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

Wednesday 20th August, 2014

Contents

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 [16], which means bigger circuits in the same wafer area. On the other hand, advances in Instruction-Level Parallelism (ILP) and microarchitecture of processors are starting to have diminishing returns [14].

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 [26].

A language for hardware design should, therefore, try to prevent as many design mistakes as possible, and *as early as possible* in the implementation chain. Also, it should facilitate the task of verification, providing techniques more efficient than exhaustive testing and model checking for circuits of fixed size.

In the software industry, especially 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 [33] and in experiments [18].

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 earlier 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 [29], developing *functional* hardware description languages, such as *μFP* [28]. Later, a trend emerged of developing EDSLs for hardware description, *hosted* in purely functional languages, such as Haskell [21]. Prominent examples of this trend are the Lava [7] family, as well as ForSyDe [27].

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 a language supporting *dependent types*, as exemplified in the paper "The Power of Pi" [24].

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 bring 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 statically-sized vectors, along with the safe head function, depicted in Listing 1.

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 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:

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

```
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 the safe head function.

```
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 $xs \equiv \text{concat}\ xss$.

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.

Dependent 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.

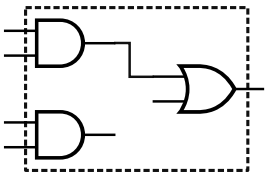

Figure 1.1: Malformed circuit with a *floating* signal.

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 logic).

In Chapter 2 we will establish the research 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. A detailed account is given of how Π-Ware works currently, what design decisions were involved in its development, and how to use it. We give examples of circuits modeled in Π-Ware and proofs of properties involving these circuits.

In Chapter 4 we compare Π-Ware with other functional and dependently-typed EDSLs, highlighting similarities and differences. This is done by commenting on sample circuits and proofs.

In Chapter 5, we conclude by pointing Π-Ware's most important current limitations, what are their causes, and trying to speculate on possible solutions. We also comment on what other related research paths could be pursued involving hardware and dependent types, based on the insights gained with Π-Ware.

Last but not least, all source code and documentation for Π-Ware can be found at `https://github.com/joaopizani/piware`.

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 spatially 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 transformation steps towards a more detailed description, while the ellipsoid shapes represent artifacts that are consumed/produced by each step.

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 several problems 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 [2] 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 (the `if` statement lacks an `else` branch, and the `y` output has no assigned value when `sel` is different than zero).

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

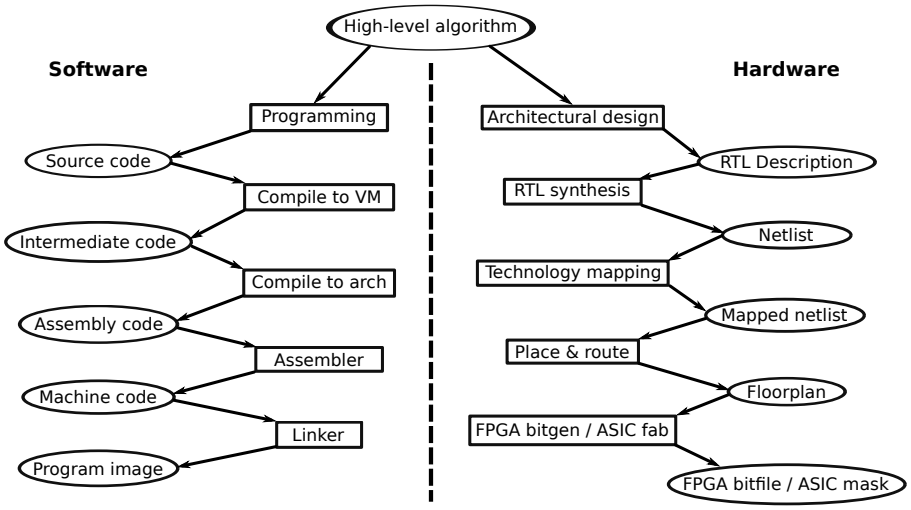

Figure 2.1: Software and Hardware refinement chains.

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

Listing 2: Unsynthesizable VHDL process.

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 as they were designed for other purposes: VHDL for simulation and C/C++ for software development.
- 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 functional 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 [28], which was in turn inspired by John Backus's FP [6]. In contrast to VHDL, which was later adapted to synthesis in an ad-hoc way, µFP was designed since the beginning to have two 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 original µFP paper [28]:

> 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.

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 a single "µ-wrapper".

In other words: a circuit with several, localized, memory elements can be converted into a circuit with a single 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:

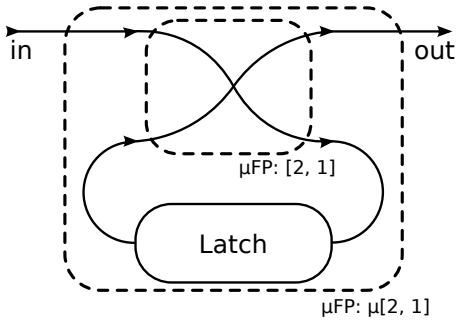

Figure 2.2: *Shift register*: a simple example of sequential circuit in μFP.

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 [17] 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 and a highly-customizable 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.

[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/

Lava

Perhaps the most popular family of hardware DSLs embedded in Haskell is the *Lava* family. Lava's first incarnation [7] was developed at Chalmers University of Technology and Xilinx, and used a shallow-embedding in Haskell, along with a monadic description 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 [11]. 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 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 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`:

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

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

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

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 [27]. 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 [25] of the experimentation project conducted as preparation for this thesis.

In ForSyDe, the central concepts are those of *process* and *signal*. Whereas in Lava only *synchronous* sequential circuits can be described – which 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).

Instead of just using (smart) constructors of a certain *circuit datatype* to build process functions, 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 `ProcFun` 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 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` page on *Hackage*[6], the authors admit that the library's goal 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 [20] 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 [7] 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* [19] 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 [5]. 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.

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

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.

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, in some dynamic languages, a string and an integer can be "added", because the language runtime first *implicitly casts* the integer to a string, effectively performing string concatenation. This *implicit* behaviour can cause all sorts of programmer mistakes to go undetected. It is also hard to reason about addition *in general* when all sorts of coercions can happen at runtime. To define addition sensibly, therefore, a type system can help by banning all programs trying to add 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 [10]:

- 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 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 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 "constant" definitions will *not have the specified type* anymore. This because the only way to get an element of the *polymorphic type* `a` is to use the parameter passed to the function. However, we could still write a wrong function with this type, for example 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 total 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. The type signature of take is shown in Listing 3.

```
take : {α : Set} {n : ℕ} (k : ℕ) → Vec α (k + n) → Vec α k
```

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

In the signature of take, a dependent function space is used both to restrict the *domain* and to express a desired property of the *codomain*. Notice how the vector parameter has a type (Vec α (k + n)) which *depends on the value* of the first (explicit) parameter: k. Also, the result type is restricted to Vec α k.

Finally, we observe that the type of take is a *necessary but not 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, 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*. Specifically, 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 system – goes back to the early days of computing, and is known by several names, among which "Curry-Howard isomorphism" and "propositions as types" [32]. 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* (Listing 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 then prove facts involving it. For example, one trivial fact is that two is smaller-than-or-equal to four, which has the trivial proof shown in Listing 5.

```
twoLEQFour : 2 ≤ 4
twoLEQFour = s≤s (s≤s z≤n)
```

Listing 5: A very simple proof involving the $\leq$ relation over natural numbers.

Following the "propositions as types" isomorphism, the type of twoLEQFour can be seen as a logic statement, and the expression to the right-hand side of the equals sign (the body of the function) as a *proof* of this statement. The proof is built with z≤n as its "basis" (the fact that zero is lesser-than-or equal to any number), producing $0 \leq 2$. Then s≤s is applied twice, producing, respectively, $1 \leq 3$ and finally $2 \leq 4$.

Of course more "interesting" and general statements can be proved. For example, the fact that the $\leq$ relation is *transitive* is stated and proved in Listing 6.

```
≤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 6: 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-crafted input type. 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.

```
take′ : {α : Set} {n : ℕ} (m : ℕ) {p : m ≤ n} → Vec α n → Vec α 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" [24], 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" [8], 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*.

This approach is significantly different than the one taken in Π-Ware. In the paper by Brady, McKinna & Hammond [8], the functional specification of a circuit is carried *in the type* of the circuit. This, according to the authors, means that "a type-correct program is a constructive proof that the program meets its specification".

In Π-Ware, on the other hand, a circuit's type only expresses its input and output types, and gives some *structural well-formedness guarantees* (correct sizing). The development of Π-Ware was already underway as this paper first came to my knowledge. Even so, the decision of keeping circuits and proofs apart (though in the same language) still seems to have been the right one; mainly because of the requirement that circuits be synthesizable to VHDL.

2.4.1 Coquet

The most significant source of inspiration for the design of Π-Ware was Coquet [9], a hardware DSL embedded in the Coq language[7]. Coquet and Π-Ware share some very important characteristics and differ on several others, with these differences caused not only by the different host languages used (Coq and Agda), but also by different implementations chosen for some core concepts.

First of all, the similarities. Coquet's circuit datatype utilizes the same concept of *structural* description as Π-Ware, with a "fundamental" gate constructor and constructors for structural combination. Also, the idea of dedicating one specific constructor for building "rewiring" circuits was inspired by Coquet's `Plug`. The `Circuit` datatype of Coquet is shown on Listing 7.

```
Context {tech : Techno}
Inductive Circuit : Type -> Type -> Type :=
| Atom : forall {n m : Type} {Hfn : Fin n} {Hfm : Fin m},
            techno n m -> Circuit n m

| Plug : forall {n m : Type} {Hfn : Fin n} {Hfm : Fin m} (f : m -> n),
            Circuit n m

| Ser  : forall {n m p : Type},
            Circuit n m -> Circuit m p -> Circuit n p

| Par  : forall {n m p q : Type},
            Circuit n p -> Circuit m q -> Circuit (n + m) (p + q)

| Loop : forall {n m p : Type},
            Circuit (n + p) (n + p) -> Circuit n m
```

Listing 7: Coquet's `Circuit` datatype.

Some differences between Coquet's `Circuit` type and that of Π-Ware (detailed in Section 3.1) are, however, immediately noticeable:

Indices are `Type`s, instead of just sizes In Π-Ware, the core circuit datatype is indexed by two natural numbers, representing the *sizes* of the circuit's input and output. In Coquet, the indices are arbitrary types, *constrained* by the `Fin` *type class*. The definition of `Fin` makes the synthesis of Coquet's circuits *interfaces* to VHDL harder. Before deciding to use natural number indices on Π-Ware, another alternative for generalization was considered, detailed in Section **??**.

[7] Referring to Coq as a language is a misnomer, as it is an interactive theorem prover that uses different languages for defining terms, interactive commands, and user-defined proof tactics

Lack of a "sum" constructor Both Coquet and Π-Ware have a "parallel composition" constructor which, given two circuits as arguments (with inputs respectively α and β), will build a circuit with the *product* $\alpha \times \beta$ as its input. Π-Ware, however, has an extra "sum" constructor, building a circuit which is capable of "deciding" which one of the two subcircuits to apply to an input depending on a *tag*. This is useful for implementing, for example, generic multiplexers.

Single fundamental gate vs. *gate library* Coquet's fundamental constructor (`Atom`) is parameterized by the `techno` instance, which gives the description of **one** technology-dependent gate (`NAND`, `NOR`, etc.). Π-Ware circuit descriptions, however, are parameterized by a *gate library*, with an attached specification function to each of its elements, as detailed in Section 3.2.2.

Another similarity between Coquet and Π-Ware is the usage of *data abstraction*, embodied in Coquet by the `Iso` type class and in Π-Ware by `Synthesizable`, described in more detail in Section 3.2.4.

The goal of data abstraction is to be able to express the specification of circuits in more user-friendly (high-level) types, while the circuits themselves are just described in terms of sizes and wires.

Finally, some of the the more fundamental differences between Π-Ware and Coquet are:

Relational vs. functional semantics Coquet defines a *relational* semantics for circuits, that is, the semantics of a circuit is a datatype defined by recursion on circuit structure. Π-Ware, on the other hand, has an executable *functional semantics*[8], in which each circuit is mapped to an Agda function over sequences of bits (or other types).

Semantics of sequential circuits Π-Ware, in contrast to Coquet, makes a clear distinction between *purely combinational* and *possibly sequential* circuits. We maintain the invariant that *purely combinational circuits never contain loops*. Also, in Π-Ware, the only constructor which builds loops also adds a *delay element*[9], eliminating the need to consider an extra "sequential fundamental gate", as in Coquet. The functional semantics of sequential circuits in Π-Ware is defined as a *causal stream function* (detailed in Section 3.3.3), while Coquet uses regular functions over streams.

[8] another term commonly used in the literature is *denotational* semantics
[9] in hardware terms, a *clocked latch*

Chapter 3

The Π-Ware library

Π-Ware is an embedded hardware description language that allows for circuit description (modelling), simulation, verification and synthesis. 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 and with Dependently-Typed Programming in general. For readers familiar with functional programming, but not Agda nor DTP, a good introduction can be found in the official Agda tutorial [23].

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 a dependent inductive family, indexed by two natural numbers, as shown in Listing 8.

```
data ℂ′ where
  Nil    : ℂ′ zero zero
  Gate   : (g# : Gates#) → ℂ′ (|in| g#) (|out| 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 8: 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* [15] in the host language.

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 through recursive definitions.
- Gate**:** Constructs a chosen circuit among those provided by a *gate library* passed as parameter to the Circuit module. Such libraries can consist of, for example, logic primitives ({NOT, AND, OR}) or even arithmetic gates (adders, etc.).
- Plug**:** Constructs a "rewiring" circuit. Performs no computation, but can be used to apply permutations and to duplicate or discard wires. The type of the function passed as argument to Plug ensures that no short circuit can occur.

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. Part of the output of the circuit given as argument to DelayLoop is passed through a memory element and *looped back* to that circuit's input. State machines and other sequential circuits can be created with this constructor.

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 *definition* of the synthesis semantics is given in Section 3.3. The same section 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:

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

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 $\_\gg'\_$, $\_|+'\_$ 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 $\_\gg'\_$.

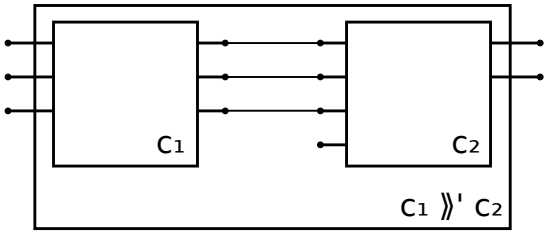

Figure 3.1: Example of circuit banned by the type of $\_\gg'\_$.

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:

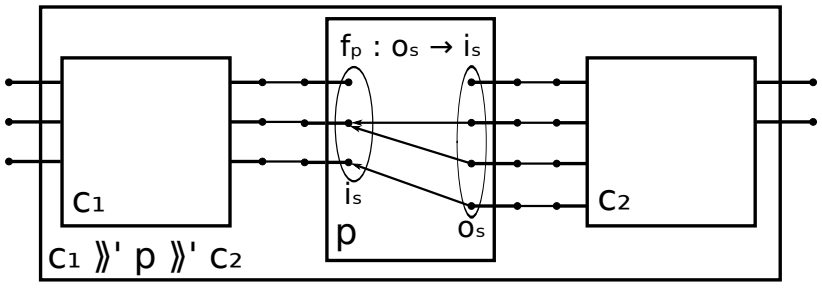

Figure 3.2: This circuit cannot be constructed because f_p is not *total*.

Purely combinational vs. possibly sequential

Circuits in Π-Ware can be *purely combinational* or *sequential*. Combinational circuits have no internal state and can be simulated without regarding past inputs. A very simple semantics is defined over the circuit datatype in order to distinguish these two classes of circuits.

The semantic function is the comb′ predicate, defined over low-level ($\mathbb{C}'$). The comb′ function returns an Agda Set which can be either ⊤ (the Set with one value) or ⊥ (the Set with no values). This returned Set can therefore be interpreted as a *truth value*. Listing 9 shows the code of the comb′ predicate

```
comb′ : {i o : ℕ} → ℂ′ i o → Set
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 9: Predicate telling whether a low-level circuit is purely combinational.

Notice how 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 combinational circuit *if and only if both* of its arguments are themselves combinational. This is expressed by using the product type constructor _×_, which 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. Altera™, for example, explains in their manual of "Recommended Design Practices"[1]:

> Combinational loop behavior generally depends on relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change, which means the behavior of the loop is unpredictable.

In Π-Ware, the semantics of the DelayLoop constructor, and the clear *partitioning* of circuits into combinational or sequential done by the comb′ predicate guarantee that:

- The only way to create a loop in a circuit (DelayLoop) also introduces a delay[2].
- Combinational simulation (ignoring past inputs) only happens when the circuit is provably cycle-free, thus stateless and free of race conditions.

3.2 Abstraction Mechanisms

Several of the definitions (modules, datatypes, functions) of Π-Ware are parameterized in order to improve code reuse. The library allows for a choice of the type of data carried

[1] http://www.altera.com/literature/hb/qts/qts_qii51006.pdf

[2] in hardware terms, a *clocked latch*.

in individual wires, the types that serve as inputs and outputs to circuits, and the set of fundamental gates from which circuits are built.

In this section we present how this flexibility is achieved, and which requirements are imposed upon the parameters in each case of parameterization. Furthermore, we discuss the motivation behind the chosen requirements, 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 [11], 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 [27], where the `ProcType` type class determines what types can be used as input and output types 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 *enumeration types* 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 belonging 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 Atomic record (shown in Listing 10) captures 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 10: 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|−1). 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 defi-

nitions 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 accommodates 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*.

Numeric types For example, some binary type equivalent to one of Haskell's fixed-size integers (**`Int8`**, **`Int16`**, etc.). The problem with big Atom types, however, is that the enumeration functions using Peano-based finite naturals as indices (Fin *n*) can become quite inefficient for very large *n*.

For this reason it is recommended to choose a *simple and small* type to use as Atom, and represent the mapping between complex types and a sequence of Atoms using the ⇓W⇑ (`Synthesizable`) type class, detailed in Section 3.2.4.

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 ($\mathbb{C}'$) 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 11.

```
W : ℕ → Set
W = Vec Atom

record Gates : Set where
  field
    |Gates|    : ℕ
    |in| |out| : Fin |Gates| → ℕ
    spec       : (g : Fin |Gates|) → (W (|in| g) → W (|out| g))

  Gates# = Fin |Gates|
```

Listing 11: Definition of a gate library: the Gates record.

The first field of the Gates record (|Gates|) 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*.

Instance for `BoolTrio`

One specific instance of gate library is already defined in the Π-Ware library (in module PiWare.Gates.BoolTrio): the usual set of boolean gates {`NOT`, `AND`, `OR`}, together with the constants {`FALSE`, `TRUE`}. The first definitions in that module establish that the library contains five gates:

```
|BoolTrio| = 5
```

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

```
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 in the library. All gates have exactly one output, while the number of inputs vary from 0 to 2.

```
|in| |out| : Fin |BoolTrio| → ℕ
|in| = λ { FalseConst# → 0; TrueConst# → 0; Not# → 1; And# → 2; Or# → 2; (Absurd# ()) }
|out| _ = 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 (|in| g) → W (|out| g))
spec FalseConst#  = const [ false ]
spec TrueConst#   = const [ true  ]
spec Not#         = spec-not
spec And#         = spec-and
spec Or#          = spec-or
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|  = |BoolTrio|
  ; |in|    = |in|
  ; |out|   = |out|
  ; spec    = spec
  }
```

Assumed correctness

Later, in Section 3.4.1, 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 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 |Gates|) → 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.

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 12.

```
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 12: High-level circuit datatype ($\mathbb{C}$).

In contrast to the low-level ($\mathbb{C}'$) circuit type, $\mathbb{C}$ is parameterized by two Sets and has only one constructor – Mk$\mathbb{C}$ – which packs the low-level circuit description with information on how to convert between the input/output Agda types (resp. α and β) and the appropriately-sized *words*[3] used in the low level. This conversion information is coded in the two first parameters passed to the Mk$\mathbb{C}$ constructor, which are *instances* of the ⇓W⇑ type class (pronounced `Synthesizable`), This type class is covered in more detail on Section 3.2.4. Roughly, it ensures that a type can be converted to *words* (vectors of Atoms).

[3] A *word* is a vector of Atoms.

Furthermore, 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.

The PiWare.Circuit module defines also *smart constructors* for ℂ, corresponding to each of the constructors at the low level. One example is parallel composition – shown here without passing instance arguments for brevity:

```
_||_ : ∀ {α γ β δ} → ℂ α γ → ℂ β δ → ℂ (α × β) (γ × δ)
Mkℂ c₁ || Mkℂ c₂ = Mkℂ (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 13), built using the `BoolTrio` gate library mentioned before. The block diagram in Figure 3.3 illustrates the architecture of such circuit.

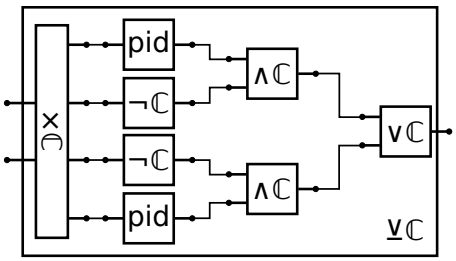

Figure 3.3: Architecture of a sample `XOR` gate.

```
⊥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 13: 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 size), 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 14 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 14: 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. Intuitively, any finite type is isomorphic to Agda's Fin n, for some n, for which an instance of ⇓W⇑ can easily be defined.

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 15.

In the down definition, the *bind* (>>=) operator is defined as concat ∘ map, i.e., first each element of the vector is serialized (using the passed instance for α), then all the words (each with size i) are concatenated, giving a total size of $(n * i)$.

To convert a serialized Vec back up, first the given word is grouped into n smaller words (each of size i), which is possible because the size of the passed word is $(n * i)$. Then, ignoring the proof object returned by group (which is in the second position of a pair, therefore we take proj_1), we map the up instance of α over each of the smaller words, obtaining in this way an element of type Vec α n.

```
⇓W⇑−Vec : ∀ {α i n} → ⦃ sα : ⇓W⇑ α {i} ⦄ → ⇓W⇑ (Vec α n)
⇓W⇑−Vec {α} {i} {n} ⦃ sα ⦄ = ⇓W⇑[ down , up ]
  where group′ : {α : Set} (n k : ℕ) → Vec α (n * k) → Vec (Vec α k) n
        group′ n k = proj₁ ∘ group n k

        down : Vec α n → W (n * i)
        down = concat ∘ map_v ⇓

        up : W (n * i) → Vec α n
        up = map_v ⇑ ∘ group′ n i
```

Listing 15: Predefined instance of ⇓W⇑ for fixed-length vectors.

Proof of bijection

It is very desirable that the pair of methods in ⇓W⇑ – ⇓ (`down`) and ⇑ (`up`) – are defined as inverses of each other, that is:

$$\forall\ (x : \alpha) \to \Uparrow\ (\Downarrow\ x) \equiv x$$
$$\forall\ (w : \mathsf{W}\ i) \to \Downarrow\ (\Uparrow\ w) \equiv w$$

With this property, the correctness of a low-level circuit ($\mathbb{C}'$) translates trivially via ⇓ and ⇑ to the correctness of it's high-level correspondent.

We considered including the proof that ⇓ and ⇑ also as methods of the ⇓W⇑ class, thereby *requiring* the user to prove this property before using any high-level type with Π-Ware. Ultimately, given the experience we had in writing these proofs for some of the ⇓W⇑ instances predefined in Π-Ware, we decided against inclusion as required methods. This would be too big of an initial burden put on a designer, who would have to prove bijection *even before* being able to write any prototype circuit using the intended types.

`Synthesizable` instance for sums

The instance of ⇓W⇑ for sums (disjoint unions) is the most complex among all predefined instances included with Π-Ware. This complexity comes mostly from the need to *distinguish* which set is concerned when converting a word "up" (⇑). Assuming that we want ⇓ and ⇑ to form a bijection (no information may be lost), let's first calculate how many atoms are necessary to represent a sum.

With a given encoding for set α ($s\alpha$: ⇓W⇑ α {i}) and one for β ($s\beta$: ⇓W⇑ β {j}), it is clear that at least $i \sqcup j$ atoms are necessary to encode $\alpha \uplus \beta$; the size of a sum is at least the maximum among the sizes of its summands. Furthermore, it is necessary to somehow encode the choice (whether the sum was built from the left (inj_1) or the right (inj_2). This requires at least one more atom, bringing the size to suc ($i \sqcup j$).

The predefined instance for sums included with Π-Ware is shown on Listing 16.

Besides the encodings of the summands (named $s\alpha$ and $s\beta$, given as *instance arguments*), this instance is also passed some extra parameters: three *atom indices* (n, m and p) and a proof that n and m are different. The atom index n indicates the Atom to be used in case the sum is built from the left (inj_1). and m for the case when it is built from the right (inj_2). They need to be different in order for the choice information not to be

```
⇓W⇑−⊎ : ∀ {α i β j} → (n m p : Atom#) {d : n ≢ m}
  → ⦇ sα : ⇓W⇑ α {i} ⦈ ⦇ sβ : ⇓W⇑ β {j} ⦈ → ⇓W⇑ (α ⊎ β) {suc (i ⊔ j)}
⇓W⇑−⊎ {α} {i} {β} {j} n m p ⦇ sα ⦈ ⦇ sβ ⦈ = ⇓W⇑[ down , up ]
  where down : α ⊎ β → W (suc (i ⊔ j))
        down = [ (λ a → (n→atom n) :: padFst i j (n→atom p) (⇓ a))
               , (λ b → (n→atom m) :: padSnd i j (n→atom p) (⇓ b)) ]

        up : W (suc (i ⊔ j)) → α ⊎ β
        up = map⊎ ⇑ ⇑ ∘ untag
```

Listing 16: Predefined instance of ⇓W⇑ for sums.

lost when synthesizing a sum type. At last, the p parameter identifies the atom used for padding.

In the down function, the [_, _] sum *destructor* is used, and is passed two functions: one to operate at the left case and one at the right. They prepend to the output word an atom identified by either n or m and pad either a or b to fit $i \sqcup j$.

In the up conversion, first the *tag* is used to transform the input word into a sum of words (W i ⊎ W j), then the map⊎ function is used to apply either the ⇑ instance for α or for β, giving the desired result type of $\alpha \uplus \beta$.

Recursive instance search

To model synthesizable datatypes, we used the Agda way of implementing Haskell's type classes, namely, to code a type class as a record type, in which each field corresponds to a method. In this analogy, instances are just elements of the record type, and class constraints are coded in Agda by passing *instance arguments* [13] to the constrained function. Arguments passed in this way will be searched among the identifiers available in scope, in a type-directed fashion, similar to how the mechanism of instance search works in Haskell.

However, there was (until recently) a big drawback to Agda's instance search implementation: it was not recursive. There were some arguments for this decision in the original paper, most notably that having fully recursive instance resolution would "expose an additional computational model" and that the instance search mechanism would need to "perform a reasonably powerful automated proof search".

During the last years, since the publication of the paper on instance arguments [13], demand has grown for recursive instance search in Agda, and in a recent contribution to the Agda codebase by Guillaume Brunerie[4], costs and benefits were again weighted, and a reasonably efficient implementation was released in Agda version 2.4.0.2 (current version at the time of writing this report).

The impact of this change for Π-Ware is immense. Recursive instance resolution removes the need for a great deal of boilerplate code. For example, before the advent of recursive resolution, modelling the XOR circuit on Listing 13 required the user to manually construct instances of ⇓W⇑ for more than 5 types ($\mathbb{B} \times \mathbb{B}$, $(\mathbb{B} \times \mathbb{B}) \times \mathbb{B}$, $\mathbb{B} \times (\mathbb{B} \times \mathbb{B})$, $(\mathbb{B} \times (\mathbb{B} \times \mathbb{B})) \times (\mathbb{B} \times \mathbb{B})$, etc.). Now all of this work is done by Agda itself. On the other hand, the typechecking time grew a little, and this could be a problem when modelling big circuits, as discussed in Section 5.1.1.

[4] https://github.com/agda/agda/commit/7ca60fd60aec1b73d9f14aaa6e1aeef184bcca79

3.3 Circuit Semantics

In this chapter we will expose and discuss the *semantics* of circuits described using Π-Ware. Π-Ware circuits can currently be interpreted in two ways: they can be *simulated* (fed with certain inputs and calculate the corresponding outputs) or *synthesized* (converted to a *netlist*). However, due to the fact the a *deep embedding* is used, other semantics can implemented with relative ease, such as timing analysis, power estimation, etc.

When presenting the simulation semantics, we will separately discuss the simulation of purely combinational circuits and of sequential ones, also recapitulating the need to distinguish between the two.

3.3.1 Synthesis semantics

The synthesis semantics establishes how Π-Ware circuits can be converted to netlists. Each element of the low-level circuit datatype ($\mathbb{C}'$) has a corresponding netlist according to the semantics, and to each circuit constructor corresponds a netlist transformation.

Figure 3.4 shows the semantic rules for the fundamental constructors of $\mathbb{C}'$, as well as for DelayLoop. To the left of each diagram there is a typing rule for that case, with the constructor's arguments (and their types) above the bar and the result type below it.

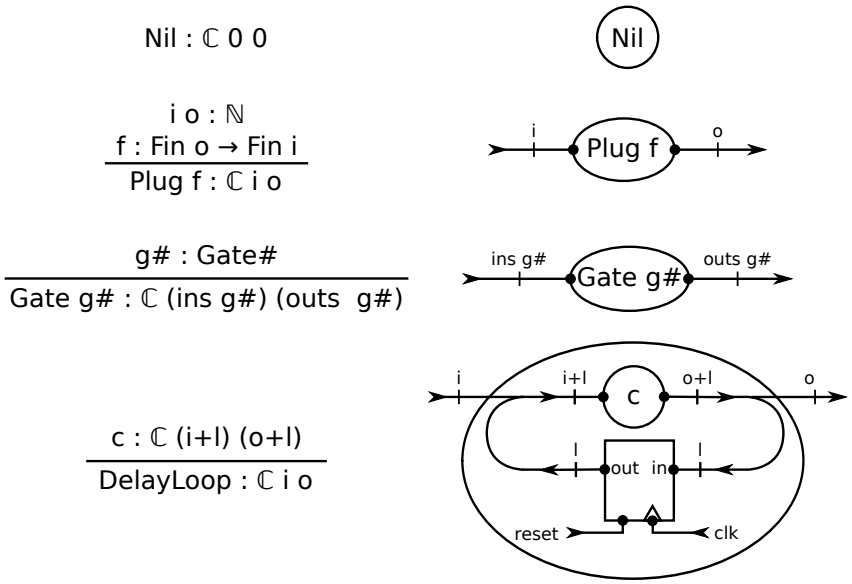

Figure 3.4: Synthesis semantics of the fundamental circuit constructors.

In the diagrams, all ellipses denote elements of type $\mathbb{C}'$, i.e., low-level circuits. Assuming the existence of a hierarchy of datatypes for representing VHDL code (as already mentioned in Section 3.2.2), each of these ellipses correspond to a VHDL *entity* (an element of VHDLEntity).

The line segments connecting the ellipses are signals (VHDLSignal) – with their size denoted by the variable above the perpendicular "dash" in the segment. Each of the little dots on the extremity of a line segment correspond to a *port declaration*

(VHDLPortDecl). The arrows on the line segments indicate the direction of the flow of information.

It is important that these diagrams *do not prescribe any placement* for the circuit components. They denote only *netlists*, i.e., directed graphs in which the nodes are elements of $\mathbb{C}'$ and the edges are the signals connecting them (labeled by signal *size*). The actual *placement* of this netlist into physical slots (of an Field-Programmable Gate Array (FPGA) or ASIC), with the accompanying *routing* of wires, is a further step in the implementation chain, *not performed by Π-Ware*.

The semantics of structural circuit constructors (sequential, parallel, sum) is shown in Figure 3.5.

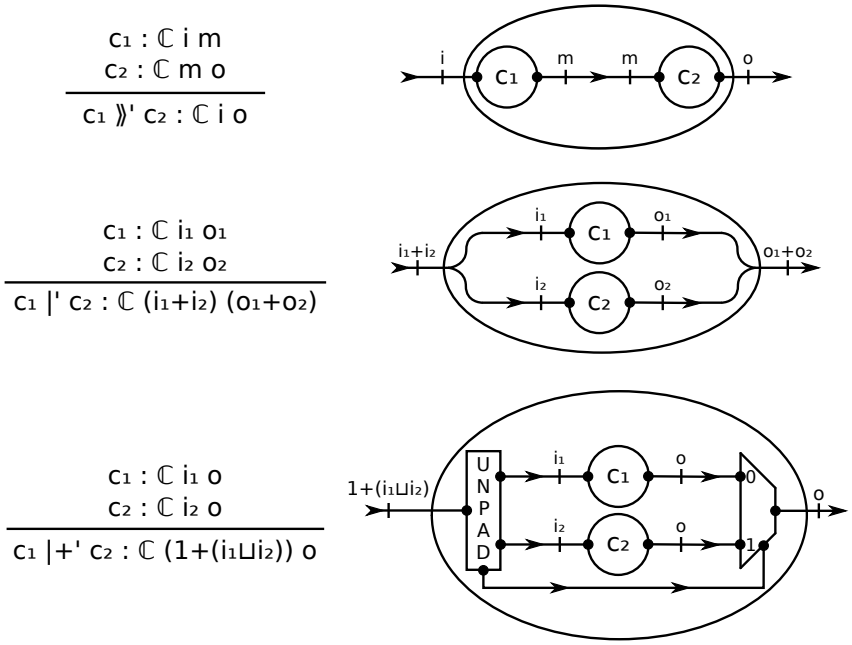

Figure 3.5: Synthesis semantics of structural circuit constructors.

In the sum case (_|+'_), the UNPAD block only "slices" the input into three outputs: the tag (first bit) is used as the selector for the *multiplexer*, and the other two outputs have, respectively, the first i_1 and i_2 bits of the input. The diagram on Figure 3.6 shows which slices of input go to each output of the UNPAD block in the sum case.

Finally, it is important to emphasize that the labels `'0'` and `'1'` were used in the multiplexer only as a mnemonic device. The same remark applies to describing the information on the wires as "bits". In reality, any Atomic type can flow through the wires, and any two Atoms can label the inputs of the multiplexer (passed to _|+'_).

3.3.2 Combinational simulation

As already mentioned in Section 3.1, the comb′ predicate determines whether its circuit argument is *purely combinational*. In fact, comb′ can be seen as one (very simple) semantics of circuits ($\mathbb{C}'$). To recapitulate, here is the type of comb′:

$$\mathsf{comb}' : \{i\ o : \mathbb{N}\} \to \mathbb{C}'\ i\ o \to \mathsf{Set}$$

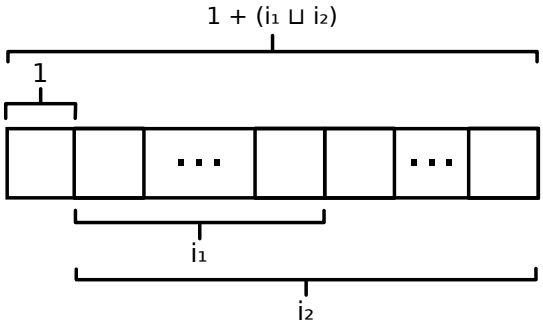

Figure 3.6: How UNPAD "slices" its input into three outputs.

This predicate is used in several places in the Π-Ware library, particularly in what concerns the *simulation* semantics. Purely combinational circuits have no notion of time or state, so their simulation semantics is modelled as just a function over *words* (Atom vectors).

The Agda function giving a time-independent (combinational) semantics for circuits requires that *an element of type* comb′ *c* be passed as implicit argument – corresponding to a proof that circuit *c* is purely combinational. Furthermore, this semantics depends on the *gate library* and the Atom type being used, which are parameters to the PiWare.Simulation.Core module as a whole. Listing 17 shows the Agda code for the combinational simulation semantics:

```
⟦_⟧′ : {i o : ℕ} → (c : ℂ′ i o) {p : comb′ c} → (W i → W o)
⟦ Nil       ⟧′ = const ε
⟦ Gate g#   ⟧′ = spec g#
⟦ Plug p    ⟧′ = plugOutputs p

⟦ DelayLoop c ⟧′ {()} v

⟦ c₁ ⟫′ c₂ ⟧′ {p₁ , p₂} = ⟦ c₂ ⟧′ {p₂} ∘ ⟦ c₁ ⟧′ {p₁}

⟦ _|′_ {i₁} c₁ c₂ ⟧′ {p₁ , p₂} =
  uncurry′ _++_ ∘ map (⟦ c₁ ⟧′ {p₁}) (⟦ c₂ ⟧′ {p₂}) ∘ splitAt′ i₁

⟦ _|+′_ {i₁} c₁ c₂ ⟧′ {p₁ , p₂} =
  [ ⟦ c₁ ⟧′ {p₁} , ⟦ c₂ ⟧′ {p₂} ]′ ∘ untag {i₁}
```

Listing 17: Agda code for the combinational simulation semantics of ℂ′.

First of all, notice the prime symbol on the function's name (⟦_⟧′), indicating that it concerns *low-level* circuits (ℂ′). From the cases for fundamental constructors, we highlight the usage of the spec function from the chosen gate library, returning the specification function for a given gate. Furthermore, the DelayLoop case can be *discharged*, i.e., it is not necessary to provide a definition in this case, as comb′ (DelayLoop *c*) evaluates to ⊥ and – by definition – no elements of this type can be constructed.

The cases for structural constructors are somewhat straightforward. Sequencing (_⟫′_) is interpreted just as function composition. In the parallel composition (_|′_)

case, first the input vector is *splitted*, creating a pair. Then the map function over pairs is used, to apply (respectively) the semantics of c_1 and c_2 over each element of the pair. Finally, both transformed pair elements are concatenated to build the output. In the sum ($\_|+'\_$) case, the untag function uses the first atom of the input to decode it into a sum. Then the sum destructor ($[\_,\_]'$) is used with the appropriate functions (semantics of c_1 and c_2) to build the output accordingly.

Notice how the (implicit) arguments of type comb′ c are matched against and passed. In the cases for structural constructors, we know that – by definition – the *whole circuit is combinational if and only if all its subcircuits are combinational*. Therefore, we can pattern match with the *pair constructor* (p_1 , p_2), and pass each of p_1 and p_2 to the calls of $[\![\_]\!]'$ on the subcircuits.

High-level combinational simulation

Analogous to the dichotomy between low-level *circuits* ($\mathbb{C}'$) and high-level ones ($\mathbb{C}$), there are also both low and high-level simulation semantics[5].

While the low-level simulation semantics ($[\![\_]\!]'$) maps a low-level circuit ($\mathbb{C}'$) to a function over *words*, the high-level simulation semantics ($[\![\_]\!]$ – shown in Listing 18) maps a high-level circuit ($\mathbb{C}$) to a function over arbitrary (synthesizable) Agda types.

```
[\![_]\!] : ∀ {α i β j} → (c : ℂ α β {i} {j}) {p : comb c} → (α → β)
[\![ Mkℂ {{ sα }} {{ sβ }} c′ ]\!] {p} = ⇑ ∘ [\![ c′ ]\!]′ {p} ∘ ⇓
```

Listing 18: Simulation semantics for high-level circuits ($\mathbb{C}$).

3.3.3 Sequential simulation

So-called *sequential* circuits have an internal state, and their behaviour is *history-dependent*: The current value of a circuit's output depends not only on the current input, but also on the history of previous inputs. Therefore, to simulate these circuits, there is the need of some concept of *time dimension* where the values of inputs and outputs are situated.

Π-Ware supports modelling and simulating a specific class of sequential circuits, namely *synchronous* sequential circuits. In synchronous sequential circuits, state transitions in all memory elements happen at the same time, and they can only happen at the (rising or falling) *edge* of a global clock circuit. Because of this *discrete* nature of signals in synchronous circuits, they are usually modelled as *streams* in hardware DSLs. This is also the way in which sequential simulation works in Π-Ware: it converts a circuit into a function over streams.

Listing 19 shows the type of the semantic function for sequential simulation of low-level ($\mathbb{C}'$) circuits:

```
[\![_]\!]* ′ : {i o : ℕ} → ℂ′ i o → (Stream (W i) → Stream (W o))
```

Listing 19: Type signature of the semantic function for low-level sequential simulation.

This type signature differs from the purely combinational case (depicted in Listing 17) in only two important ways:

[5] There is no such "high-level" concept for the *synthesis* semantics, as it concerns actual hardware.

- There is no argument of type comb′ c anymore requiring the circuit to be purely combinational.
- The result of evaluating the circuit is now "lifted" to the Stream setting, that is, it becomes Stream (W i) → Stream (W o). instead of W i → W o.

The implementation of the history-dependent behaviour needed in sequential simulation uses so-called *causal streams* [30]. Even though the type Stream (W i) → Stream (W o) is convenient as an *interface* to interact with sequential circuits, it does not reflect accurately the way in which synchronous sequential circuits actually work.

The type of a general stream function (Stream α → Stream β) can be read as:

> Given an infinite sequence of values of type α, produce an infinite sequence of values of type β.

As such, a general stream function has no notion of *current* value, neither of past, and it *can* "look ahead", that is, use future input values to calculate the next output value. One example of such a future-anticipating behaviour can be found in the tail function located in the Data.Stream module:

```
tail : ∀ {α} → Stream α → Stream α
tail (x :: xs) = ♭ xs
```

This behaviour cannot be realized as a hardware circuit, as it would imply that the circuit outputs in clock cycle t the contents of its input in clock cycle $t + 1$.

Causal step semantics

Therefore, in Π-Ware, the *sequential semantics* for circuits ([[_]]∗′) uses a helper definition, a so-called *causal step* semantics. A *causal step function* is a function which, given a *causal context*, produces *one* next value of the output stream. Listing 20 shows the types used in Π-Ware to represent these concepts.

```
Γc : (α : Set) → Set
Γc = List⁺

_⇒c_ : (α β : Set) → Set
α ⇒c β = Γc α → β
```

Listing 20: Types used to model causal streams.

A *causal context* with values of type α is written as Γc α. It is implemented as a *type synonym* of Agda's non-empty lists. In this way, there is always a current value (head), while the past (tail) might be empty. In some places, when pattern matching non-empty lists, a comma (_,_) is used instead of the normal *cons* operator (_::_), to avoid confusion with general lists.

The *causal step semantics* ([[_]]c) maps a circuit into a *causal step function*, i.e., given a circuit, it returns the function which has current and past inputs as arguments and produces the immediately next output as result. The code for [[_]]c is shown in Listing 21.

```
⟦_⟧c : {i o : ℕ} → ℂ′ i o → (W i ⇒c W o)
⟦ Nil                    ⟧c (w⁰ , _) = ⟦ Nil ⟧′ w⁰
⟦ Gate g#                ⟧c (w⁰ , _) = ⟦ Gate g# ⟧′ w⁰
⟦ Plug p                 ⟧c (w⁰ , _) = plugOutputs p w⁰
⟦ DelayLoop {o = j} c {p} ⟧c         = take_v j ∘ delay {o = j} c {p}

⟦ c₁ ⟫′ c₂ ⟧c          = ⟦ c₂ ⟧c ∘ map⁺ ⟦ c₁ ⟧c ∘ tails⁺
⟦ _|′_ {i₁} c₁ c₂ ⟧c = uncurry′ _++_ ∘ map ⟦ c₁ ⟧c ⟦ c₂ ⟧c ∘ unzip⁺ ∘ splitAt⁺ i₁

⟦ _|+′_ {i₁} c₁ c₂ ⟧c (w⁰ , w⁻) with untag {i₁} w⁰ | untagList {i₁} w⁻
... | inj₁ w₁⁰ | w₁⁻ , _  = ⟦ c₁ ⟧c (w₁⁰ , w₁⁻)
... | inj₂ w₂⁰ | _  , w₂⁻ = ⟦ c₂ ⟧c (w₂⁰ , w₂⁻)
```

Listing 21: Causal step semantics for sequential circuits.

In the Nil, Gate and Plug cases, the purely combinational semantic function is called (ignoring past inputs), passing (implicitly) the necessary proof of purity. In the definition of ⟦_⟧c for _⟫′_, the context of c_1 is just the context of the sequence block as a whole, while the context of c_2 consists of the *outputs* of c_1. These outputs are calculated by *mapping* the causal step function (⟦_⟧c) over *all previous* contexts of c_1 (from most recent to oldest, given by tails⁺).

The parallel composition (_|′_) case has the expected definition: it splits the input context pointwise at offset i_1, then applies ⟦ c_1 ⟧c and ⟦ c_2 ⟧c correspondingly and concatenates the results. In the sum (_|+′_) case, the past part of the context is seggregated into a product with only *left summands* in the first projection and *right summands* in the second. The type of untagList helps to clarify its purpose:

```
untagList : ∀ {i j} → List (W (suc (i ⊔ j))) → List (W i) × List (W j)
```

These two seggregated pasts are then passed as the pasts of c_1 and c_2, respectively. The definition of _|+′_ is completed by pattern matching on the sum generated by untag $\{i_1\}$ w_0, which gives the current input (w_1^0 resp. w_2^0) to be passed in the recursive calls with the subcircuits (⟦ c_1 ⟧c resp. ⟦ c_2 ⟧c).

It is important to note that, had we used a naïve stream semantics for sequential circuits, it would not be possible to define untagList. Given a stream of sums, we *cannot* produce a product of streams in which each component has only *left summands* or *right summands*. Using a naïve stream semantics, the function to seggregate the input stream would need to have the following type:

```
seggregateStream : ∀ {α β} → Stream (α ⊎ β) → Stream α × Stream β
```

If, for example, the input stream contains no left summand whatsoever, then no element is available to build the first component of the pair – a *stream of lefts* – therefore there is no total definition for seggregateSums. On the other hand, if we substitute an infinite input stream for a causal context then the "seggregation" becomes easy: in case the input context contains no left summand, the first element of the product is simply an empty context.

DelayLoop is the most important case of ⟦_⟧c, and a bit more involved. It calls a helper function (delay) expressing the behaviour of the memory element *inside* the DelayLoop block (as depicted in Figure 3.4). Listing 22 shows the definition of delay.

```
delay : (c : ℂ′ (i + l) (o + l)) {p : comb′ c} → W i ⇒c W (o + l)
delay c {p} = uncurry′ (delay′ c {p})
  where
    delay′ : ∀ (c : ℂ′ (i + l) (o + l)) {p : comb′ c} → W i → List (W i) → W (o + l)
    delay′ c {p} w⁰ []          = ⟦ c ⟧′ {p} (w⁰ ++ replicate (n→atom Fz))
    delay′ c {p} w⁰ (w⁻¹ :: w⁻) = ⟦ c ⟧′ {p} (w⁰ ++ drop o (delay′ c {p} w⁻¹ w⁻))
```

Listing 22: Function expressing the behaviour of a circuit memory element.

It consists of just "uncurrying" another local definition (delay′) which actually performs the work. This indirect definition is necessary to pass Agda's *termination checker*, which requires recursive calls to be performed on at least one *structurally smaller* argument – in this case, the list of past inputs ($w^{-1} :: w^-$ becomes w^{-1}).

The c parameter to delay is the *combinational* subcircuit inside the DelayLoop block, thus its type $(c : (i + l) (o + l))$ includes the amount of wires (l) "looping back" through the memory element. In the base case (empty past), delay′ simulates ⟦ c ⟧′ with an *undefined* state – consisting of Atoms indexed by zero (Fz). In the recursive case, the state is calculate by a recursive call, where the "current" input has been substituted by the first past value (w^{-1}) and the "past" by the even farther past (w^-).

The current implementation of delay is quite inefficient, as the recursive call calculates the same values several times; a problem similar to what happens in a naïve recursive implementation of the Fibonacci sequence. As in the Fibonacci case, *memoization* techniques could also be used to improve the performance of delay.

From step function to stream function

The *causal step function* (⟦_⟧′) produces only the next value of the output stream, but the desired interface for simulation is one which, given a stream of inputs, produces a stream of outputs, i.e, with type Stream (W i) → Stream (W o).

To "bridge" this gap, the runc function is used (Listing 23). It was translated into Agda from the paper "The essence of dataflow programming" [30].

```
runc : ∀ {α β} → (α ⇒c β) → (Stream α → Stream β)
runc f (x⁰ :: x⁺) = runc′ f ((x⁰ , []) , ♭ x⁺)
  where runc′ : ∀ {α β} → (α ⇒c β) → (Γc α × Stream α) → Stream β
        runc′ f ((x⁰ , x⁻) , (x¹ :: x⁺)) =
          f (x⁰ , x⁻) :: ♯ runc′ f ((x¹ , x⁰ :: x⁻) , ♭ x⁺)
```

Listing 23: Producing a stream function from a causal step function.

The implementation is based on a helper function (runc′), implemented using Agda's *coinduction* features. These features are located in the Coinduction module of the standard library.

A good introduction to coalgebras, coinduction and corecursion can be found in Bart Jacobs's "Introduction to coalgebra" [19]. In essence, algorithms written using coinduction are a way to deal with (potentially) infinite data structures in total languages.

In these languages, recursive/inductive functions need to *terminate*, and in Agda the *termination checker* uses a syntactic heuristic to ensure termination: recursive calls

must be performed with *structurally smaller* arguments. On the other hand, *corecursive/coinductive* functions must be *productive*: they must produce the "next" piece of information in finite time. Agda's heuristics to ensure productivity is to look out for *guardedness*. Research about coinduction and its implementation in total languages is thriving, and the reader is directed to [4] for more information.

In the case of Π-Ware, it suffices to say that the corecursive call of runc′ is indeed guarded (therefore productive), and that using runc we can finally define a *sequential simulation semantics* (Listing 24) which both has a convenient interface (stream-based) and models synchronous hardware faithfully (is causal).

```
⟦_⟧*′ : {i o : ℕ} → ℂ′ i o → (Stream (W i) → Stream (W o))
⟦_⟧*′ = runc ∘ ⟦_⟧c
```

Listing 24: The sequential simulation semantics.

3.4 Proving circuit properties

The main reasons for embedding Π-Ware in a dependently-typed programming language are twofold:

- Use dependent types to *prevent design mistakes*.
- Use dependent types as a logic system to *prove properties* about circuits.

While in sections 3.1 and 3.3 we focused on how Π-Ware makes use of dependent types to prevent some classes of mistakes early in the design process, in this section we show what kind of *proofs involving circuits* are possible in Π-Ware and how they work.

Different classes of statements about circuits arise depending on what kind of *semantics* is being used.

- With a *simulation* semantics we can state and prove properties about *functional behaviour*, i.e, the relation between a circuit's inputs and outputs. For example, we can state that a circuit's output values always remain within some range, or that two circuits *have equal behaviour*, i.e., always produce the same outputs given the same inputs.
- With a *synthesis* semantics, on the other, properties about circuit *structure* can be stated and proven. For example, we can state that a circuit never exceeds a certain number of fundamental gates, or that the *longest chain* of gates is never longer than a predefined length.
- Furthermore, several other sorts of static analyses can be carried with the introduction of new semantics, such as timing analysis, power consumption analysis, etc.

In this section we focus mainly on statements and proofs involving circuit *functional behaviour* (those involving the *simulation semantics*), as the examples included in the Π-Ware library belong to this class.

3.4.1 Functional correctness

In hardware design, there are two closely related activities which aim to ensure the quality of a circuit: *verification* and *validation*. *Verification* of a circuit encompasses all sorts of tests and procedures to ensure that a circuit *meets a given specification*. *Validation*, on the other hand, is concerned with ensuring that the circuit *and its specification* effectively meet the needs of involved stakeholders. These definitions (originating from the PMBOK® guide) were made into an IEEE standard [3] in 2011.

In the hardware design community, two tongue-in-cheek expressions are used as a mnemonic device for this distinction between validation and verification:

- When performing **verification**, the question asked is "did we build the *thing right*"
- When performing **validation**, the question asked is "did we build the *right thing*"

In the software *formal methods* community, the term *correctness* is used to denote the goal of verification. A *correct* algorithm, in this sense, is an algorithm that meets its specification. We use the same *correctness* terminology applied to hardware, therefore we are interested in showing the *correctness of circuits*.

More specifically, we focus on establishing correctness by means of *formal verification*. This means that *both* specification and implementation are described formally, and we seek a formal proof that the implementation meets the specification. *Functional* correctness refers to the input/output behaviour of the circuit, i.e., that for each input the *correct* output is produced (as prescribed by the specification).

In Π-Ware's case, the circuit model itself (element of $\mathbb{C}'$) is considered the *implementation*, while the *specification* of a circuit is given by an appropriately-typed Agda function. Let's consider the simple example of an `XOR` circuit, already shown in Listing 13 but repeated here for convenience:

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

We can write the *specification function* in basically two ways: as a truth table (listing all possible input combinations) or using the boolean functions from Agda's standard library. The truth table specification is shown in Listing 25

```
⩒ℂ−spec−table : (B × B) → B
⩒ℂ−spec−table (false , false) = false
⩒ℂ−spec−table (false , true ) = true
⩒ℂ−spec−table (true  , false) = true
⩒ℂ−spec−table (true  , true ) = false
```

Listing 25: Specification function for ⩒ℂ as a truth table.

We can also use the `xor` function from Agda's standard library (module `Data.Bool`) as specification. The type signature of `xor` is as follows:

```
_xor_ : Bool → Bool → Bool
```

```
⩒ℂ−spec−subfunc : (B × B) → B
⩒ℂ−spec−subfunc = uncurry′ _xor_
```

Listing 26: Specification function for ⩒ℂ based in Agda's _xor_.

If we want to make the _xor_ based specification have the same type as ⩒ℂ−spec−table, we can use the uncurry′ function from Data.Product. Then, we arrive at the ⩒ℂ−spec−subfunc definition, shown in Listing 26.

The proof of functional correctness for ⩒ℂ can be written in different ways, depending on which specification function we choose. If we use the truth table specification, then the proof can also only proceed by *case analysis*, as shown in Listing 27.

```
⩒ℂ−proof−table : ∀ a b → ⟦ ⩒ℂ ⟧ (a , b) ≡ ⩒ℂ−spec−table (a , b)
⩒ℂ−proof−table false false = refl
⩒ℂ−proof−table false true  = refl
⩒ℂ−proof−table true  false = refl
⩒ℂ−proof−table true  true  = refl
```

Listing 27: Functional correctness proof of ⩒ℂ, with ⩒ℂ−spec−table as specification.

In this kind of proof (by case analysis), the lack of a tactic language in Agda is a disadvantage: even though **all** cases have the same right-hand side (refl), they still have to be listed explicitly. We expect, however, that by using *proof by reflection* [31] such proofs can be automated. Further investigation is left as future work.

The correctness proof for ⩒ℂ using ⩒ℂ−spec−subfunc is shown in Listing 28.

```
⩒ℂ−proof−subfunc : ∀ a b → ⟦ ⩒ℂ ⟧ (a , b) ≡ ⩒ℂ−spec−subfunc (a , b)
⩒ℂ−proof−subfunc = ⩒ℂ−xor−equiv
```

Listing 28: Functional correctness proof of ⩒ℂ, specified by ⩒ℂ−spec−subfunc.

An important remark is that this "higher-level" proof (⩒ℂ−proof−subfunc) could *also* be completed by simple case analysis, and therefore *proof by reflection* could be used as a general tactic for all proofs involving circuits definitions with *fixed size*. But in the case of ⩒ℂ−proof−subfunc we chose to complete the proof in a different fashion, for illustrative purposes.

The circuit evaluation (⟦⩒ℂ⟧) reduces to an expression involving the functions _∧_ (AND), _∨_ (OR) and not, while the specification (Agda's _xor_) is ultimately defined by pattern matching. Therefore we use a lemma (⩒ℂ−xor−equiv) which shows the equivalence of Agda's _xor_ and a function involving _∧_, _∨_ and not. This lemma has the following type:

```
⩒ℂ−xor−equiv : ∀ a b → (not a ∧ b) ∨ (a ∧ not b) ≡ (a xor b)
```

Proofs can also be written regarding whole *families of circuits*. For example, we can have a generically-sized definition for an AND circuit, as follows (low-level):

```
andN′ : ∀ n → ℂ′ n 1
andN′ zero    = Gate TrueConst#
andN′ (suc n) = pid′ {1} |′ andN′ n ⟩⟩′ Gate And#
```

One of the properties that this definition must satisfy is, for example, that when all inputs are true, the output must be true, regardless of the number of inputs. This corresponds to the following Agda definition:

```
proof−andN−core−alltrue : ∀ n → ⟦ andN′ n ⟧′ {andN′−comb n} (replicate true) ≡ [ true ]
proof−andN−core−alltrue zero  = refl
proof−andN−core−alltrue (suc n) = cong (spec−and ∘ (_::_ true)) (proof−andN−core−alltrue n)
```

The property is proved for *any* n, therefore fundamentally differently than the approaches using automated verification (calling SAT solvers), such as Lava. The proof proceeds by induction on the *number of inputs* of the circuit (n) and the inductive case (suc n) has a generally applicable structure: it combines the inductive hypothesis with the correctness of the underlying circuit used in the recursive definition of andN′.

In this case, the underlying circuit is simply an AND gate (Gate And#). Therefore its *specification function* is the one defined in the *gate library*, named spec−and.

3.4.2 Sequential correctness

Properties about the *sequential* behaviour of circuits are more complex to state, and especially to prove. Let's start by describing a simple 1-bit register in Π-Ware. This register has two inputs (of 1 bit each) and one output (also of 1 bit), the block diagram of this sample register is shown in Figure 3.7.

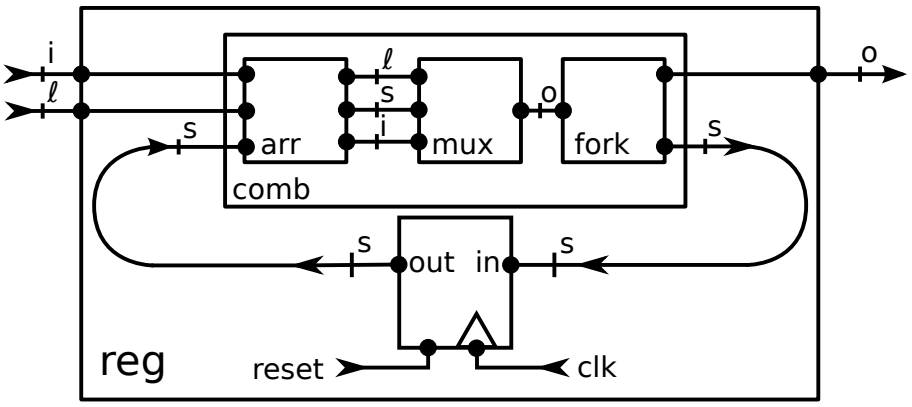

Figure 3.7: Block diagram for the register described in Listing 29.

The first input is the bit effectively being read and stored; this input is named "i". The second input is a control bit, telling whether or not the first input should be "loaded" into the register state; its name is therefore "ℓ" (for *load*). The register's output pin simply exposes the register's internal state at any point in time.

Also, this circuit has both an *asynchronous reset* and *clock* inputs, which are present in the synthesized netlist but *not* in the Π-Ware model. This, in fact, applies to all sequential circuits: they all result in netlist with two extra inputs (*reset* and *clock*) which cannot be observed through an Agda interface.

The Π-Ware code for this circuit (reg) is shown in Listing 29.

```
reg : ℂ (B × B) B
reg = delayℂ comb
  where rearrange = pSwap || pid ⟫ pALR ⟫ pid || pSwap
        comb      = rearrange ⟫ mux2to1 ⟫ pFork×
```

Listing 29: Sample register circuit description.

The way in which reg is defined matches precisely the block diagram: delayℂ introduces the loopback and the flip-flop "around" the combinational subcircuit (comb). This combinational subcircuit consist mainly of a *multiplexer*, "choosing" whether to use the current state or current input as next state. Lastly, this multiplexer is wrapped by some Plugs that serve only to change the ordering of some wires, performing no computation at all.

The behaviour of this circuit is such that, with the "load" ping high, it will output its current input *and* set the state of the flip-flop. With the "load" ping low, the output in clock cycle $n + 1$ is equal to the input value in clock cycle n. One example of this behaviour is shown below:

```
loads inputs : Stream Bool
loads  = true :: ♯ (true  :: ♯ (false :: ♯ repeat false))
inputs = true :: ♯ (false :: ♯ (true  :: ♯ repeat false))

load2 = take 7 ([[ reg ]]* $ zipWith _,_ inputs loads) ≡ true ◁ replicate false
```

In this example, the "load" pin is high for two clock cycles, and afterwards remains forever low. We are "sampling" seven elements from the register's (infinite) output Stream and comparing them to an *expected* vector on the right-hand side of the equality (_≡_). We also renamed the vector prepending operator (*cons*) to _◁_, in order to avoid confusion with the Stream prepending operator (_::_). Notice how the register output is "imitating" the input, because during the first two clock cycles the "load" pin is kept high. After that, a change in input will not cause a change in output.

Naturally, we would like to prove these sorts of properties about sequential circuits for *infinite* input and output streams. We would like to proof, for example, that if the "load" pin is lowered and kept low *forever*, then the output value will *never* change anymore. However, proofs involving coinductive types in general (and Streams in specific) are harder than regular proofs in Agda.

The first difficulty is related to a notion of *equality* between Streams. Usually, equality proofs in Agda use *propositional equality*, defined as a datatype (_≡_) in Agda's standard library (Relation.Binary.PropositionalEquality). It boils down to the fact that two terms are considered equal *if and only if* they have identical (syntactically speaking) *normal forms*.

But then, terms constructed by coinduction do not *normalize* as usual. Terms which intuitively we would expect to be equal have *suspension points* and will not compare equal according to *propositional equality*. The proper notion of "sameness" for infinite Streams is that of *bisimilarity*: two Streams are said to be *bisimilar if and only if*:

- Their *head* is *equal − propositionally equal*
- Their *tail* is *bisimilar*

Thus, the proof of bisimilarity can also be seen as a *stream of equality proofs*, one for each position in the Streams being compared. The definition of _≈_ is contained in Agda's standard library (Data.Stream), and reproduced here for convenience:

```
data _≈_ {α : Set} : Stream α → Stream α → Set where
  _::_ : ∀ {x y xs ys}
           (head≡ : x ≡ y) (tail≈ : ∞ (♭ xs ≈ ♭ ys)) → (x :: xs) ≈ (y :: ys)
```

Due to these difficulties with bisimilarity we do not currently have a proof of correctness for the sequential behaviour of reg. We also believe that this difficulty is somewhat related to the way in which the *causal step semantics* is converted into a *stream semantics* by the runc function, described in Listing 23.

Unfortunately, these investigations will have to remain for future work. However, we expect to be able to attack this issue by developing more general *proof combinators* for both combinational and sequential circuits.

Chapter 4

Discussion and related work

The goals of Π-Ware and the techniques used in the library were very much inspired by the EDSLs already mentioned in Sections 2.2.1 and 2.4, some of which were more deeply studied in a project predating this thesis.

In this chapter we provide an overview of Π-Ware's features by means of example circuits and drawing comparisons with related work.

4.1 Functional hardware EDSLs

µFP

Π-Ware is a *deeply*-embedded DSL, and was made so in order to allow for multiple interpretations or *semantics* of circuits. The mapping between Π-Ware circuits and different semantics is similar to the behaviour of µFP [28]. Each constructor of Π-Ware's low-level circuit datatype ($\mathbb{C}'$) corresponds to two semantics, as in µFP: the *simulation* semantics and the *synthesis* semantics. Even though µFP is not embedded, and therefore does not have "circuit constructors" as (deep) EDSLs would have, we can still relate µFP's composition to Π-Ware's sequencing, µFP's tupling to Π-Ware parallel composition and so forth, keeping the analogy.

There is a fundamental difference in what regards the *synthesis* semantics of Π-Ware and µFP: the synthesis semantics of µFP prescribes a *floorplan*, while Π-Ware synthesis only specifies a *netlist*. A *netlist* is a directed graph with logic gates as nodes and signals (wires) as edges. Producing a *floorplan* goes one (slight) step further, by specifying relative physical positions between the components, such as "besides", "adjacent to", etc. In Π-Ware's case, it is the job of *other* tools in the implementation chain to take Π-Ware's generated VHDL as input and perform placement and routing.

A last idea from µFP which inspired Π-Ware greatly was the definition of a single operator capable of introducing *state* into circuits. It is also the only way to introduce loops in circuits. This construct is called the "µ combining form" in µFP, and actually gave the language its name, as µFP is an extension of Backus's FP [6] with exactly this one extra combining form.

The Π-Ware equivalent of µ is called the DelayLoop constructor, but it works in exactly the same way: given a combinational circuit, it "loops" part of the output back into that circuit's input, passing it through a memory element (a *clocked latch*). Being the *only* way to introduce loops, it also ensures that circuits contain no *combinational loops*.

The example of a shift register – with its μFP geometrical semantics shown in Figure 2.2 – has the following equivalent description in Π-Ware:

```
shift : ℂ B B
shift = delayℂ pSwap
```

When simulated, this circuit produces an output stream in which the tail is a copy of the input stream and the head is an "undefined" Atom (the Atom indexed by zero). The output stream can be said to be "shifted" by one clock cycle, thus the circuit's name.

Making use of dependent types, Π-Ware "tags" circuits with information on whether or not they have cycles (and thus state). In this way, *combinational simulation* (which disregards past inputs) only happens when a circuits is *provably* cycle-free.

Lava

From Lava [11], Π-Ware borrowed the idea of *connection patterns*. Connection patterns are functions that, given one or more circuits as arguments, produce a larger, more powerful circuit by connecting its arguments in a certain *pattern*. Usually these connection patterns are parameterized by a *size*, and are defined by induction on it.

One example in Lava of such a connection pattern is `compose`, the sequential composition of a list of circuits. The code for `compose` is shown on Listing 30.

```
compose [] inp = inp
compose (circ : circs) = out
    where x   = circ inp
          out = compose circs x
```

Listing 30: Lava's `compose` connection pattern.

And for comparison, Π-Ware's version of it (named seqs′) is shown in Listing 31.

```
seqs′ : ∀ {k io} → Vec (ℂ′ io io) k → ℂ′ io io
seqs′ {_} {io} = foldr (const $ ℂ′ io io) _⋙′_ pid′
```

Listing 31: Π-Ware's version of the `compose` connection pattern.

Even though seqs′ is written by using a fold on the vector of circuits and Lava's `compose` uses explicit recursion, the behaviour of both is the same: they connect the given circuit in series, i.e., with the output of circuit n connected to the input of circuit $n + 1$. It is important to notice, however, how Π-Ware uses an *explicit* sequential combination *constructor* in the fold (_⋙′_), while Lava uses Haskell's own function application syntax for this – the output of the first circuit in the list (`x = circ inp`) is fed to the remainder of the "chain".

In Lava's `compose`, an ordinary Haskell `List` is used as the collection of circuits to be connected. Therefore, as Haskell's lists are homogeneous, all of its elements – in this case, all of the circuits to be combined – must have the *same type*. Using Agda, this unnecessary constraint imposed upon the collection of circuits could be lifted, and a more general pattern achieved.

The minimum requirement to connect a collection of circuits in series is that their *interfaces match*, i.e., the output of circuit n has the same size as the input of circuit

$n+1$. The PiWare.Patterns module is still very much a work-in-progress, and therefore such a generalized sequencing pattern is not yet implemented, but we expect that the concept of *typed lists* as defined in a paper by McKinna and Wright [22] will be useful to capture this notion of *constrained* heterogeneous list.

Moving away from connection patters, one aspect in which Π-Ware improves significantly upon Lava is the *size checking* of circuits. As already mentioned briefly in Section 2.2.1, Lava's usage of Haskell lists to express generically-sized circuit definitions leaves some mistakes to be detected only at runtime.

For example, in the case of a generically-sized adder, the two inputs are expected to be of the *same size*, but the `List` type cannot provide this guarantee, thus if mismatching inputs are provided, the error will only be detected at runtime. Notice how in the example Lava definition of a binary adder, the case with input lists of mismatching sizes is left undefined, making a pattern match failure possible:

```
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 contrast, Π-Ware makes use of dependent types and has its circuit type ($\mathbb{C}'$) *indexed by the size* of circuit inputs and outputs, thus enforcing these constraints easily. Listing 32 shows the example of a ripple-carry added modelled in Π-Ware.

```
ripple : (n : ℕ) → let W = Vec B n in ℂ (B × W × W) (W × B)
ripple zero     = pid || pFst ⟫ pSwap
ripple (suc m) =
      pid     || (pUncons || pUncons ⟫ pIntertwine)
  ⟫   pAssoc
  ⟫ fadd    || pid
  ⟫   pALR
  ⟫ pid     || ripple m
  ⟫   pARL
  ⟫ pCons  || pid
  where pAssoc = pARL ⟫ pARL || pid
```

Listing 32: Example of a ripple-carry adder modelled in Π-Ware.

Notice how the input to the circuit defined in ripple is a product where two of the factors *must be of the same type*, namely, a boolean vector of size n, for any n. Furthermore, the description *style* changed drastically: in Π-Ware, circuits are described in a *nameless* fashion, i.e., internal signals cannot be named and connected arbitrarily, *plugs* must be used to do the job of *rewiring* (reordering, forking, etc.). Some examples of plugs used in ripple are pCons, pALR and pIntertwine, all of them defined in PiWare.Plugs together with several others.

Finally, perhaps the biggest improvement of Π-Ware when compared to Lava and similar EDSLs is the possibility to use high-level, Agda types to model and simulate circuits. In contrast, Lava's circuit descriptions are essentially untyped – there is only a

choice between `Bool`, `Int`, and tuples and lists thereof. For example, when modelling a CPU one might want to have a datatype definition corresponding to instructions of that CPU. In Lava, the best bet would be a simple *type synonym*, something along these lines:

```
type SB = Signal Bool
type ALUControlBits = (SB, SB, SB, SB, SB, SB)
type HackInstruction = (SB, SB, SB, SB, SB, SB, SB, SB, SB, SB, ALUControlBits)
```

This description is not only cumbersome to work with (tuples don't have as many useful associated functions as sized vectors), but also error-prone. Any other tuple of 16 bits which might happen to be in scope (for example, from a register bank or some other part of the circuit), can be passed *by mistake* to a function accepting a `HackInstruction`, and the type checker will not be able to detect this mistake.

In Π-Ware, one can normally design a custom datatype for HackInstruction and, by defining an instance of ⇓W⇑ (`Synthesizable`) for it, have circuits operating over this high-level datatype.

ForSyDe

ForSyDe [27] and Π-Ware are both deeply embedded, but in ForSyDe the user writes combinational circuits as Haskell functions and then they are *reified* by Template Haskell into an AST that ForSyDe actually uses to build the circuit at compile-time. On the other hand, circuits in Π-Ware are written with a specific syntax, using specific constructors and combinators.

In terms of type safety, ForSyDe already improved a little over Lava, by essentially *emulating dependent types*. While ForSyDe's "emulation" is far from the full power of dependent types (implementing only type-level naturals and statically-sized vectors), it is already enough to make circuit descriptions more "typed", less error prone and easier to understand.

The range of types which can be *synthesized* with ForSyDe, however, is still small compared to Π-Ware. The already mentioned datatype for a CPU instruction, for example, can easily be used as a circuit's input or output type in Π-Ware, but not in ForSyDe. In ForSyDe, the most typed possible representation would have an *enumerated type* for the *opcodes*[1], but simple sized vectors for the other fields (addresses, etc.); ForSyDe can synthesize enumerations, but not other, more complex forms of user-defined datatypes.

This limitation has to do with the different ways in which Π-Ware and ForSyDe define the semantics of a *synthesizable type*. In ForSyDe, to synthesize a datatype means to actually translate it to an equivalent in VHDL. Standard VHDL [1] has (limited) typing capabilities (namely, enumerations and records are supported), so ForSyDe will translate a Haskell `Int16` to a correspondingly ranged integer in VHDL, and a Haskell `OpCode` enumerated type will result in a corresponding VHDL enumeration.

Π-Ware, on the other hand, generates untyped VHDL, where all circuit ports and signals are vectors of the chosen Atom type (for example, `std_logic`). In this way, any type which can be mapped into a vector of Atoms – by writing an instance of ⇓W⇑ – can be used as circuit IO, and the circuit will be able to be synthesized. This design decision was taken as we believe that all correctness checking and architectural manipulation of circuits should be done at the meta (Agda) level anyway, so when a VHDL netlist is generated, a typed representation would not bring enough benefits. Furthermore, the

[1] Shortening of "operation codes", usually written in assembly language as `ADD`, `LOAD`, etc.

typed VHDL generated by ForSyDe is problematic in terms of tool support. When – in the experimentation project [25] carried out as preparation for this thesis – we tried to implement ForSyDe-generated VHDL on an FPGA, it could be processed by one manufacturer's toolchain (Altera™) but not the other (Xilinx™).

To conclude, when comparing Π-Ware to functional hardware EDSLs in general, the most significant feature is the capability to *prove properties of circuit behaviour*, and to be able to formulate statements and proofs *in the same language as the circuits themselves*. Some functional hardware EDSLs (Lava, for example), are capable of exporting formulas for model checking with *external* tools (such as SAT solvers), but only by using dependently-typed programming proofs can be written in the same language as circuits are described.

4.2 Dependently-Typed Hardware EDSLs

With Coquet [9] Π-Ware shares the basic ideas about circuit *syntax*. Both use a *structural* representation, in which specific constructors of the circuit datatype represent patterns of connection between subcircuits. Also, both Π-Ware and Coquet use indexed data families to represent circuits, in which the indices carry information about the circuit's input and output ports.

The difference lies in exactly what indices are used: Π-Ware's circuits[2] are indexed simply by two natural numbers, representing the *size* of a circuit's input and output. For example, the type of an n-bit adder in Π-Ware would be something like the following:

$$\mathsf{adder} : \mathbb{C}'\ (n + n)\ (\mathsf{suc}\ n)$$

An instance of such an adder (for a particular n) can add two n-bit numbers, producing an output of size $n + 1$. The information that this circuit has *two* inputs, however, is not expressed in the signature – The signature says that the circuit has *one input of size* $n + n$. Π-Ware's low-level descriptions (which are the ones effectively synthesized) are translated into VHDL entities with *always* one input and one output. The *knowledge of how the circuit's ports are organized* is not present in the VHDL level, only in Agda.

Coquet, on the other hand, allows for more *structured* types to be used as indices in the `Circuit` data family. In fact, any type belonging to Coquet's `Fin` (finite) class can be used. The problem, however, is that the requirement imposed upon types in order to belong to `Fin` is that they need to be *enumerable*, that is, there needs to be a list defined with all the type's elements. Enumerations, products, coproducts, etc. all can easily have instances of this type class. But the class provides *no information on how to synthesize them*, i.e., how to write the VHDL *port declaration* when a circuit's input is a complex sum-of-products.

[2] low-level circuits, in the $\mathbb{C}'$ family

Chapter 5

Conclusions

Hardware design is an important and *complex* activity, with strict requirements. The relevance of hardware design skills is growing, due to a combination of the continued effect of Moore's law (which still holds for the near-to-mid-term future [16]) and diminishing returns for microarchitecture optimizations in traditional CPUs [14].

The combination of these effects creates a demand for what is usually called *heterogeneous computing*, where more and more algorithms benefit from implementation in hardware, and these application-specific components need to have support from operating systems and application software. In this scenario, programmers are pushed to think more and more about hardware design issues, without necessarily having the traditional engineering background.

Functional programming has for a long time allowed for programming in a more abstract and principled fashion, further from details of any specific machine. Building a system using a pure functional programming language makes that system easier to analyze and easier for its properties to be stated and proven.

Given these advantages when developing software, it should come as no surprise that a long-standing line of research is concerned with applying the ideas of functional programming to hardware design. Functional hardware description languages, and later *embedded* domain-specific languages for hardware were developed, and several studies have shown that some of the advantages brought to software development by adopting the functional paradigm could also benefit the world of hardware.

More recently, programming languages with *dependent type systems* were developed, creating the paradigm of Dependently-Typed Programming (DTP). Many researchers regard DTP as being the "next step" in terms of type safety and expressivity, when compared to traditional functional programming. Therefore we investigate in this thesis whether *even more* benefits can be brought to the field of hardware design by DTP, improving upon the advances of functional hardware description languages.

Inspired by the ideas in Coquet [9], and by the strengths and weaknesses of other hardware DSLs studied in a previous project, we developed a hardware DSL – Π-Ware – *embedded in a dependently-typed language* and using as much as possible of the cutting-edge developments in DTP. The development of Π-Ware had an exploratory character: having identified limitations of other functional hardware DSLs, we tried to use features of dependent types in general (and of the Agda language in specific) to solve them.

Π-Ware does allow for more type safety in circuit designs, prevents some classes of frequent design mistakes and provides a framework in which specification, simulation,

synthesis and verification of hardware can be performed *in the same language*. We use dependent inductive families to ensure basic *well-formedness* of circuits, and dependent records (type classes) with recursive instance search to allow for abstraction of the types handled by circuits and the underlying circuit technology (gate family). The Curry-Howard isomorphism [32] gives us a logic interpretation of Agda, in which we can write *types* stating properties of circuits and give *proofs* for them.

However, there are still some "rough edges" in the current implementation of Π-Ware: features that due to low priority and/or time constraints are currently not as polished as we wish them to be. Furthermore, during the development of Π-Ware library, some interesting ideas came up about the application of DTP to hardware development which deserve future investigation.

5.1 Future work

5.1.1 Current limitations and trade-offs in Π-Ware

At the beginning of Π-Ware's development, we were considering how should the low-level circuit type ($\mathbb{C}'$) be indexed. It was clear that the two indices should characterize in some way the circuit's *interface*, that is, its input and output ports. Initially, we defined a *closed universe* of types which low-level circuit descriptions could "operate on". This universe was, essentially, closed under products (_×_), coproducts (_×_) and vectors. Had this attempt gone forward, the type of a low-level description for $\underline{\vee}\mathbb{C}'$ (XOR) might have looked as follows – where A indicates one Atom:

$$\underline{\vee}\mathbb{C}' : \mathbb{C}'\ (\mathsf{A} \boxtimes \mathsf{A})\ \mathsf{A}$$

However, trying to enforce the desired well-formedness rules and perform simulation with this universe construction turned out to be more complex than expected, and ultimately we decided to use *sizes* as the indices of $\mathbb{C}'$. This decision mainly impacts the synthesis semantics: with no *structural* information about circuit ports in the low-level, the generated VHDL has a `Port` declaration with only *one* input and *one* output (of appropriate size).

Another current trade-off in Π-Ware concerns the ⇓W⇑ (`Synthesizable`) type class: high-level simulation semantics ($[\![\_]\!]$) *requires* that the ⇓ (serialization) and ⇑ (deserialization) functions be *inverses* of each other. Currently, however, this requirement is *not* enforced by requiring a *proof* of this fact as a field of ⇓W⇑.

The argument for not enforcing this requirement through the type system is to allow prototyping of circuits without the burden of proof for all types involved. On the other hand, the current implementation is certainly not the only way to provide this flexibility: We could require the proof of inversibility as a field of ⇓W⇑ anyways, and expect the designer to postulate the proofs while prototyping. When ready for synthesis, the `safe` flag can be used with Agda to forbid all postulates.

Lastly, describing and trying to prove facts about complex circuit definitions in Π-Ware can be quite inefficient: type-checking a generically-sized ripple-carry adder can already consume up to 2GB of RAM. This is not, however, an issue exclusively with Π-Ware; other Agda libraries suffer with the same problem. Agda's own documentation *wiki* suggests, in a page dedicated to performance tuning, to use some of Agda's experimental features, such as:

- Proof *irrelevance*, when a proof argument is not used in the body of a definition.

- Marking proofs as abstract, so that they are not unfolded.

5.1.2 Directions for future research

As mentioned briefly in Section 3.2.4, any finite Agda datatype can have an instance of ⇓W⇑ (`Synthesizable`), given a certain encoding. One further research path would, therefore, involve *generic programming*; more specifically, creating a sums-of-products view of Agda datatypes – similar to what has been done in the paper "True Sums of Products" [12].

It would also be necessary to define a generic one-to-one correspondence between the elements of these sums-of-products and vectors of Atoms, independently of what Atomic instance is used. Haskell's `binary`[1] library is likely to provide some inspiration in this regard. Both of these endeavours would involve using Agda's *reflection* [31] mechanism: in order to create a sum-of-products encoding for any (finite) datatype, and to extract a serialized sequence of Atoms from the sum-of-products.

A generally desirable effort would be to prove *in Agda* several algebraic and meta-theoretical properties of Π-Ware. For example, properties such as associativity, commutativity and identity of circuit constructors and combinators could be proven, allowing them to fit into the structures defined in the Algebra module of Agda's standard library. Some properties could even help to guide possible optimization steps in the synthesis: proving μFP's [28] "state law" regarding the DelayLoop constructor, for example, could enable big memory blocks to be subdivided and placed nearer to the combinational block operating on their contents.

Lastly, Π-Ware adopts a relatively *low-level* approach to system description: we describe gates, sizes, wires, etc. It would certainly be interesting to investigate the usage of Dependently-Typed Programming for modelling other levels of abstraction, such as instruction sets, processor microarchitectures, etc. in a *layered* fashion.

[1] `https://hackage.haskell.org/package/binary`

Bibliography

[1] Ieee standard vhdl language reference manual. *IEEE Std 1076-1987*, pages 0_1--, 1988.

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

[3] Ieee guide--adoption of the project management institute (pmi(r)) standard a guide to the project management body of knowledge (pmbok(r) guide)--fourth edition. *IEEE Std 1490-2011*, pages 1--508, Nov 2011.

[4] Thorsten Altenkirch and Nils Anders Danielsson. Termination checking nested inductive and coinductive types. In *Proceedings of workshop on partiality and recursion in interactive theorem provers, PAR'10*. Citeseer, 2010.

[5] 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.

[6] 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.

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

[8] 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.

[9] Thomas Braibant. Coquet: a Coq library for verifying hardware. In *Certified Programs and Proofs*, pages 330--345. Springer, 2011.

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

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

[12] Edsko de Vries and Andres Löh. True sums of products.

[13] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. *ACM SIGPLAN Notices*, 46(9):143--155, 2011.

[14] 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.

[15] 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.

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

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

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

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

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

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

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

[23] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1--2, New York, NY, USA, 2009. ACM.

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

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

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

[27] 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.

[28] 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.

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

[30] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 2--18, Berlin, Heidelberg, 2005. Springer-Verlag.

[31] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In *Implementation and Application of Functional Languages*, pages 157--173. Springer, 2013.

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

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