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

Friday 25th July, 2014

Chapter 1

Introduction

Several factors have been causing an increasing demand for hardware acceleration of algorithms, and there is also 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 [14].

In the software industry, specially in application domains requiring high correctness assurance (such as information security, machine control, etc.), functional programming techniques have long been used to improve productivity and reduce the need for extensive testing and debugging. Advocates of functional programming claim productivity gains of up to 10 times compared to programmers working in an imperative environment.

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)
```

These advantages of functional programming have been confirmed in practice – and functional programming languages are becoming more popular in specific problem domains. Furthermore, 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#, which is equivalent to Haskell's `Maybe`.
- Python's generator expressions, inspired by list comprehensions and lazy evaluation.

In a certain way, we can compare the power of the tools and techniques used nowadays in hardware design to the early days of software development. Of course there are inherent and fundamental differences between the two activities, but this comparison leads us to ask whether recent ideas from programming language research, specially those related to functional programming, could 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 [17], developing functional hardware description languages, such as μFP [16]. Later, a trend emerged of developing Embedded Domain-Specific Languages (EDSLs) for hardware description, hosted in purely functional languages, such as Haskell [12]. Prominent examples of this trend are the Lava [4] family, as well as ForSyDe [15].

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

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

Staying on the "path" of functional programming but going further in type system power we reach Dependently-Typed Programming (DTP). 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 approximate specification and implementation. The type signature of a function can give many more guarantees about its behaviour. A classical example when introducing DTP is the type of sized vectors, along with the safe head function, depicted in listing 1.

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

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

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

In this example, the parameter to head cannot be empty – its size will be at least 1 (suc zero). This is expressed in the parameter's type: Vect α (suc n). When checking the totality of head (whether all cases are covered by pattern

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

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 argument to the function. We can also use dependent types to make the return type of a function more precise. The group function for sized vectors in Agda's standard library has the following type:

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

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

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

Therefore, it returns a collection of groups "sliced" from the original vector, 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 whose elements are 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 also want to make the typing of the circuits as strong as possible, to eliminate as many classes of design mistakes as possible.

Depedent type systems allow us to easily express several 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.

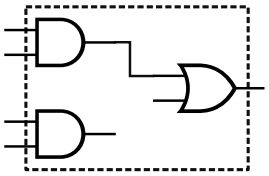

Figure 1.1: Malformed circuit with a "floating" signal.

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 EDSL hosted in the Agda programming language, supporting simulation for combinational and synchronous sequential circuits. Furthermore, the user of Π-Ware can prove expressive properties about circuits in the logic of Agda itself (intuitionistic first-order logic).

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

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

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

Chapter 2

Background

2.1 Hardware Description

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

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

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

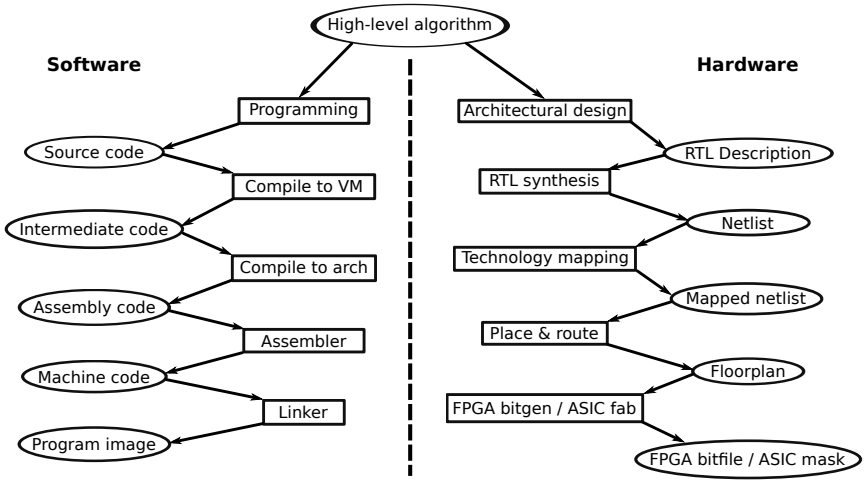

Figure 2.1: Software and Hardware refinement chains.

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

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

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

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

Listing 2: Unsynthesizable VHDL process

Still on the differences between software and hardware development, another important aspect is the level of automatization in the chains (fig. 2.1): usually, all transformation steps in the software chain are automatic. On the hardware chain the crucial "Architectural design" step is manual.

The complex task of making explicit the space-time tradeoff 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 language (VHDL, C) is not expressive enough.
- Specification, verification and synthesis are all done with different tools and different languages.
  - Even behavioural VHDL can be considered practically a different language than Register-Transfer Level VHDL.

Functional programming languages have been touted as a solution to both of these problems. A functional program is a more abstract and expressive specification of an algorithm than a C function or VHDL entity. Also, functional languages could be used to both write the specification of a circuit's behaviour, as well as it's 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 VHSIC Hardware Description Language (VHDL) was being designed and standardized. Even though VHDL and Verilog ended up "winning" and becoming the de facto industry standards, it is still useful to take a look at the ideas behind these early HDLs, as some of them inspired current approaches.

The idea was then to come up with new functional languages, specialized to do hardware description. One of the prominent early examples in this set is μFP [16], which was in turn inspired by John Backus's FP [3]. In contrast to VHDL, which was later adapted to synthesis in an ad-hoc way, μFP was designed since the beginning to have both interpretations:

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

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

The μFP language is an extension of Backus's FP, and contained only one extra combining form: μ. The μ combining form is responsible for the creation of functions (circuits) with internal state. According to the seminal μFP paper [16]:

> 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 one single "μ-wrapper".

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

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

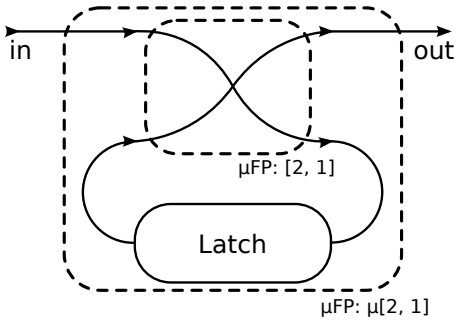

Figure 2.2: Shift register: a simple example of sequential circuit in μFP.

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 [9] of embedded Domain-Specific Languages (DSLs), a trend emerged of also having embedded HDLs. Functional programming languages were a natural fit for hosting DSLs. In particular, the Haskell programming language proved to be a popular choice of host, due to features such as lazy evaluation, highly-customizable and overloadable syntax.

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

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

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

[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 [4] 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 [7]. Currently, Lava has several dialects, such as Chalmers-Lava, Xilinx-Lava, York-Lava and Kansas-Lava. We base our examples in Chalmers-Lava [5], which can be said to be the "canonical" dialect, and also the most actively developed.

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

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

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

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

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

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

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

This limitation of Lava can be solved by dependent types, namely by using statically-sized vectors. Another limitation of Lava is related to the way in which it solves the observable sharing question: in order to detect sharing and

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

cycles, the equality over the `Signal` type is defined as an equality over the references used. Therefore, comparisons of signals that were "created" in different sites will fail, even though their values are the same. For example, the following expressions will evaluate to `False`:

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

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

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 [15]. ForSyDe and Lava differ substantially in description style and internal workings, and a more complete comparison of the two can be found in (cite experimentation project).

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

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

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

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

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

Notice how the `adderFun` process function is built from a "regular" haskell function (`adderFun'`), which is then quoted (by the `[d|` quasi-quoter), processed by `newProcFun` and finally spliced back into place. The process function (combinational) 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 quite any haskell function can be quoted and processed by `newProcFun` and turned into a ForSyDe process function: the argument and return types of

a `ProcType` must belong to the `ProcType` type class. Instances of this class are provided only for:

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

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

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

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

Pointed notation Declarations with point-free notation are not accepted

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

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

Hawk and Cλash

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

Hawk [11] is a Haskell-hosted EDSL with a different target than the ones presented until now: instead of modelling circuits, it intends to model microarchitectures. Therefore, it has a higher level of abstraction than Lava or even ForSyDe.

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

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

- Haskell's `List` datatype doesn't quite match the intended semantics for Hawk's `Signals`: the preferred semantics for `Signal` should be a truly infinite, coinductive stream. The authors mention a parallel effort of them to embed Hawk in the Isabelle theorem prover, which achieved the desired

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

semantics. In Π-Ware, we make use of coinductive streams [10] to model the inputs and outputs of synchronous sequential circuits, exactly as the authors of Hawk desired.

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

Finally, we would like to mention Cλash [2]. Even though it is not, strictly speaking, an embedded DSL, it is close enough to deserve a quick look.

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 super-compilation.

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, it would be silly to add an integer number and a character string. In fact, it is hard to imagine an addition operation accepting such arguments and having nice algebraic properties. To define addition sensibly, therefore, we need the help of a type system, to "ban" all programs that would try to add these "incompatible" values.

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

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

The most basic form of abstraction – values depending on values (the concept of functions) – is supported in practically all type systems. The result of functions can also depend on the types of the arguments (this is called polymorphism).

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. Now 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 would be a comparison-based sorting algorithm, in which depending on the type of the elements in the collection, different comparison operators will be used.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.2 Dependent types

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

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

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

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

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

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

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

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

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

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

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

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

This remark – that a programming language (with it's type system) can also be seen as a logic – goes back to the early days of computing, and is known by several names, among which "Curry-Howard isomorphism" and "propositions

as types" [18]. It initially was "discovered" as a connection between the simply-typed lambda calculus and intuitionistic propositional logic. As decades went by, this correspondence was found to be much more general, and several connections were drawn between diverse typed lambda calculi and logics.

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

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

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

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

Listing 4: Order relation (⩽) over naturals, as an Agda indexed data family.

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

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

Listing 5: Proof that the ⩽ relation is transitive.

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

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

```
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" [13], three examples are shown of how to embed DSLs in Agda and get corresponding Domain-Specific Type Systems (equipped with desirable properties) "for free".

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

2.4 Hardware and dependent types

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

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

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

Chapter 3

The Π-Ware library

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

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

3.1 Circuit Syntax

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

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

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

The indices of $\mathbb{C}'$ represent, respectively, the size of the circuit's input and output. The size of a circuit input or output port can be thought of as the number of "wires" entering (resp. leaving) that circuit. This notion will become much more concrete in section 3.3, with a detailed presentation of the simulation semantics.

The several constructors of $\mathbb{C}'$ "calculate" the port sizes of the circuits they construct, based on the sizes of the given arguments. These calculations implement structural well-formedness rules for circuits, ensuring that, for example:

- No wires are floating
- Short-circuits (having two or more sources connected to the same load) cannot happen

```
data ℂ′ where
  Nil  : ℂ′ zero zero
  Gate : (g# : Gates#) → ℂ′ (ins g#) (outs g#)
  DelayLoop : {i o l : ℕ} (c : ℂ′ (i + l) (o + l)) {p : comb′ c} → ℂ′ i o

  -- Structure related
  Plug : {i o : ℕ} → (f : Fin o → Fin i) → ℂ′ i o
  _⟫′_ : {i m o : ℕ} → ℂ′ i m → ℂ′ m o → ℂ′ i o
  _|′_ : {i₁ o₁ i₂ o₂ : ℕ} → ℂ′ i₁ o₁ → ℂ′ i₂ o₂ → ℂ′ (i₁ + i₂) (o₁ + o₂)
  _| +′ _ : {i₁ i₂ o : ℕ} → ℂ′ i₁ o → ℂ′ i₂ o → ℂ′ (suc (i₁ ⊔ i₂)) o
```

Listing 6: The core circuit type (ℂ′) of Π-Ware.

3.2 Abstraction Mechanisms

- Data abstraction (atom vectors x arbitrary synthesizable types)
  - Atom (enumeration axioms)
  - Word / Synthesizable
    - Proof search?
- Gate abstraction (Technology primitives x Semantic fundamentals)
  - Circuits descriptions are parameterized by a set of fundamental gates
    - Always combinational
    - Their correctness is assumed

  For synthesis, still need to give these gates a description in terms of technology primitives

3.3 Circuit Semantics

- Two types of evaluation: combinational and sequential
- Combinational eval requires a proof that the circuit contains no loops
  - Eval of a fundamental gate is just its definitional behaviour
- For sequential circuits we use a causal stream semantics
  - Current output depends on the current input and (possibly) on the past input
  - Different than plain Stream functions
- There's also an eval function which allows the circuit to be viewed as function from Stream to Stream

Chapter 4

Conclusions

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

4.1 Discussion and Related Work

Lorem ipsum…

4.2 Future work

- What problems Π-Ware still has.
  With suggestions for how to solve them, if possible
- More broadly, considering Π-Ware, what would be the “next steps”.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] 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.

[16] 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.

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

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