designed since the beginning to have both interpretations:

Behavioural Each "primitive" circuit as well as each combining form (higher-order function) has an attached *functional semantics*, used in simulation.

Geometric Each combining form has a typical geometric interpretation. For example, sequential composition of two circuits c_1 and c_2 will result in a floorplan in which c_2 is placed "adjacent to" c_1 and connected to it by the required wires.

The µFP language is an *extension* of Backus's FP, and contained only one extra combining form: µ. The µ combining form is responsible for the creation of functions (circuits) with internal state. According to the seminal µFP paper [21]:

> The meaning of µf is defined in terms of the meaning of f. The functional "out" hides the state so that while $M(f)$ maps a sequence of input-state pairs to a sequence of output-state pairs, out($M(f)$) just maps a sequence of inputs to a sequence of outputs. For a given cycle, the next output and the next state depend on the current input and the current state.

One very simple example of a circuit with internal state is a shift register. In Figure 2.2, we can see the structure of this circuit. Each of the dotted boxes represents a µFP expression. The smaller box denotes a combinational circuit (with 2 inputs and 2 outputs) that swaps its inputs. The bigger box corresponds to the application of the µ combinator to the smaller one. It adds the indicated latch and a feedback loop, creating a stateful circuit.

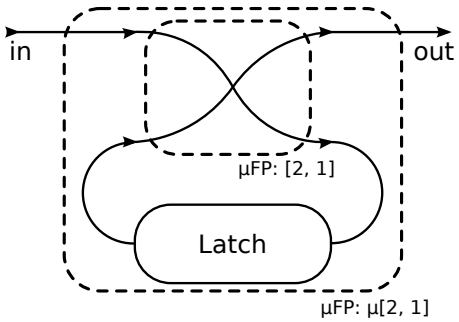

Figure 2.2: *Shift register*: a simple example of sequential circuit in µFP.

By being a conservative extension of Backus's FP, almost all algebraic laws of FP also hold for μFP. Furthermore, the μ combining form has useful algebraic laws of its own. The most notable of these laws states that the composition of two "μ-wrapped" functions can be converted to one single "μ-wrapper".

In other words: a circuit with several, localized, memory elements can be converted into a circuit with one "centralized" memory bank and a combinational block. If we apply this rewrite rule from right to left, we can see it as a form of "optimization", in which we start with a single memory bank and refine the design towards one in which memory elements sit closer to the sub-circuits using them.

Two of the core ideas of μFP served as inspiration for Π-Ware:

Double interpretation As in μFP, each circuit and circuit combinator in Π-Ware has two distinct semantics: they can be simulated (functional semantics) or synthesized to a netlist (geometric semantics).

Single sequential constructor As in μFP, there is only *one way* in Π-Ware to construct a sequential circuit. Also, the same law regarding the "state-introducing" constructor holds in Π-Ware, but because we use Agda as host language, this meta-theoretical property can be proven *in the same language* in which circuits are described.

2.2.1 Embedded Functional Hardware Description

With the growing popularity and advocacy [12] of *embedded* DSLs, a trend emerged of also having embedded HDLs. Functional programming languages were a natural fit for hosting DSLs. In particular, the Haskell programming language proved to be a popular choice of host, due to features such as lazy evaluation, highly-customizable and overloadable syntax.

Some examples of highly-successful DSLs implemented in Haskell are:

- Attoparsec [1] (monadic parsing)
- Accelerate [2] (data-parallel computing)
- Esqueleto [3] (SQL queries)
- Diagrams [4] (2D and 3D vector graphics)

Also, some of the most popular and powerful Embedded Hardware Description Languages (EHDLs) are hosted by Haskell. Let us now present some of these EHDLs and discuss some of their limitations, which will lead to the question of how to solve them using dependent types.

Lava

Perhaps the most popular family of hardware DSLs embedded in Haskell is the *Lava* family. Lava's first incarnation [4] was developed at Chalmers University of Technology and Xilinx, and used a shallow-embedding in Haskell, along with a monadic description

[1] `https://github.com/bos/attoparsec`
[2] `https://github.com/AccelerateHS/accelerate/wiki`
[3] `https://github.com/prowdsponsor/esqueleto`
[4] `http://projects.haskell.org/diagrams/`

style to handle naming and sharing. These circuit monads were parameterized, and by using different instances, different interpretations could be given to a single circuit description, such as simulation, synthesis, or model checking.

The design of Lava suffered significant changes later on, abandoning the monadic interface and adopting an observable sharing solution based on reference equality [7]. Currently, Lava has several dialects, such as *Chalmers-Lava*, *Xilinx-Lava*, *York-Lava* and *Kansas-Lava*. We base our examples in *Chalmers-Lava* [5], which can be said to be the "canonical" dialect, and also the most actively developed.

In Lava, circuits are written as normal Haskell functions, by using pattern matching, function application and local naming. One extra restriction is that the types of the arguments are constructed by the `Signal` type constructor. For example, a negation gate in Lava would have the following type signature:

```
inv :: Signal Bool -> Signal Bool
```

Another restriction is that all circuit inputs must be "uncurried", i.e., even though the *circuit* has several inputs, the *function* modeling that circuit must have only one argument, with all inputs in a *tuple*. A `NAND` gate in Lava would look like this:

```
nand2 :: (Signal Bool, Signal Bool) -> Signal Bool
nand2 (a, b) = inv (and2 (a, b))
```

Finally, due to the instances provided for the `Signal` type, the only way to "aggregate" bits in Lava is by using tuples and lists. Using only these structures we forego some type safety that Haskell *could* provide. For example, an n-bit binary (ripple-carry) adder in Lava has the following description:

```
adder :: (Signal Bool, ([Signal Bool], [Signal Bool]))
      -> ([Signal Bool], Signal Bool)

adder (carryIn, ([] ,[]))    = ([], carryIn)
adder (carryIn, (a:as, b:bs) = (sum:sums, carryOut)
    where (sum, carry)     = fullAdd (carryIn, (a, b))
          (sums, carryOut) = adder (carry, (as, bs))
```

In this circuit description, there is an *expectation* that both inputs have the same size. When this expectation is not met, a *run-time* error will occur during simulation. This happens because the given definition is *partial*: the cases for `(carryIn, ([], b:bs))` and `(carryIn, (a:as, []))` are left undefined.

This limitation of Lava can be solved by dependent types, namely by using statically-sized vectors. Another limitation of Lava is related to the way in which it solves the observable sharing question: in order to detect sharing and cycles, the equality over the `Signal` type is defined as an equality over the references used. Therefore, comparisons of signals "created" in different sites will fail, even though their values are the same. For example, the following expressions will evaluate to `False`:

```
test1 :: Bool
test1 = low == low

test2 :: Bool
test2 = simulate adder (low, ([low], [low])) == low
```

[5] `https://hackage.haskell.org/package/chalmers-lava2000`

This limitation is also not present in Π-Ware, due to the way in which we describe circuits in a structural fashion, completely avoiding the need to deal with observable sharing.

ForSyDe

Another example of EHDL in Haskell is ForSyDe [20]. ForSyDe and Lava differ substantially in description style and internal workings, and a more complete comparison of the two can be found in the final report of the experimentation project we conducted as preparation to this thesis [18].

In ForSyDe, the central concepts are those of *process* and *signal*. Whereas in Lava only *synchronous* sequential circuits can be described -- that is also the case in Π-Ware -- in ForSyDe processes can belong to synchronous, asynchronous and continuous *models of computation*.

In the synchronous model of computation (which we studied more deeply), process constructors take a combinational function (called *process function* in ForSyDe jargon) and turn it into a synchronous sequential circuit (for example a state machine).

But, instead of just using (smart) constructors of a certain "circuit datatype" to build this process function, ForSyDe relies on *Template Haskell* to *quote* regular Haskell syntax into an Abstract Syntax Tree (AST) -- which is then further processed by ForSyDe. For example, the description of an adder in ForSyDe would look like the following:

```
adderFun :: ProcFun (Int8 -> Int8 -> Int8)
adderFun = $(newProcFun [d|
        adderFun' :: Int8 -> Int8 -> Int8
        adderFun' x y = x + y
    |])

adderProc :: Signal Int8 -> Signal Int8 -> Signal Int8
adderProc = zipWithSY "adderProcess" adderFun
```

Notice how the `adderFun` *process function* is built from a "regular" haskell function (`adderFun'`), which is then *quoted* (by the `[d|` quasi-quoter), processed by `newProcFun` and finally *spliced* back into place. The (combinational) process function is then "lifted" into the synchronous sequential setting by the `zipWithSY` *process constructor*, which just "zips" the inputs signals (streams) by applying the given process function *pointwise*.

Not any Haskell function can be quoted and processed by `newProcFun` and turned into a ForSyDe process function: the argument and return types of a `ProcType` must belong to the `ProcType` type class. Instances of this class are provided only for:

Primitive types `Int`, `Int8`, `Int16`, `Int32`, `Bool`, `Bit`

Enumerated types User-defined enumerations, with derived instances for `Data` and `Lift`

Containers Tuples and fixed-length vectors (`Data.Param.FSVec`), holding a type of the above two categories and unrestrictedly nested.

Under certain conditions, ForSyDe is able to generate VHDL netlists from the system description. In order for a ForSyDe system description to be *synthesizable*, all *process functions* need to comply with some extra requirements:

Pointed notation Declarations with point-free notation are not accepted

Single-clause To be synthesizable, the body of a *process function* cannot have multiple clauses, and it cannot have `let` or `where` blocks. This essentially forbids recursion inside *process functions*. Pattern matching is only possible by using the `case` construct.

Despite all these limitations, ForSyDe still provides a more "typed" approach to hardware description than Lava, with perhaps its most distinctive feature being the usage of fixed-length vectors. In the `parameterized-data` package page on *Hackage* [6], the authors admit that the library's function is to provide "type-level computations and *emulate dependent types*". Therefore it is not a stretch to assume that using *actual* dependent types could improve on the ideas proposed by ForSyDe.

Hawk and Cλash

Finally, in this short review of functional hardware EDSLs, there are two more alternatives to be mentioned: Hawk and Cλash.

Hawk [15] is a Haskell-hosted EDSL with a different target than the ones presented until now: instead of modelling circuits, it intends to model *microarchitectures*. Therefore, it has a higher level of abstraction than Lava or even ForSyDe.

Models in Hawk are executable, and *shallow-embedded*. However, Hawk uses a technique similar to the original Lava version [4] in order to also allow for symbolic evaluations: The descriptions are "parameterized" by type classes (in Hawk's case they are `Instruction`, `Boolean`, etc.), and by providing different instances, different *interpretations* are achieved.

The authors of Hawk mention some shortcomings of Haskell which they met, among which:

- Haskell's `List` datatype doesn't quite match the intended semantics for Hawk's `Signals`: the preferred semantics for `Signal` should be a truly infinite, coinductive *stream*. The authors mention a parallel effort of them to embed Hawk in the Isabelle theorem prover, which achieved the desired semantics. In Π-Ware, we make use of *coinductive streams* [14] to model the inputs and outputs of synchronous sequential circuits, exactly as the authors of Hawk desired.

- The type class system of Haskell is limited: the authors mention the desire to be able to explicitly provided specific instances at specific sites, and also the desire to use *views* on datatypes. Both features are available on *Agda*.

Finally, we would like to mention Cλash [2]. Even though it is not (strictly speaking) an *embedded* DSL, it has very powerful features.

Cλash was developed at the University of Twente, as an independent DSL, i.e, with a compiler of its own. However, its source language is strongly based on Haskell, and Cλash's compiler reuses several pieces of machinery from Haskell's toolchain (specifically, GHC), in order to perform its term-rewriting and supercompilation.

The circuit models in Cλash are higher-order, polymorphic functional programs, and they can be synthesized to VHDL netlist. In this sense, Cλash serves as a good *target* with which to compare hardware EDSL development efforts: it is not -- in itself -- embedded, but has the same goals.

[6] `http://hackage.haskell.org/package/parameterized-data-0.1.5`

2.3 Dependently-Typed Programming

2.3.1 Type systems

A type system, in the context of programming languages, serves the purpose of grouping values, so that meaningless and potentially undesirable operations are avoided. For example, it would be silly to add an integer number and a character string. In fact, it is hard to imagine an addition operation accepting such arguments and having nice algebraic properties. To define addition sensibly, therefore, we need the help of a type system, to ban all programs that would try to add these incompatible values.

Type systems can have very different properties and be implemented in very different ways. Some of the ways in which type systems can be categorized are [6]:

- Weak vs. strong
- Static vs. dynamic
- Polymorphism (parametric, ad-hoc, subtyping)

The most basic form of abstraction -- values depending on values (the concept of functions) -- is supported in practically all type systems. In some systems, the result type of functions can also depend on the *types* of the arguments; this property is called *polymorphism*. There are two kinds of polymorphism relevant to our discussion:

- In *parametric polymorphism*, functions are written without mentioning any specific type, and can therefore be applied to any instantiation of the type variable. A typical example of a parametric polymorphic function is obtaining the length of a list, where the same definition works for any type of element in the list.
- In *ad-hoc polymorphism*, the behaviour of a function varies with the type of the inputs. In Haskell, ad-hoc polymorphism is implemented in the *type class system*. A typical example of an ad-hoc polymorphic function is a comparison-based sorting algorithm in which -- depending on the type of the elements in the collection -- different comparison operators are be used.

Polymorphism adds *expressivity* and makes a type system stronger, in the sense that it allows for more precise specifications. For example, if we want to implement a "swap" function for pairs in Haskell, we could do start by specifying the type of the function in a non-polymorphic way:

```
swap' :: (Int , Int) -> (Int , Int)
```

There are several definitions satisfying the above type which do not swap the elements. For example, one "wrong" implementation would output a constant pair:

```
swap' _ = (1 , 1)
```

Notice how the argument is ignored (we use the "don't care" pattern). To *rule out* this class of wrong implementations, we could make the type polymorphic:

```
swap'' :: (a , a) -> (a , a)
```

Now any "constant" definition will *not anymore fit the type specified*. This because the only way to get an element of the *polymorphic type* `a` is to use what is in the parameter passed to the function. However, we could still write a wrong function with this type, namely the identity:

```
swap'' (x , y) = (x , y)
```

This is possible because our type does not *yet* reflect a precise enough specification of `swap`. On the type we have now, the types of both elements in the pair are the same. If we lift this artificial restriction, we get the type signature which *fully specifies* `swap`.

```
swap :: (a , b) -> (b , a)
```

Now, *any definition with this type* is a correct definition of swap. Because of the way in which we use type variables (`a` and `b`) in the signature, there is only one possible implementation of `swap`:

```
swap (x , y) = (y , x)
```

Going one step further, some type systems support types that depend on other types, these are called *type operators* or *type-level functions*. In Haskell, this form of abstraction is implemented as regular parameterized type constructors (`List`, `Maybe`, etc.) and as *indexed type families*.

The last step then in our "ladder" of type system expressiveness is *dependent types*.

2.3.2 Dependent types

A *dependent type* is a type that depends on a value. A typical example of dependent type is the type of *dependent pairs*, in which the *type* of the second element depends on the *value* of the first:

```
record Σ (α : Set) (β : α → Set) : Set where
  constructor _,_
  field
    fst : α
    snd : β fst
```

In a dependent type system, we can also have functions in which the return *type* depends on the *value* of a parameter. These functions belong to the so-called *dependent function space*.

For example, we can imagine having a take function for vectors which makes use of this possibility to have a more precise specification. First of all, when indexing or obtaining a prefix from a vector, we need to ensure that the index (or amount to be extracted) is within bounds. That is, we cannot take more elements than the size of the vector.

To achieve this goal, we define a datatype (Fin n) of natural numbers smaller than n.

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n} → Fin n → Fin (suc n)
```

With this definition, Fin 0 is a type with no elements, while Fin 1 has 1 element (zero), Fin 2 has 2 elements (zero and suc zero), and so forth…Thus, the elements of Fin n correspond to natural numbers smaller than n.

Now, having defined the Fin n datatype, we can write the type signature for take:

```
take : {α : Set} {n : ℕ} (k : Fin (suc n)) → Vec α n → Vec α (toℕ k)
```

Listing 3: A "safe" prefix-taking function for sized vectors.

On the signature of take, dependent types are used both to restrict the *domain* ($k \leq n$) and to express a desired property of the *result* (length of the returned vector should be k). Finally, we observe that the type of take is not a *sufficient condition* for what we would intuitively conceive as being a "correct" prefix-taking function. There are "wrong" implementations which still would have this type, for example returning a constant vector of size k. This type is, however, provides many more *static guarantees*: All implementations violating bounds and producing incorrectly-sized results are ruled out *by the type checker at compile-time*.

Until now, we have approached languages with dependent types as a way to help with *programming*. Specificaly, we gave examples on how using dependent types in function's type signatures can rule out incorrect implementations *by design*. Dependently-typed languages can also be seen from a *logical* point of view.

This remark -- that a programming language (with it's type system) can also be seen as a logic -- goes back to the early days of computing, and is known by several names, among which "Curry-Howard isomorphism" and "propositions as types" [23]. It initially was "discovered" as a connection between the simply-typed lambda calculus and intuitionistic propositional logic. As decades went by, this correspondence was found to be much more general, and several connections were drawn between diverse typed lambda calculi and logics.

The system of dependent types which we use in this thesis (the one used by *Agda*) corresponds to *intuitionistic predicate logic*.

Even though there are logics connected to other -- less powerful -- typed lambda calculi, the one used in Agda is expressive enough to radically improve its area of use. In a language with dependent types, we can not only write *programs*, but also *proofs*.

For example, in Agda, we can model the less-than-or-equal order relation using an *indexed data family* 4, where the indices are the usual Peano naturals.

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : {n : ℕ}  → zero ≤ n
  s≤s : {m n : ℕ} → m ≤ n → suc m ≤ suc n
```

Listing 4: Order relation ($\leq$) over naturals, as an *Agda* indexed data family.

Having defined this relation, we can prove several useful properties of it: for example, the fact (5) that this relation is transitive.

```
≤trans : {a b c : ℕ} → a ≤ b → b ≤ c → a ≤ c
≤trans z≤n _ = z≤n
≤trans (s≤s ab′) (s≤s bc′) = s≤s (≤trans ab′ bc′)
```

Listing 5: Proof that the $\leq$ relation is transitive.

As proofs are first-class citizens, they can be passed as arguments to functions, and returned as well. This provides yet another powerful way of expressing requirements of function arguments, and showing that the result returned satisfies some desired property.

For example, we have already shown (in listing 3) how to have a "safe" version of a prefix-taking function for statically-sized arrays: by using a specially-designed type for the desired amount. Now we have another way to express the same requirement: we can explicitly require a proof argument be passed, guaranteeing that the amount to be "taken" is *less than or equal* to the array size.

$$\mathsf{take}' : \{\alpha : \mathsf{Set}\}\ \{n : \mathbb{N}\}\ (m : \mathbb{N})\ \{p : m \leq n\} \to \mathsf{Vec}\ \alpha\ n \to \mathsf{Vec}\ \alpha\ m$$

This capability for precisely defining and enforcing requirements and invariants make dependently-typed programming languages very good candidates for hosting EDSLs. In the paper "The power of Pi" [17], three examples are shown of how to embed DSLs in Agda and get corresponding *Domain-Specific Type Systems* (equipped with desirable properties) "for free".

The examples in "The power of Pi" -- specially the embedding of *Cryptol*, a low-level cryptographic DSL -- served as inspiration, and lead us to investigate how dependently-typed programming can benefit hardware design, the main question targeted by this project.

2.4 Hardware and dependent types

We are aware of three attempts in the literature of bringing dependent types to improve hardware design. Each of them has provided us with specific insights and ideas, some of which Π-Ware incorporates. Let us now briefly review these works, and highlight the most important contributions of each.

The paper "Constructing Correct Circuits" [5], from 2007, gives a clear example of how dependent types can *tie together* specification and implementation. In this paper, the authors give a mapping between the usual, Peano-based, naturals and binary numbers, which they then use to build a (ripple-carry) binary adder which is *correct by construction*.

- Elaborate on [8]
- Describe Coquet and elaborate on which points Π-Ware borrows from it.

Chapter 3

The Π-Ware library

Π-Ware is an EHDL that allows for circuit description (modelling), simulation, reasoning (proving correctness or other properties), and synthesis to netlists. In this chapter we will describe in detail how the Π-Ware library is organized, what principles are behind some of the most important design decisions taken in its development, and how to use Π-Ware to model, simulate and reason about circuit behaviour.

The reader is assumed to be familiar with the *Agda* programming language, as this is the language in which Π-Ware is embedded. An introductory-level knowledge of dependent type theory in general is also appreciated.

3.1 Circuit Syntax

The most basic activity allowed by Π-Ware is the *description* of circuits: as already mentioned briefly in the introduction, Π-Ware is *deeply* embedded, which means there is a *datatype* whose values are circuits.

A deep embedding was chosen in order to allow for semantics other than execution (simulation). Particularly, the possibility of *synthesizing* circuit models was a requirement kept in mind throughout the whole development.

The circuit datatype ($\mathbb{C}'$) is the most *fundamental* of the whole library. It is defined as an dependent inductive family, indexed by two natural numbers, as shown in Listing 6.

```
data ℂ′ where
  Nil    : ℂ′ zero zero
  Gate   : (g# : Gates#)  → ℂ′ (ins g#) (outs g#)
  Plug   : ∀ {i o}        → (f : Fin o → Fin i) → ℂ′ i o

  DelayLoop : ∀ {i o l} (c : ℂ′ (i + l) (o + l)) {p : comb′ c} → ℂ′ i o

  _⋙′_    : ∀ {i m o}       → ℂ′ i m   → ℂ′ m o   → ℂ′ i o
  _|′_    : ∀ {i₁ o₁ i₂ o₂} → ℂ′ i₁ o₁ → ℂ′ i₂ o₂ → ℂ′ (i₁ + i₂) (o₁ + o₂)
  _|+′_   : ∀ {i₁ i₂ o}     → ℂ′ i₁ o  → ℂ′ i₂ o  → ℂ′ (suc (i₁ ⊔ i₂)) o
```

Listing 6: The core circuit type ($\mathbb{C}'$) of Π-Ware.

A *structural representation* of a circuit is achieved by the constructors of $\mathbb{C}'$. This is in stark contrast with the the description style used in the Lava family, for example. In Lava, the constructors of the circuit datatype represent solely logic (or arithmetic) gates, and metalanguage (Haskell) constructs such as application, tupling and local naming are used to represent sequencing, parallel composition, loops and sharing. In Π-Ware, on the other hand, explicit constructors represent these combinations, avoiding the need to implement some form of *observable sharing* [10].

The indices of $\mathbb{C}'$ represent, respectively, the *size* of the circuit's input and output. This can be thought of as the number of "wires" entering (resp. leaving) that circuit. Notice that the representation of inputs and outputs used in $\mathbb{C}'$ is untyped and unstructured. However, there is another -- higher-level -- circuit datatype ($\mathbb{C}$), which adds a layer of typing *on top* of $\mathbb{C}'$, and constitutes the actual intended "interface" between the user (hardware designer) and Π-Ware. This abstraction layer will be discussed in more detail on Section 3.2.3.

To better understand the reasoning behind the design of the low-level $\mathbb{C}'$ datatype, its constructors can be considered to belong to one of three categories:

Fundamental: These construct "predefined", or "atomic" circuits, with no sub-components.

- Nil: The *empty circuit*. Performs no computation and has neither inputs nor outputs. It is mainly useful as a "base case" when building large circuits with recursive definitions.
- Gate: Constructs a chosen circuit among those provided by a *gate library* passed as parameter to the Circuit module.
- Plug: Constructs a "rewiring" circuit. Performs no computation, but can be used to apply permutations, associativity and to duplicate or discard wires.

Structural: Represent ways in which smaller circuits can be interconnected to build a bigger one.

- $c_1 \ggg' c_2$: Sequential composition. Given c_1 and c_2, connects the output of c_1 into the input of c_2.
- $c_1 |' c_2$: Parallel composition. Splits the input into two parts, connected to c_1 and c_2, and rejoins the outputs of c_1 and c_2 into a single output.
- $c_1 |+' c_2$: Tagged branching. Based on the value of a *tag* given in one of the input wires, feed the remaining input wires into *either* c_1 or c_2.

Delay: The DelayLoop constructor belongs to a category of its own. Its purpose is to construct a state-holding circuit given a purely combinational circuit as argument.

These descriptions are just a rough summary of the *synthesis semantics* of Π-Ware, that is, how each circuit constructor creates a netlist, given netlists as arguments. The precise *definitions* of the synthesis semantics is given in Section 3.3. The same section also contains a detailed definition and explanation of a simulation semantics for Π-Ware circuits, both purely combinational and sequential ones.

3.1.1 Design discipline enforced by circuit constructors

The several constructors of $\mathbb{C}'$ "calculate" the port sizes of the circuits they construct, based on the sizes of the circuits given as arguments. These calculations implement *structural well-formedness* rules for circuits. One example of such rule can be seen in the case of parallel composition:

$$\_|'\_ \;:\; \forall\ \{i_1\ o_1\ i_2\ o_2\} \;\to \mathbb{C}'\ i_1\ o_1 \to \mathbb{C}'\ i_2\ o_2 \to \mathbb{C}'\ (i_1 + i_2)\ (o_1 + o_2)$$

In this case, the input (resp. output) size of the composed circuit equals the *sum* of the input (resp. output) sizes of the constituent sub-circuits. Other such rules are also imposed by _⟫'_, _|+'_ and Plug. Together, all of them ensure:

No floating wires Circuit sizes need to match in order for the usage of a constructor to be type-correct. In particular, no circuit *input* wires are ever left "disconnected". Figure 3.1 illustrates an example of a situation banned by the sizing rule present in the type of _⟫'_.

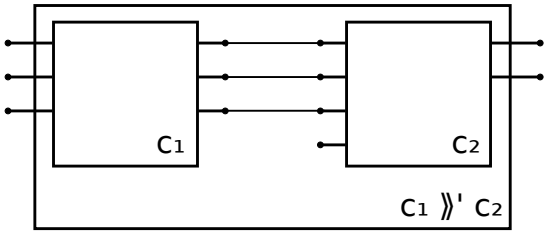

Figure 3.1: Example of circuit banned by the type of _⟫'_.

No short-circuits The Plug constructor, the only one which can be used for "rewiring" purposes, has a type which forbids connecting multiple sources to the same load. Its argument is a *function from outputs to inputs*. In this way, definitions connecting multiple inputs to the same output are banned, as they are *not* functions from outputs to inputs. Also, when a plug is used between two sub-circuits c_1 and c_2, a definition in which an *input* of c_2 would be left "disconnected" is disallowed by Agda, as such a definition would not be *total*. The diagram on Figure 3.2 represents such a banned situation:

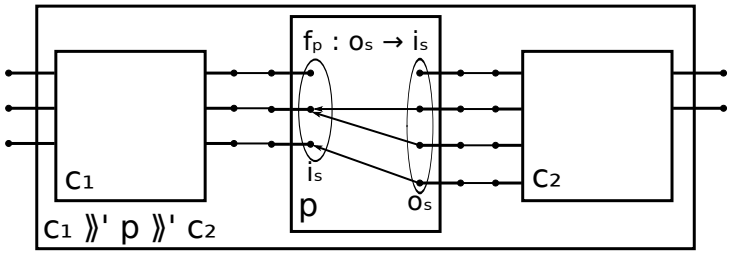

Figure 3.2: This circuit cannot be constructed because f_p is not *total*.

Purely combination vs. possibly sequential

Another piece of information "carried" together with circuits is whether it is *purely combinational*, i.e., has no internal state, no notion of "history", and the value of its current outputs depend only on the value of its current inputs.

This information is represented by a metalanguage truth value, an Agda Set which can be either ⊤ (the Set with one value -- "top") or ⊥ (the Set with no values -- "bottom").

```
comb′ Nil           = ⊤
comb′ (Gate _)      = ⊤
comb′ (Plug _)      = ⊤
comb′ (DelayLoop _) = ⊥
comb′ (c₁ ⋙′ c₂)    = comb′ c₁ × comb′ c₂
comb′ (c₁ |′ c₂)    = comb′ c₁ × comb′ c₂
comb′ (c₁ |+′ c₂)   = comb′ c₁ × comb′ c₂
```

Listing 7: Predicate telling whether a low-level circuit is purely combinational.

The predicate comb′ over low-level ($\mathbb{C}'$) circuits defines when is a circuit purely combinational. The code for comb′ is shown on Listing 7.

Notice that the fundamental constructors Nil, Gate and Plug *always* build purely combinational circuits, while DelayLoop always produces circuits with internal state. The structural constructors (_⋙′_, _|′_ and _|+′_) yield a purely combinational circuit only when *both* of its argument are themselves purely combinational. This is expressed by using the product type constructor _×_, which in the Set level corresponds to a logical conjunction.

This categorization of circuits as stateful or stateless has one main goal: To avoid the evaluation of *combinational loops*. Their presence in a circuit is almost always a design mistake, and by combining the DelayLoop constructor with the comb′ predicate we guarantee that:

- The only way to create a loop in a circuit (DelayLoop) also introduces a delay.
- Combinational evaluation (ignoring past inputs) only happens when safe.

3.2 Abstraction Mechanisms

Several of the definitions (modules, datatypes, functions) of Π-Ware are parameterized in order to provide for better code reuse. The library allows for a choice of the type of data carried in individual wires, the types that serve as inputs and ouputs to circuits, and the set of fundamental gates from which circuits are built.

In this section we present how this flexibility is achieved, which requirements are imposed upon the parameters in each case of parameterization. Also, the motivation behind the chosen requirements is presented, based on the general goals of Π-Ware as well as the wish to impose as few constraints as possible.

3.2.1 Atoms

In EDSLs of the current Lava family [7], only values of type `Bool` and `Int` can be carried by the "wires", and circuits can only operate on inputs and outputs which are tuples or lists of these basic types.

There is a little more flexibility in ForSyDe [20], where the `ProcType` type class determines what values of what types can be used as inputs and outputs of *process functions* (combinational circuits). ForSyDe has predefined instances of `ProcType` for fixed-width numeric types (`Int8`, `Int16`, etc.) and `Bool`, while relying on *Template Haskell* to generate instances for user-defined *enumerations* at compile time.

Π-Ware's approach is similar to ForSyDe's, in that it shares the same goal (to "carry" in the wires any type that belongs to a type class). However, Π-Ware does not use any metaprogramming, and uses dependent types to ensure that the provided instances satisfy desired properties. The type class Atomic (shown in Listing 8) expresses in Π-Ware what are the requirements for a type to be carried in the wires of a circuit.

```
record Atomic : Set₁ where
  field
    Atom        : Set
    |Atom| − 1  : ℕ
    n → atom    : Fin (suc |Atom| − 1) → Atom
    atom → n    : Atom → Fin (suc |Atom| − 1)

    inv − left  : ∀ i → atom → n (n → atom i) ≡ i
    inv − right : ∀ a → n → atom (atom → n a) ≡ a

  |Atom| = suc |Atom| − 1
  Atom#  = Fin |Atom|
```

Listing 8: The Atomic type class.

The fields Atom and |Atom| − 1 determine, respectively, the Agda Set to be used as basis and *how many elements* of this Set are to be used as atoms (the number of atoms is actually |Atom| = suc |Atom| − 1).

Then there are the fields n → atom and atom → n, which define a *bijection* between the Atom type and Fin |Atom| (naturals from zero to |Atom|). Finally, the two last fields in Atomic are proofs that n → atom and atom → n are inverses of each other, therefore proving that they are also bijections.

Instance for `Bool`

Included with Π-Ware there is already a predefined instance of Atomic for Bool (the boolean type in Agda's standard library). Knowing the meaning of each field, the definitions comprising the instance are mostly easy to understand. First, we start by defining the cardinality of the set of values we are interest in (B is used as synonym for Bool):

```
|B| − 1 = 1
|B| = suc |B| − 1
```

Now, before defining the bijections that enumerate the set of atoms, we give more convenient *names* to the elements of Fin |B| used in the enumeration by using an Agda feature known as *pattern synonyms*:

```
pattern False#  = Fz
pattern True#   = Fs Fz
pattern Absurd# n = Fs (Fs n)
```

The usage of the Absurd# pattern becomes clearer when we take a look at the definitions of the bijections themselves:

```
n → B : Fin |B| → B
n → B = λ { False# → false;  True# → true;  (Absurd# ()) }

B → n : B → Fin |B|
B → n = λ { false → False#;  true → True# }
```

In the case of the Absurd# pattern, there is no need to provide a right-hand side to the equation, as the Agda typechecker can verify that there is no x such that Fs (Fs x) belongs to Fin 2.

Finally, there are the proofs stating that n → B has a two-sided inverse (namely, B → n), and, therefore, it is a bijection. The proofs use simple case analysis (and reflexivity).

```
inv − left − B : ∀ i → B → n (n → B i) ≡ i
inv − left − B = λ { False# → refl;  True# → refl;  (Absurd# ()) }

inv − right − B : ∀ b → n → B (B → n b) ≡ b
inv − right − B = λ { false → refl;  true → refl }
```

Other possibly interesting instances

Even though currently Bool (from the Agda standard library) is the only type for which there is already a predefined instance of Atomic, other types could be interestingly used as atoms. For example:

Three-valued logic types Representing the values `TRUE`, `FALSE` and `UNKNOWN`

IEEE1164 nine-valued logic Standardized logic used widely in industry as a VHDL/Verilog type. Besides the usual `FALSE` and `TRUE` logical values (represented respectively by '0' and '1') it accomodates also the following others:

- `'Z'`: *High impedance*, means output is not being driven.
- `'X'`: *Strong drive, unknown logic value.*
- `'W'`: *Weak drive, unknown logic value.*
- `'L'`: *Weak drive, logic zero.*
- `'H'`: *Weak drive, logic one.*
- `'U'`: *Uninitialized.* Used as default initial value in simulation.
- `'-'`: *Don't care.*

3.2.2 Gate library

Besides being parameterized by the type of Atoms that can be carried in their wires, Π-Ware low-level circuit descriptions (ℂ′) are also parameterized by a *library of fundamental gates* upon which circuits are built.

There are several interesting possible gate libraries which could be used to describe circuits, among which:

- *Functionally complete* sets of boolean functions such as {`NAND`}, {`NOR`} or {`NOT`, `AND`, `OR`} (predefined in the Π-Ware library).

- Arithmetic primitives over boolean vectors, such as {_+_, _*_}.
- A logic-arithmetic library together with some purpose-specific primitives, for example cryptographic and Digital Signal Processing (DSP) functions.

A gate library is defined as an element of the Gates record type, shown in Listing 9.

```
W : ℕ → Set
W = Vec Atom

record Gates : Set where
  field
    |Gates| − 1  : ℕ
    ins outs     : Fin (suc |Gates| − 1) → ℕ
    spec         : (g : Fin (suc |Gates| − 1)) → (W (ins g) → W (outs g))

  |Gates|  = suc |Gates| − 1
  Gates#   = Fin |Gates|
```

Listing 9: Definition of a gate library: the Gates record.

The first field of the Gates record (|Gates| − 1) determines *how many* gates the library has, and consequently also the *the type used as gate index* (or "gate identifier"), which is Gates# = Fin |Gates|.

For each gate identifier, the functions ins and outs determine, respectively, the size of the input and output of that gate; therefore together they determine the gate's *interface*.

Finally, the spec field determines the *functional specification* of a given gate identifier. Notice how the whole Gates module is parameterized by an instance of Atomic. In this way, the spec function yields, given a gate identifier, a function over *words* (Atom vectors) of the correct length. This functional specification of each fundamental gate is used for the *simulation semantics*.

Assumed correctness

Later, in Section 3.4, we will give a precise definition for what is considered the *functional correctness* of a circuit; its *compliance* to a certain functional specification.

Usually, the proof of functional correctness for a circuit is built using, in some way, the correctness proofs of its subcomponents. However, fundamental gates have no subcomponents, so their correctness is assumed. By definition, the simulation semantics of a fundamental gate is equal to the spec function given for it in the gate library definition.

Relation to synthesis

Even though fundamental gates are "atomic" (have no subcomponents) *at the Π-Ware level*, they are not in any way directly synthesizable to VHDL. To convert a Π-Ware circuit to VHDL, each fundamental gate in the library being used needs to have a corresponding excerpt of VHDL code representing it.

This feature is not integrated in the current version of Π-Ware, but assuming the existence of a hierarchy of datatypes representing VHDL code, and specifically the existence of a type VHDLEntity, we could have an extra field in the Gates record, as such:

```
netlist : (g : Fin (suc |Gates| − 1)) → VHDLEntity (ins g) (out g)
```

The VHDLEntity is also parameterized by the input and output sizes of the circuit, tying the Π-Ware fundamental gate interface with the VHDL abstract syntax.

Instance for `BoolTrio`

One specific instance of gate library is already "shipped" with Π-Ware: the usual set of boolean gates {`NOT`, `AND`, `OR`}, together with the constants {`FALSE`, `TRUE`}. The first definitions establish that this library contains five gates:

```
|BoolTrio| − 1 = 4
|BoolTrio|     = suc |BoolTrio| − 1
```

Then, as already done in the instance of Atomic, we define some *pattern synonyms* to make the gate identifiers easier to read:

```
pattern FalseConst#  = Fz
pattern TrueConst#   = Fs Fz
pattern Not#         = Fs (Fs Fz)
pattern And#         = Fs (Fs (Fs Fz))
pattern Or#          = Fs (Fs (Fs (Fs Fz)))
pattern Absurd# n    = Fs (Fs (Fs (Fs (Fs n))))
```

The next step is to define the *interface* of each gate on the library. All gates have exactly one output, while the number of inputs vary from 0 to 2.

```
ins outs : Fin |BoolTrio| → ℕ
ins = λ { FalseConst# → 0; TrueConst# → 0; Not# → 1; And# → 2; Or# → 2; (Absurd# ()) }
outs _ = 1
```

Finally, the last piece of information needed to define the gate library is the functional specification of each gate.

```
spec : (g : Fin |BoolTrio|) → (W (ins g) → W (outs g))
spec FalseConst#  = const [ false ]
spec TrueConst#   = const [ true  ]
spec Not#         = λ { (x :: ε)       → [ not x ] }
spec And#         = λ { (x :: y :: ε)  → [ x ∧ y ] }
spec Or#          = λ { (x :: y :: ε)  → [ x ∨ y ] }
spec (Absurd# ())
```

Notice that the gates use Bool from Agda's standard library as their Atom type. Also the logic operators from the standard library (Data.Bool) are used in the specification. With all the necessary pieces of the puzzle in hand, the definition of the BoolTrio gate library is as follows:

```
BoolTrio : Gates
BoolTrio = record {
  |Gates| − 1 = |BoolTrio| − 1
  ; ins       = ins
  ; outs      = outs
  ; spec      = spec
  }
```

3.2.3 High-level circuit datatype

When presenting the low-level circuit datatype ($\mathbb{C}'$) on Section 3.1, it was mentioned that the representation using $\mathbb{C}'$ is untyped and unstructured, i.e., all circuit inputs and outputs at that level are just sequences of Atoms (for example, bits).

However, the user of Π-Ware is not supposed to actually model circuits using the $\mathbb{C}'$ datatype. There is a layer of abstraction sitting on top of $\mathbb{C}'$, and circuits at this higher level have inputs and outputs with Agda types. This layer is coded as the $\mathbb{C}$ datatype, shown in Listing 10.

```
data ℂ (α β : Set) {i j : ℕ} : Set where
  Mkℂ : ⦃ sα : ⇓ W ⇑ α {i} ⦄ ⦃ sβ : ⇓ W ⇑ β {j} ⦄ → ℂ' i j → ℂ α β {i} {j}
```

Listing 10: High-level circuit datatype ($\mathbb{C}$).

First of all, notice the *lack of a prime symbol in the name of the datatype*. This is a general naming convention: whenever there are two versions of a definition, one at a low level of abstraction and one higher, the low-level gets the same name of the high-level one, with a prime symbol as suffix.

In contrast to the low-level ($\mathbb{C}'$) circuit type, $\mathbb{C}$ is parameterized by two Agda Sets, and has only one constructor - Mk$\mathbb{C}$ - which "packs" the low-level circuit description with the information on how to convert between the input/output Agda types (resp. α and β) and the appropriately-sized *words* used in the low level. This conversion information is coded in the ⇓ W ⇑ type class (pronounced "Synthesizable"), which is covered in more detail on Section 3.2.4.

Furthermore, the PiWare.Circuit module defines also *smart constructors* for $\mathbb{C}$, corresponding to each of the constructors at the low level. One example is parallel composition:

```
_||_ : ∀ {α i γ k β j δ l} → ⦃ sα : ⇓ W ⇑ α {i} ⦄ ⦃ sγ : ⇓ W ⇑ γ {k} ⦄
                           → ⦃ sβ : ⇓ W ⇑ β {j} ⦄ ⦃ sδ : ⇓ W ⇑ δ {l} ⦄
  → ℂ α γ {i} {k} → ℂ β δ {j} {l} → ℂ (α × β) (γ × δ) {i + j} {k + l}
_||_ ⦃ sα ⦄ ⦃ sγ ⦄ ⦃ sβ ⦄ ⦃ sδ ⦄ (Mkℂ c₁) (Mkℂ c₂) =
  Mkℂ ⦃ ⇓ W ⇑ −× ⦃ sα ⦄ ⦃ sβ ⦄ ⦄ ⦃ ⇓ W ⇑ −× ⦃ sγ ⦄ ⦃ sδ ⦄ ⦄ (c₁ |' c₂)
```

Using a *gate library* together with these smart constructors, one can model hardware circuits operating over Agda types. One very simple example is a `XOR` circuit (shown in Listing 11), built using the `BoolTrio` gate library mentioned before. The block diagram in Figure 3.3 illustrates the architecture of such circuit.

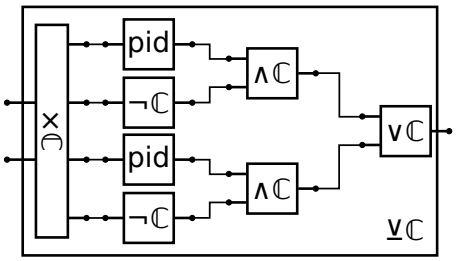

Figure 3.3: Architecture of a sample `XOR` gate.

```
⊥ℂ ⊤ℂ : ℂ ⊤ B
⊥ℂ = Mkℂ (Gate FalseConst#)
⊤ℂ = Mkℂ (Gate TrueConst#)

¬ℂ : ℂ B B
¬ℂ = Mkℂ (Gate Not#)

∧ℂ ∨ℂ : ℂ (B × B) B
∧ℂ = Mkℂ (Gate And#)
∨ℂ = Mkℂ (Gate Or#)

⊻ℂ : ℂ (B × B) B
⊻ℂ =  pFork×
   ⟫ (¬ℂ || pid ⟫ ∧ℂ) || (pid || ¬ℂ ⟫ ∧ℂ)
   ⟫ ∨ℂ
```

Listing 11: Example of a `XOR` gate built with the `BoolTrio` library.

Again in this example, as customary, we renamed Bool to B, and pid is the *identity plug*. The identity plug can be substituted in the netlist by a bus (with the adequate sizes), and is defined (together with other useful plugs) in the PiWare.Plugs module.

3.2.4 The `Synthesizable` type class

The "connection" between the low-level ($\mathbb{C}'$) and high-level ($\mathbb{C}$) circuit types is done by the ⇓ 𝕎 ⇑ type class (pronounced "Synthesizable"). Instances of the ⇓ 𝕎 ⇑ class define a mapping between a given Agda type and an appropriately-sized *word* type. Word types (𝕎 *n*, for some *n*) are *synthesizable* to VHDL vectors, thus the name of the class. Listing 12 shows the definition of ⇓ 𝕎 ⇑:

```
W : ℕ → Set
W = Vec Atom

record ⇓ W ⇑ (α : Set) {i : ℕ} : Set where
   constructor ⇓ W ⇑ [_,_]
   field
      ⇓ : α → W i
      ⇑ : W i → α

open ⇓ W ⇑ {{ ... }}
```

Listing 12: The ⇓ 𝕎 ⇑ (Synthesizable) type class.

The type class has two parameters: the type which it encodes (α), and the size of the *word type* to which α corresponds (i). Notice that the size parameter is passed *implicitly*, because in some occasions Agda might be able to automatically infer (by unification) this value.

One can imagine that all *finite types* (types with finitely many elements) can be made synthesizable under the definition of ⇓ 𝕎 ⇑, given a certain encoding scheme. We do

not give a proof of this statement, but one way of doing it would be to translate Agda datatype definitions into a sum-of-products view and then use a standardized encoding scheme (for example, some form of *Church encoding*).

Π-Ware does *not* provide facilities for mapping arbitrary Agda datatypes into a sum-of-products view (we believe this is the domain of a *generic programming* library, not of a hardware EDSL). However, Π-Ware *does* provide predefined instances of ⇓ 𝕎 ⇑ for units (⊤), booleans (Bool), products (_ × _), vectors (Vec) and sums (_ ⊎ _), in order to facilitate working with complex types when modelling hardware.

One interesting instance to look at is the one for Agda's standard library vectors (Vec), shown in Listing 13.

```
⇓ W ⇑ -Vec : ∀ {α i n} → ⦃ sα : ⇓ W ⇑ α {i} ⦄ → ⇓ W ⇑ (Vec α n)
⇓ W ⇑ -Vec {α} {i} {n} ⦃ sα ⦄ = ⇓ W ⇑ [ down , up ]
  where down : Vec α n → W (n * i)
        down v = v >>= ⇓

        up : W (n * i) → Vec α n
        up w = map_v ⇑ (proj_1 $ group n i w)
```

Listing 13: Predefined instance of ⇓ 𝕎 ⇑ for fixed-length vectors.

3.3 Circuit Semantics

Functional semantics for simulation…

- Two types of evaluation: combinational and sequential
- Combinational eval requires a proof that the circuit contains no loops

  Eval of a fundamental gate is just its *definitional behaviour*
- For sequential circuits we use a *causal stream semantics*

  Current output depends on the current input and (possibly) on the past inputs

  Different than plain Stream functions
- There's also an eval function which allows the circuit to be viewed as function from Stream to Stream

3.3.1 Combinational evaluation

3.3.2 Sequential evaluation

Initially we tried to write a semantics yielding simply a function over streams.

But this is not quite a good semantic model for a synchronous sequential circuits, as a stream function is allowed to "look into future values" in order to produce the output stream, while a sequential circuit can only use information about its past inputs when calculating the next output.

Therefore, causal streams…

3.4 Proving circuit properties

Chapter 4

Conclusions

- Reiterate what was achieved
- Prioritize what still needs to be done, or would be nice if done

4.1 Discussion and Related Work

Lorem ipsum…

4.2 Future work

- What problems *Π-Ware* still has.

  With suggestions for how to solve them, if possible
- More broadly, considering *Π-Ware*, what would be the ``next steps".

Bibliography

[1] Ieee standard vhdl synthesis packages. *IEEE Std 1076.3-1997*, pages i--, 1997.

[2] Christiaan Baaij and Jan Kuper. Using rewriting to synthesize functional languages to digital circuits. In Jay McCarthy, editor, *Trends in Functional Programming*, volume 8322 of *Lecture Notes in Computer Science*, pages 17--33. Springer Berlin Heidelberg, 2014.

[3] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613--641, August 1978.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. *SIGPLAN Not.*, 34(1):174--184, September 1998.

[5] Edwin Brady, James McKinna, and Kevin Hammond. Constructing correct circuits: Verification of functional aspects of hardware specifications with dependent types. *Trends in Functional Programming*, 8:159--176, 2007.

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471--523, December 1985.

[7] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Advances in Computing Science—ASIAN'99*, pages 62--73. Springer, 1999.

[8] Solange Coupet-Grimal and Line Jakubiec. Certifying circuits in type theory. *Formal aspects of computing*, 16(4):352--373, 2004.

[9] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365--376, June 2011.

[10] Andy Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117--128, New York, NY, USA, 2009. ACM.

[11] Bernd Hoefflinger. Itrs: The international technology roadmap for semiconductors. In *Chips 2020*, pages 161--174. Springer, 2012.

[12] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[13] Paul Hudak and Mark P Jones. Haskell vs. ada vs. c++ vs. awk vs.... an experiment in software prototyping productivity. Technical report, Citeseer, 1994.

[14] Bart Jacobs. Introduction to coalgebra. towards mathematics of states and observations. draft of a book, 2005.

[15] John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within haskell. *SIGPLAN Not.*, 34(9):60--69, September 1999.

[16] Simon Marlow et al. Haskell 2010 language report.

[17] Nicolas Oury and Wouter Swierstra. The power of pi. *SIGPLAN Not.*, 43(9):39--50, September 2008.

[18] J. P. Pizani Flor. Comparing functional embedded domain-specific languages for hardware description. Technical report, Utrecht University, 2013.

[19] D. Price. Pentium fdiv flaw-lessons learned. *Micro, IEEE*, 15(2):86--88, Apr 1995.

[20] Ingo Sander and Axel Jantsch. Formal system design based on the synchrony hypothesis, functional models, and skeletons. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 318--323. IEEE, 1999.

[21] Mary Sheeran. mufp, a language for vlsi design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 104--112, New York, NY, USA, 1984. ACM.

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

[23] Philip Wadler. Propositions as types. *Unpublished note*, 2014.

[24] Ulf Wiger and Ericsson Telecom Ab. Four-fold increase in productivity and quality-industrial-strength functional programming in telecom-class products. 2001.