Π-Ware: Hardware Description and Verification in Agda

João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling

Utrecht University
Department of Information and Computing Sciences
3584CC – Utrecht – The Netherlands
`{J.P.PizaniFlor,W.S.Swierstra,Y.Sijsling}@uu.nl`

Abstract

There is a long tradition of modelling digital circuits using functional programming languages. This paper demonstrates that by employing *dependently typed programming languages*, it becomes possible to define circuit descriptions that may be simulated, tested, verified and synthesized using a single language. The resulting domain specific embedded language, Π-Ware, makes it possible to define and verify entire families of circuits at once. We demonstrate this by defining an algebra of parallel prefix circuits, proving their correctness and further algebraic properties.

1998 ACM Subject Classification B.7.2 Integrated Circuits – Design Aids – Verification, D.3.2 – Language Classifications – Functional Languages, F.3.1 – Logics and Meanings of Programs – Specifying and Verifying and Reasoning about Programs

Keywords and phrases Dependently-Typed Programming, Agda, EDSL, Hardware Description Languages, Functional Programming

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

There is a long tradition of using functional programming to model hardware circuits. Dating as far back as the 1980's, there have been many different proposed Domain-Specific Languages (DSLs) for the design and verification of circuits [21, 1, 19, 4]. Initially these languages were mostly standalone, but later *embedded* DSLs for hardware were developed, hosted in functional languages such as Haskell or ML.

All of these *functional hardware* DSLs have some limitations with respects to *(type) safety* or aspects of the design process that they support. Most of them allow for simulation, some support synthesis to Register-Transfer Level (RTL) netlists; others are focused on verification (using a combination of SMT solvers, automated theorem provers, or interactive proof assistants). We would argue, however, that none of these are *unified* typed languages for the design, simulation, verification, and synthesis of hardware circuits.

Yet the need for better tools for the design of custom hardware accelerators is greater than ever. The performance of general purpose processors is becoming increasingly harder to improve, as we have already long ago faced the power wall and Instruction-Level Parallelism (ILP) shows diminishing returns [8]. There is a growing demand for hardware acceleration that is both easy to maintain and accurate.

This paper presents Π-Ware[1], an Hardware Description Language (HDL) embedded in *Agda* [17, 16], a dependently-typed programming language. Π-Ware provides a single, strongly-typed language

[1] Project hosted at `http://piware.alvb.in`

21st International Conference on Types for Proofs and Programs.
Editors: Billy Editor and Bill Editors; pp. 1–24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

which supports the definition, synthesis, simulation, testing and formal verification of complex circuits. After giving a high-level overview of the language and its features (Section 2), this paper makes the following contributions:

- Π-Ware has been designed to be a safe and strongly-typed language. Our embedding (Section 3) makes use of dependent types to provide safety guarantees beyond the ones offered by HDLs embedded in simply-typed functional languages. The embedding is parameterized by the type of fundamental data over the wires and the library of fundamental gates being used. For example, instead of using only logic gates, one could add binary arithmetic operators to the fundamental library. This raises the level of abstraction of the description and simplifies verification.
- Unlike other embeddings in proof assistants (such as Coquet [4]), Π-Ware circuits are executable. We defined functional semantics for both combinational and sequential circuits (Section 4). This may seem trivial, but providing a total semantics of circuits that run forever requires some care. Specifically, we define a *causal stream-based* semantics to model the simulation of sequential circuits.
- As the circuit semantics is executable, we can use it to test our designs. Furthermore, for finite domains, we can automatically derive a proof by exhaustive testing (Section 5). More generally, the full power of Agda as an interactive theorem prover can be used to prove properties of *circuit generators*. These generators are usually defined using some sort of recursive pattern (*connection pattern*), and proofs about them rely on induction following the same pattern (*proof combinator*).
- Finally, we show how all these features may be combined in a single case study: the verification of parallel prefix circuits (Section 6). We describe generally this family of circuits in terms of Π-Ware primitives and verify its behaviour. In particular, we formulate and proof algebraic laws involving operators and transformations over parallel prefix circuits, providing machine-verified versions of proofs previously developed on paper [11].

2 Overview

We can best illustrate circuit models in Π-Ware by analyzing a relatively simple example. Let us model a 2-way multiplexer (mux). For convenience, we consider that booleans are being carried over the wires, and that we have the usual set of fundamental gates at our disposal: $\{\text{NOT}, \text{AND}, \text{OR}\}$.

A first step when designing a circuit is to think of its *specification*. For such a small circuit as mux, a *truth table* defines it concisely enough. Our mux has two data inputs (A and B), and also a selection input (S). It should behave in such a way that, whenever ($S = 0$), the output (Z) should be equal to input A, otherwise the output should be equal to input B. This is expressed in Table 1.

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 1 Truth table specification of mux.

From the truth table we can (straightforwardly) derive a boolean formula:

$$Z = (A \land \neg S) \lor (B \land S) \tag{1}$$

From this logical formula, a designer could then *implement* a circuit with the structure shown in Figure 1. This kind of graphical model is often known as *block diagram*.

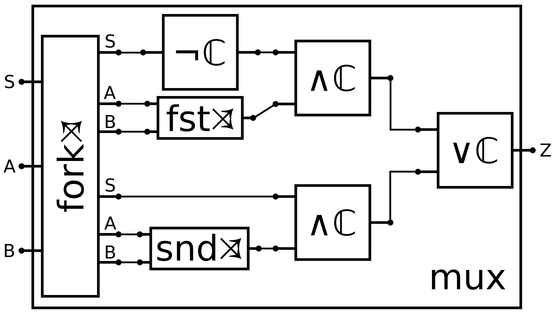

Figure 1 Block diagram of `mux`.

We can also view such a diagram in a different way, by considering the fundamental gates present, and grouping them using *sequential* (_⟫_) and *parallel* (_||_) composition. This corresponds exactly to the definition of mux in Π-Ware, shown in Listing 1.

```
infixl 4 _⟫_
infixr 5 _||_

mux : ∀ {s} → ℂ {s} 3 1
mux =  fork×⤪
   ⟫    (¬ℂ || fst⤪₁ ⟫ ∧ℂ) || (id⤪₁ || snd⤪₁ ⟫ ∧ℂ)
   ⟫    ∨ℂ
```

1 Π-Ware model of `mux`.

In this description, we use three kinds of fundamental gates: `AND` (∧ℂ), `OR` (∨ℂ) and `NOT` (¬ℂ). Notice how the circuit type is *indexed* by the *sizes* of the input and output. The *total* size of all inputs of mux amounts to 3 and total size of all outputs amounts to 1. Besides the fundamental gates and composition, we also use some blocks to do *rewiring*. The circuit called fork×⤪ outputs two exact copies of its input bus side-by-side, while fst⤪$_1$ and snd⤪$_1$ select respectively the first and second wire from an input of size 2.

The mux example shows how Π-Ware circuits are described in a low level of abstraction. Circuits are combined in an *architectural* way, and there is no way *in the DSL* to refer to intermediary results (no binding). We discuss how this low-level description relates to other levels of abstraction in Section 7.

In the definition of mux, the size parameters to the ℂ type constructor are constants, but they need not be in general. Using dependent types, we can precisely define and reason about *circuit generators*. For example, in Listing 2 we define muxN, an inductively defined circuit generator. A term "muxN n" takes two inputs of size n and produces an output of the same size. In this definition, nil⤪ stands for the *empty* circuit (has zero outputs and does no computation), while adapt⤪ does the necessary reordering of the wires (associativity and commutativity).

```
muxN : ∀ n {s} → ℂ {s} (1 + (n + n)) n
muxN zero      = nil⤨
muxN (suc n)   = adapt⤨ n  ⟫  mux ∥ muxN n
```

```
adapt⤨ : ∀ n {s} → ℂ {s} (1 + ((1 + n) + (1 + n))) ((1 + 1 + 1) + (1 + (n + n)))
```

2 Π-Ware model of the muxN generator.

Even though circuit semantics is only given in Section 4, we can already see what would be possible given a functional semantics for our examples. For now, let's assume the semantic function for circuits has the following type:

```
⟦_⟧ : ℂ i o → (Vec Bool i → Vec Bool o)
```

That is, it takes a circuit with inputs of *size* i and outputs of *size* o, and returns its semantics: a function between appropriately-sized binary words. One first possibility is to just *test* mux to gain confidence in its correctness. As dependent types may perform evaluation during type checking, we can formulate our tests as a type checking problem, requiring that our circuit will compute the required type by definition:

```
test1 : ⟦ mux ⟧ (false  :: (true :: false :: [])) ≡ (true   :: [])
test2 : ⟦ mux ⟧ (true   :: (true :: false :: [])) ≡ (false  :: [])
```

This approach works fine to check the correctness of the simple mux: just write one test for each line of the truth table. However, we cannot write an *infinite* number of tests, so to definitely convince ourselves of the correctness of muxN we will want to *prove* a more general statement, such as:

```
muxN⊑ite : ∀ n → ⟦ muxN n ⟧ (s :: (a ++ b)) ≡ if s then b else a
```

Here we can regard the if_then_else_ function as a *specification* for muxN, and muxN⊑ite as a *proof* that muxN complies with this specification. This proof can be written by induction on n. The base case will be trivially true (as [] is the single inhabitant of Vec Bool zero). The successor case will rely on a *similar correctness proof of* mux, along with the inductive hypothesis and auxiliary lemmas involving adapt⤨ (commutativity, associativity, etc.).

We discuss proofs of circuit (generator) properties more thoroughly in Section 5. In that section we also talk about a notion of *equivalence* between circuits and several algebraic properties of circuit constructors and *combinators*. As a prerequisite for verification, however, we first must precisely define the syntax and semantics of circuits, respectively in Sections 3 and 4.

3 Circuit structure

The *syntax* of Π-Ware models gives a low-level description of a circuit's *architecture*, indicating how fundamental *gates* are connected to each other to perform a certain task. This style approximates *block diagrams* usually drawn by hardware designers, but with a key distinction: in Π-Ware, components are connected to each other in a *nameless* fashion, without explicitly naming ports or wires.

As Π-Ware is a *deeply-embedded* DSL, the syntax of the language is defined by a datatype (called ℂ). Our DSL distinguishes between *combinational* and *sequential* circuits. In summary, sequential circuits can be said to have *internal state*, while combinational ones do not. We denote this distinction by *indexing* the ℂ type with an element of IsComb:

```
data IsComb : Set where σ ω : IsComb
```

By convention, we consider circuits indexed with the ω (omega) value to be sequential. The choice was slightly motivated by the usage of Σ^{ω} in mathematics to represent the set of all infinite sequences over a given alphabet Σ. With the combinational/sequential distinction clear, we present ℂ in Listing 3, the core of the whole library.

```
data ℂ : {s : IsComb} → ℕ → ℕ → Set where
  Gate  : ∀ g {s}            → ℂ {s} (|in| g) (|out| g)
  Plug  : ∀ {i o s} → i ⤪ o  → ℂ {s} i o

  _⟫_   : ∀ {i m o s}          → ℂ {s} i m     → ℂ {s} m o     → ℂ {s} i o
  _||_  : ∀ {i₁ o₁ i₂ o₂ s}   → ℂ {s} i₁ o₁   → ℂ {s} i₂ o₂   → ℂ {s} (i₁ + i₂) (o₁ + o₂)

  DelayLoop : ∀ {i o l} → ℂ {σ} (i + l) (o + l) → ℂ {ω} i o
```

3 The circuit datatype (ℂ).

The first thing to notice is that besides IsComb, the circuit datatype (ℂ) is also indexed by two natural numbers. These correspond, respectively, to the *total* number of input wires into the circuit and total number of output wires from the circuit. We were strongly influenced in our circuit syntax design choices by Coquet [4], especially in the usage of dependent types in the (_⟫_) and (_||_) constructors to enforce sizing constraints.

In order to facilitate discussion of the constructors of ℂ, we categorize them as either *primitive* or *composite*: composite constructors take arguments of type ℂ, while primitive ones do not. First, we look at the primitive constructors:

- Circuits constructed with Gate are the smallest possible ones *with computational content*. The whole PiWare.Circuit module is parameterized by a *gate library* (detailed in Section 3.2), and by calling Gate we simply pick one of those gates to use as building block.
- The other primitive constructor is Plug, which is necessary due to the *nameless* fashion in which we compose circuits. Since it is impossible to refer to any specific circuit port we cannot, for example, map the “first” output of a circuit to the “second” input of another. Plugs are required to do *rewiring*, but they perform *no computation*.

The argument to the Plug constructor has type i ⤪ o, and this is just a synonym for a (first-order representation of a) function from output wires (indices) to input wires (indices).

```
_⤪_ : ℕ → ℕ → Set
i ⤪ o = Vec (Fin i) o
```

An intuitive definition for (i ⤪ o) would be (Fin o → Fin i), but we opted for the Vec representation in order to get easier combination of plugs and easier proofs. Using such a mapping, no Plug can ever be built containing any information other than the origin of each output wire.

The composite constructors in Π-Ware represent ways in which smaller circuits can be connected to form a larger one. First, let us focus on the most interesting of them: DelayLoop. Both other composite constructors (_⟫_ and _||_) *preserve* the IsComb index. The DelayLoop constructor, however, is an exception: it is the only way to build a sequential circuit given a combinational one as argument.

This single possible way to *introduce state* makes the definition of circuit semantics simpler and, as the name hints, we make sure to always introduce a *clocked delay* at each occurrence of DelayLoop.

In this way we avoid *combinatorial loops* in the circuit, which can make circuit analysis significantly more complex [9].

The remaining composite constructors of ℂ are:

- Sequential composition (_⟫_), which connects the output of one circuit to the input of another. The indices ensure that the interfaces are *compatible*, i.e, that they have the same size.
- Parallel composition (_||_), that creates a combined circuit which has as inputs (resp. outputs) the inputs (resp. outputs) of *both* constituent subcircuits.

The careful indexing of the sequential and parallel composition, together with the type of _⤪_, ensure that some simple design mistakes are *prevented by construction*. Namely, floating wires are forbidden by _⟫_: in a term "c_1 ⟫ c_2", the output size of c_1 needs to equal the input size of c_2. Also, because Plug takes a *function* from outputs to inputs, only one source can be assigned to each load (no short-circuits). Lastly, the *totality* of the argument to Plug ensures that no plug output can be left unassigned.

As already mentioned, our circuit syntax is strongly influenced by Coquet [4]. Some differences are the partitioning of circuits by the IsComb tag into *combinational* or *sequential*, the first-order Plugs, and the type of the DelayLoop constructor, which in our case does not allow nesting of state.

In Π-Ware, circuits are parameterized both by the type of data travelling in the wires (an Atomic type) and by a set of fundamental Gates upon which all circuits are built. The first design choice taken when describing circuits in Π-Ware is which Atomic type to use, so let's start with that.

3.1 Atom types

In hardware descriptions written in VHDL or Verilog, often the information carried on the wires is modelled as bits. This description stays close to a physical implementation and thus remains popular. However, sometimes it is useful to think of other types being carried in the wires: for example, a small enumeration type better describes the possible states of a state machine. In Π-Ware, all circuit descriptions are *parameterized* by the type of element carried over the wires.

Types that can be carried over circuit ports and wires are called *atomic* types. Elements of these types are considered to have *no parts* and cannot be *inspected* by Π-Ware in any way. Some very simple examples of atomic types are: Bool, (named) enumerations and Fin n (for some n).

Perhaps the simplest useful example of an atomic type is the often-used Bool. When using the interface offered by Π-Ware, we can *enumerate* the elements of Bool, and we can *test* whether any two elements of Bool are equal, but no other information can be extracted.

In order to be used as an atomic type, a given type must be *finite* and *non-empty*. We pack the type itself together with these requirements in the Atomic record (Listing 4), therefore all circuit descriptions must be parameterized by an *instance* of such record.

```
record Atomic : Set₁ where
  field  Atom : Set
         enum : FiniteNonEmpty Atom

  open FiniteNonEmpty enum public

  W  = Vec Atom
```

4 The `Atomic` record.

The first *field* (Atom) of the Atomic record is the Agda Set denoting the type of elements carried over one wire. The second field (enum) is an instance of the FiniteNonEmpty record (shown in Listing 5).

```
record FiniteNonEmpty {ℓ} (α : Set ℓ) : Set ℓ where
  field finite   : Finite α
        default  : α

  open Finite finite public
```

5 The FiniteNonEmpty record.

We witness the finiteness of Atom by a bijection with Fin *n*, and the default field shows that the type in question has at least one inhabitant. The reason to forbid empty types from being used as Atom lies in the semantics of DelayLoop.

We will cover circuit semantics with more detail in Section 4 but, in summary, each occurrence of DelayLoop prepends one extra element to the circuit's output stream. This extra element will have type Vec Atom *n* (for arbitrary n), and thus we need to have *at least one arbitrary value* of type Atom at our disposal (default).

As a last remark, we make W *n* a synonym for Vec Atom *n*. Thus in any context parameterized by an instance of the Atomic record, we can refer to words of atoms in a more convenient way.

3.2 Fundamental Gates

The mux example from Listing 1 was constructed using the usual boolean gates (AND, OR, and NOT). Instead of hardwiring the choice of fundamental gates in the definition of circuits, Π-Ware allows users to choose their own collection of *fundamental gates*. These could be the boolean gates mentioned above, but more complex circuits, such as muxes, registers, or arithmetic circuits could all be regarded as fundamental, depending on the particular design.

To define a particular choice of fundamental gate library, users are required to define a suitable Agda record specifying the interface and semantics of each gate. This record (called Gates) is shown in Listing 6.

```
record Gates : Set₁ where
  field Gate        : Set
        |in| |out|  : Gate → ℕ
        spec        : ∀ g → (W (|in| g) → W (|out| g))
```

6 The Gates record.

The type of gate identifiers is stored in the Gate field, and there are functions that assign to each gate identifier its corresponding number of inputs (|in|), number of outputs (|out|), and specification function (spec). Notice the highly dependent type of spec and of Gates as a whole: the return type of spec depends on it's *g* parameter and on the |in| and |out| fields. The type of |in| and |out| depends, in turn, on Gate.

The choice of fundamental gates strongly influences circuit correctness proofs: the correctness of the Atomic and Gates are *assumed* rather than proved.

To perform boolean logic with our circuits, we will want to use any *functionally complete* set of boolean gates. A particularly simple such set is {NAND}, which contains only the negated AND gate.

First, we must define how many input and output ports does each gate in the library have:

```
|in| |out| : NandGate → ℕ
|in|  ⊼ℂ′ = 2
|out| ⊼ℂ′ = 1
```

Notice that the parameter of the |in| and |out| functions is of type NandGate. This is the type of *gate identifiers* in the library. There are no requirements imposed on a type to satisfy this role, but for readability we use a named enumeration. Having defined the interface of each gate in our library (there is only one), we then define the specification function:

```
spec−⊼ℂ : W 2 → W 1
spec−⊼ℂ (x ∷ y ∷ []) = [ not (x ∧ y) ]
```

There are no restrictions imposed by Π-Ware on which kind of gate should or should not be present in a library, and higher-level Atomic and Gates instances can make designs much simpler. For example, with an Atomic instance defined to represent 8-bit signed integers, there can be a useful Gates library containing some set of modular arithmetic operators over these integers.

As another example of gate library, Π-Ware also includes BoolTrio, a gate library operating over booleans with three boolean operations (`NOT`, `AND`, `OR`) and two constant gates (`FALSE` and `TRUE`). We specify the behaviour of the gates using the boolean functions from Agda's standard library (Data.Bool).

When *simulating* a Π-Ware circuit, we will use the specification functions of the gate library used in that circuit. Likewise, in proofs of circuit correctness, the fundamental gates are assumed to be correct. Therefore the elements in a Gates library can be seen as fundamental in two ways:

- Fundamental *behaviour*, as they have no subparts.
- Fundamental *functional correctness*, as it is assumed.

3.3 Abstraction levels

Throughout this paper, we will deal with circuit models and circuit semantics only in terms of their *size* (the ℂ datatype is indexed by two natural numbers, representing the sizes of a circuit's input and output). However, Π-Ware offers a thin *data abstraction* layer, allowing Agda types in circuit's inputs/outputs (instead of Vec Atom).

A typed circuit is defined as just a wrapper record around a sized circuit (ℂ). Therefore, the computation still is performed over words, but the description of a typed circuit contains information on how to *convert* between elements of the involved Agda types and the correspondingly-sized words.

This thin layer makes mainly *simulation* and testing more convenient and less verbose (no need to always build vectors to compare with in testing, for example). However, as the computation is still performed over words, proving a circuit's correctness will still rely on lemmata involving the *sized* level (vectors, atoms).

We discuss with more detail in Section 7 how this layer of data abstraction influences modelling and verification, and what could be other possible ways of raising the level of abstraction in circuit description.

4 Circuit semantics

Due to the choice of using deep embedding to implement our DSL, it is possible to write several different semantics for circuit models.

When talking about deeply embedded languages, a semantic function is just a function mapping the Abstract Syntax Tree (AST) of our DSL to a desired *carrier* type. All of the circuit semantics currently implemented in Π-Ware are *compositional*, which means that they can be defined by *folding* the ℂ type with an algebra.

The module PiWare.Circuit.Algebra defines the *algebra type* for ℂ (as a record), along with the associated *catamorphism* (fold). There are two algebra types: one for combinational circuits (ℂσA) and one for (possibly) sequential ones (ℂA). The only difference between them is that a case for DelayLoop is absent from ℂσA. Here we show the algebra type for combinational circuits (ℂσA):

```
record ℂσA : Set where
  field GateA  : ∀ g#              → T (|in| g#) (|out| g#)
        PlugA  : ∀ {i o} → i ⤪ o  → T i o
        _⟫A_   : ∀ {i m o}         → T i m   → T m o   → T i o
        _||A_  : ∀ {i₁ o₁ i₂ o₂}   → T i₁ o₁ → T i₂ o₂ → T (i₁ + i₂) (o₁ + o₂)
```

We use Agda's feature of *sections* (similar to Coq's sections) to parameterize the catamorphism by an algebra instance and the algebra type itself by a *carrier* (called T). Within the section, the definition and type of catamorphism become very simple and understandable.

Listing 7 shows the catamorphism for combinational circuits (cataℂσ). Notice how (due to the σ index) there is no need to define a clause for DelayLoop.

```
cataℂσ : ∀ {i o} → ℂ {σ} i o → T i o
cataℂσ (Gate g)   = GateA g
cataℂσ (Plug f)   = PlugA f
cataℂσ (c₁ ⟫ c₂)  = cataℂσ c₁ ⟫A cataℂσ c₂
cataℂσ (c₁ || c₂) = cataℂσ c₁ ||A cataℂσ c₂
```

7 Catamorphism for combinational circuits.

4.1 Combinational simulation

As a particular example of such a compositional semantics, we defined *executable* simulation for Π-Ware circuit models, which maps circuits to the domain of Agda functions. This semantics is *executable* in the sense that, by applying the function obtained using the semantics to an input, the same output should be calculated as if the circuit had been implemented in hardware and run.

When getting the simulation semantics of a combinational circuit, we want to obtain a function between appropriately-sized words, that is, a circuit of type "ℂ i o" should result in a function of type "W i → W o". Thus the carrier type for the combinational simulation algebra is:

```
W⟶W : ℕ → ℕ → Set
W⟶W m n = W m → W n
```

With the appropriate carrier defined, we get a very simple type and definition for the combinational simulation function:

```
⟦_⟧ : ∀ {i o} → ℂ {σ} i o → W⟶W i o
⟦_⟧ = cataℂσ simulationσ
```

```
gateσ        = spec
plugσ p ins  = tabulate (flip lookup ins ∘ flip lookup p)
seqσ         = flip _∘′_
parσ f₁ f₂   = uncurry′ _++_ ∘ map f₁ f₂ ∘ splitAt′ _
```

8 Simulation semantics for combinational circuits.

Notice how the type of the semantic function *requires* the interpreted circuit to be combinational (it must be indexed by σ). In this way, the algebra used (simulationσ) does not have a field for the DelayLoop case. We show on Listing 8 the definitions for each of the fields in the combinational simulation algebra.

The cases for sequential (seqσ) and parallel composition (parσ) are self-evident, relying respectively on function composition and map over products. In the case of fundamental gates, we simply rely on that gate's specification function. This leaves the most interesting definition: plugσ.

In the case of a Plug, we build the output word *pointwise* by using tabulate. The tabulate function from Agda's standard library "fills" a (Vec *α n*) by evaluating a given function of type (Fin *n* → *α*) on all points of its domain. In our case, each of these points is an output index (element of Fin *o*). First, we lookup the output index in the plug mapping *p*, obtaining the corresponding input index. Then we use this index to lookup the input word and place the correct Atom on the output.

Let's now consider a simple example circuit, its simulation semantics and the involved types, to better understand how all these definitions fit into place. We consider a two-input NAND gate (⊼ℂ), with the following type and definition:

```
⊼ℂ : ∀ {s} → ℂ {s} 2 1
⊼ℂ = ∧ℂ ⟫ ¬ℂ
```

The gate is described using two-input conjunction (∧ℂ) and one-input negation (¬ℂ) as building blocks, and these pieces come from the library of fundamental gates that we are using (PiWare.Gates.BoolTrio). By evaluating ⊼ℂ we then obtain the following function:

```
⟦ ⊼ℂ ⟧ : W⟶W 2 1
⟦ ⊼ℂ ⟧ = spec ¬ℂ′ ∘ spec ∧ℂ′
```

To simulate ⊼ℂ we rely on the specification functions of both gates and in function composition. The names ¬ℂ′ and ∧ℂ′ are just the gate *identifiers*. If we reduce the expression above further (expanding spec and W⟶W), we obtain the following:

```
⟦ ⊼ℂ ⟧ : W 2 → W 1
⟦ ⊼ℂ ⟧ = λ { (x ∷ y ∷ []) → [ not (x ∧ y) ] }
```

The verification of circuits and circuit generators will be discussed in detail in Section 5. But it is already clear that it will rely heavily on laws involving vectors, as well as algebraic properties of the fundamental gates and plugs used in the design.

4.2 Sequential simulation

Sequential circuits are those in which their output at any given instant may depend not only on a combination of the input at the same instant, but on the *sequence* of previous inputs.

In Π-Ware, we model only the *discrete time domain*, and therefore a circuit's input *signal*[2] is *piecewise constant* (as well as its output signal). Because of this characteristic, we can model both input and output signals as Streams[3], in which each element of the Stream is a word.

Perhaps the simplest example of a circuit with internal state is shift. This circuit will output at any clock cycle t the value present on its input at the preceding cycle. The architecture of shift consists simply of one DelayLoop and one Plug:

```
shift : ℂ {ω} 1 1
shift = DelayLoop swap⤨₁
```

For circuits which have the ω (omega) tag in their type, we need to use the *sequential execution semantics* (⟦_⟧ω). In the specific case of shift, the function obtained via the sequential simulation semantics will have the following type:

```
⟦ shift ⟧ω : Stream (W 1) → Stream (W 1)
```

The function obtained via ⟦_⟧ω consumes and produces a Stream of adequately-sized words. To explain in detail how the sequential semantics is actually defined, however, we have to mention a key distinction between digital circuits and stream functions in general: An unconstrained stream function (that is, an arbitrary element of type Stream α → Stream β) can (possibly) *look into the future*. Considering Streams over the discrete time domain, one simple example of stream function that "looks into the future" is tail.

```
tail : ∀ {α} → Stream α → Stream α
tail (in₀ ∷ in₁₊) = ♭ in₁₊
```

The element at position 0 in the output of tail depends on the input at position 1, and so forth. Sequential circuits clearly cannot show this behaviour (at least not if we want to physically implement them). As we want our sequential circuits to be synthesizable to actual hardware, we should ensure that our semantics will only ever produce *causal stream functions*.

One way to define a causal stream function is as the unfolding of a function producing only the *next* output, given the current and past inputs. We call these functions *causal step functions*, and they are defined as follows:

```
_⇒c_ : ∀ {ℓ₁ ℓ₂} (α : Set ℓ₁) (β : Set ℓ₂) → Set (ℓ₁ ⊔ ℓ₂)
α ⇒c β = Γc α → β
```

The symbol Γc means *causal context*, and it is defined simply as a non-empty list (List⁺), that is, a pair of the head (current value) with a possibly-empty tail (past values).

```
Γc : ∀ {ℓ} (α : Set ℓ) → Set ℓ
Γc = List⁺
```

Coming back to the definition of simulation semantics for sequential circuits, we can now establish the *carrier* type for the algebra of sequential circuits as being a *causal step function* between words of the appropriate length. Then, the *causal* simulation of a circuit is defined as a catamorphism over ℂ with the simulationc algebra:

[2] By signal we mean a function over the time domain.

[3] A stream is an infinite list, i.e., a list without the Nil constructor.

```
W⇒cW : ∀ i o → Set
W⇒cW i o = W i ⇒c W o
```

```
⟦_⟧c : ∀ {i o} → ℂ i o → W⇒cW i o
⟦_⟧c = cataℂ {aσ = simulationσ} simulationc
```

The definitions of the fields of simulationc make use of the fields in the combinational algebra (simulationσ). The fields GateA and PlugA simply take the *present* value from the causal context and pass it to the corresponding combinational field (gateσ and plugσ), thus we don't show them here. The interesting cases are DelayLoop, as well as the composite constructors (_⟫_ and _||_):

```
delayc {_} {o} f = takeᵥ o ∘ delay o f
seqc f₁ f₂  = f₂ ∘ map⁺ f₁ ∘ pasts
parc f₁ f₂  = uncurry′ _++_ ∘ map f₁ f₂ ∘ unzip⁺ ∘ splitAt⁺ _
```

The delay function is responsible for taking the regular word function taking l extra wires (l stands for "loop") and transforming it into a causal step function (depending on the history of inputs instead of the current state). According to this semantics, a circuit built with DelayLoop corresponds to a *Mealy machine*, where the state has size l, and the combinational circuit inside of it calculates *both* the next output and the next state.

By calling our causal semantic function (⟦_⟧c) over a circuit we obtain a causal *step function*. Then, by just unfolding this step function we obtain the *causal stream function* which is actually the user-facing type for the simulation of sequential circuits:

```
⟦_⟧ω : ∀ {i o} → ℂ i o → (Stream (W i) → Stream (W o))
⟦_⟧ω = runc ∘ ⟦_⟧c
```

In contrast with the type of ⟦_⟧ (the combinational semantics), the type of ⟦_⟧ω makes no requirement on how the circuit parameter should be *indexed*. This means that the sequential semantics can be used to obtain a stream function from both sequential and combinational circuits. In the case of evaluating a combinational circuit using ⟦_⟧ω, the obtained stream function just applies the calculation performed by the circuit *pointwise* on the stream (ignoring the past).

5 Verification

Π-Ware also allows for the *verification* of circuit models. The kind of properties that can be stated and verified depends on the semantics being used. With Π-Ware in its current form, we can express mainly *functional* specifications, that is, those related to the input/output characteristics of the circuit. Furthermore, due to the embedding in a dependently-typed language, Π-Ware allows for both *testing* of any specific circuit, as well as *proofs* of correctness for *circuit generators*.

Tests and proofs can be written which check constraints on the outputs or witness arbitrary relations between the inputs and outputs of a circuit. In particular, the Design Under Test (DUT) can be verified to have the same input/output behaviour as an Agda function *assumed to be correct*. Also, we have defined a notion of *extensional equivalence* between circuits, allowing us to prove algebraic properties of circuit constructors and combinators, as well as to define provably-correct semantics-preserving circuit transformations.

5.1 Testing

Testing can be used by a designer to *gain confidence* in the functional correctness of a model early in the design process, before attempting to write proofs in their full generality. Writing test cases can also be a useful way to capture requirements from whoever commissioned the circuit, thus aiding in *validation*.

Using only the simulation functions (⟦_⟧ and ⟦_⟧c), *manual* test cases can already be written: this method is usually called *unit testing*. These are some examples of unit tests for a two-input mux (the leftmost boolean in the input vector being the *selection* bit):

```
test−mux₁ : ⟦ mux ⟧ (false :: (true :: false :: [])) ≡ [ true ]
test−mux₁ = refl

test−mux₂ : ⟦ mux ⟧ (true :: (true :: false :: [])) ≡ [ false ]
test−mux₂ = refl
```

Going further then unit testing, the focus of this subsection is on Π-Ware's facilities to help *test automation*. For circuits with inputs and outputs of small size, verification via *exhaustive checking* is feasible, and our ultimate goal is to make this as automatic and concise as possible.

The first step of abstraction from manually-written test cases is to have an Agda function serving as specification of the circuit behaviour. This means that, for any possible input to the circuit, evaluating the function with this input will produce an output assumed to be correct.

Continuing with our mux example, let's check its correctness by comparing it with a specification function. First of all, the simulation semantics of mux has the following type:

```
⟦ mux ⟧ : W 3 → W 1
```

Looking at this type, and thinking about the expected behaviour of mux (*selecting* one of two inputs), a reasonable candidate for specification is as follows:

```
ite : W 3 → W 1
ite (s :: a :: b :: []) = [ if s then b else a ]
```

The first required task for automatic exhaustive checking is to *generate* all possible values of the circuit's input type. For this, we need the input type to have an instance of the Finite record, which embodies a *bijection* between the type in question and Fin *n*. The definition of Finite is shown in Listing 9.

```
record Finite {ℓ} (α : Set ℓ) : Set ℓ where
  field |α|      : ℕ
        mapping  : α ↔′ Fin |α|

  open Inverse′ mapping public
```

9 The `Finite` record.

There are some instances of Finite defined in the Π-Ware library for primitive types (amongst which Bool), along with products, sums and vectors. Specifically, the input type of ⟦ mux ⟧ is W 3 (equal to Vec Bool 3), so the necessary Finite instance relies on the pre-defined instances for vectors and booleans.

Having a way to generate all values of a type, we can create a vector containing all of them. More interestingly, we can create a (heterogeneous) vector containing all of the *proofs* that each value satisfies a certain predicate.

We use the fact that a type is Finite to define an ∀-introduction rule for it. Then, because Atom is required to be Finite, and because Vec *α* *n* is also Finite (for any finite *α* and any n), we define ∀-introduction for *words*:

```
∀−W : ∀ {n} {P : W n → Set} {ps : vec↑ (tabulate (P ∘ fromFinite ⦃ Finite−W ⦄))}
      → (∀ w → P w)
```

In particular, we can then simply use ∀−W to prove properties involving all possible inputs of a circuit. As long as the chosen property *P* is *decidable*, the *ps* parameter will reduce to a nested product of units and Agda will infer it.

The property in which we are particularly interested is whether a circuit and a given specification function *agree* on a certain input.

```
_⊑?_at_ : ∀ {i o} (c : ℂ {σ} i o) (f : W i → W o) → (W i → Set)
c ⊑? f at w = T ⌊ (⟦ c ⟧ w) ≟W (f w) ⌋
```

This relation relies on a decidable equality over output words of the checked circuit (_≟W_), and uses it to compare the result obtained by running both the circuit simulation and specification function. By passing this relation as the *P* parameter in ∀−W, we get the function we ultimately wanted: check⊑.

```
check⊑ : ∀ {i o} (c : ℂ i o) (f : W i → W o) {ps : vec↑ (tabulate (c ⊑? f at_ ∘ fromW i))} → c ⊑? f
check⊑ c f {ps} = ∀−W {P = c ⊑? f at_} {ps}
```

This function performs automatic exhaustive checking, in order to verify that a circuit complies with a given specification function. It is feasible to use check⊑ for verification of small circuits such as mux, or for small parts of bigger designs (parts with few ports). However, for big designs, and in particular to verify circuit *generators*, we need to resort to manually-written proofs.

5.2 Proofs

The key advantage brought to verification by using a dependently-typed language is that properties can be proven not only of any *specific* circuit, but of *circuit generators*. Circuit generators are *parameterized* definitions from which for each value of the parameter, a different circuit can be derived.

The term "circuit generator" itself comes from the Lava [1] EDSL, but the idea of parameterized definitions is *at least* as old as VHDL's *generics* [13]. The parameters of these circuit generators are usually structural properties of the circuit, such as the *sizes* or amount of inputs and outputs of a circuit. Another example would be configuring how many clock cycles does the input get delayed in a shift register.

Usually, these definitions will be *recursive*, and thus the proofs of statements involving these generators will then be performed by induction. A circuit generator muxN, that selects between *two* inputs of size *n* each, has the following type and definition:

```
muxN : ∀ n {s} → ℂ {s} (1 + (n + n)) n
muxN zero     = nil⤨
muxN (suc n)  = adapt⤨ n ⟫ mux || muxN n
```

For a given value of the parameter n, this definition produces a circuit with input size 1 + (n + n) (1 selection bit, plus n bits for each input) and output size n. The base case is a circuit with one input and zero outputs, and that matches the size of the empty plug (nil⤨). In the recursive case, we connect the 2-input mux and the recursive call (muxN n) in parallel, and we need a Plug (called adapt⤨) to make the right wires meet the right ports. By the type of adapt⤨ it should be clear how it performs this routing:

```
adapt⤨ : ∀ n {s} → ℂ {s} (1 + ((1 + n) + (1 + n))) ((1 + 1 + 1) + (1 + (n + n)))
```

The first 3 bits in the output size of adapt⤨ (1 + 1 + 1) are exactly those needed by the mux, while the remaining ones are consumed by muxN n.

As already mentioned before, the choice of specification function has a significant impact on the proof of correctness for a circuit. In the case of muxN, the specification is iteN:

```
iteN : ∀ n → W (1 + (n + n)) → W n
iteN zero     _                  = []
iteN (suc n)  (_ :: ab)          with splitAt (suc n) ab
iteN (suc n)  (s :: .(a ++ b))   | a , b , refl = if s then b else a
```

In iteN, the tail of the input is split into two equal parts and we use if_then_else_ to choose (based on the selection bit), which of the two parts will be the output. Then the proof that muxN complies with its specification will need to follow the same induction pattern used to define the specification itself. Namely, we need to pattern match on n and do a case analysis on the result of splitting the tail of the input.

```
muxN⊑iteN : ∀ n → muxN n ⊑ iteN n
muxN⊑iteN zero     (_ :: []) = refl
muxN⊑iteN (suc n)  (_ :: ab)          with splitAt (suc n) ab
muxN⊑iteN (suc n)  (s :: .(a ++ b))   | a , b , refl = muxN⊑iteN′
```

Unfortunately, the full proof of muxN⊑iteN is a bit too long to be completely analyzed here (we abbreviate at muxN⊑iteN′). The proof relies of course on the proof of correctness for mux (the basic circuit with type ℂ 3 1). It also relies on properties of the adapt⤨ plug, ensuring that it's semantics essentially commutes and associates the arguments of functions in the necessary way.

5.3 Connection patterns

We are interested not only in proving properties of circuits in isolation, but also about the behaviour of so-called *connection patterns*[4]. Connection patterns are just functions taking circuits as inputs and producing circuits as outputs. Typically they use the constructors of ℂ to *connect* their arguments in a certain fashion, thus the name.

A (very simple) example of connection pattern is parsN, which connects n copies of a given circuit in parallel. The type and definition of parsN are:

```
parsN : ∀ {k i o s} → ℂ {s} i o → ℂ {s} (k ∗ i) (k ∗ o)
parsN {k} {i} {o} c = subst₂ ℂ  (∗-sum-replicate k i) (∗-sum-replicate k o)
                                (pars (replicateI₂ {n = k} c))
```

[4] This name comes from Lava as well.

Notice how the input and output sizes of the combined circuit are *statically guaranteed* to be correct, as they are calculated from the input/output sizes of the circuit passed as parameter. This definition (parsN) is a special case of a more general pattern: instead of replicating the same circuit n times, we can connect a whole vector of (different) circuits in parallel. This is achieved by pars:

```
pars : ∀ {n s} {is os : Vec ℕ n} (cs : VecI₂ (ℂ {s}) is os) → ℂ {s} (sum is) (sum os)
pars {is = []}      {[]}      []I₂          = nil⤨
pars {is = _ ∷ _}  {_ ∷ _}  (c ∷I₂ cs)  = c ∥ pars cs
```

As the parameter of pars we need a special kind of vector, ensuring that only elements of types built with a given type constructor ($\mathbb{C}$) can be present in the vector. This special kind of vector is VecI$_2$, what we call an *index-heterogeneous vector* [5]. It is heterogeneous in the sense that its elements have different types, but only the indices vary, and the type constructor is fixed for all elements.

Another case of basic connection pattern is seqsN, taking a circuit and connecting n copies of it in sequence:

```
seqsN : ∀ k {s io} → ℂ {s} io io → ℂ {s} io io
seqsN k = seqs ∘ replicate {n = k}
```

Notice how the input and output sizes of the argument circuit are the same (io). This is because the _⟫_ constructor forces the output size of a circuit in this sequence to match the input size of the next.

Also seqsN is a special case of a general pattern: connecting a vector with n circuits in sequence. For this connection to be even *possible*, we need the input/output sizes of the circuits in the vector to be *pairwise compatible*. This means that for each circuit, its output size must be equal to the input size of the next. To this end, we adapt the work done on *type-aligned sequences* [18] to a dependently-typed setting.

We are currently working in establishing lemmata about the behaviour of these connection patterns in order to make proofs involving their usage simpler. For example, the simulation behaviour of seqsN can be shown to be that of the iterate function, that is:

$$\forall\, w \to [\![\ \mathsf{seqsN}\ k\ c\ ]\!]\ w \equiv (\mathsf{iterate}\ k\ [\![\ c\ ]\!])\ w$$

Furthermore, we are also working on expressing connection patterns as folds over the underlying (indexed heterogeneous) vectors, as that would allow for more general and powerful laws.

5.4 Circuit equivalence

Until now we have talked about relations between a circuit and a function — such as the *complies with* relation (i.e. "$c \sqsubseteq f$"). However, it is also very important to have an *equivalence relation* between circuits themselves. Given a properly-defined such relation, we can then have at our disposal laws like "c ⟫ id⤨ ≋ c", allowing for *provably safe* circuit optimizations.

We have defined such a notion of circuit equivalence *up-to-simulation* for *combinational* circuits, and a similar notion (and laws) for sequential circuits is left for future work. In this section we explain the several iterations we have gone through until achieving the current definition of circuit equivalence, correcting a small issue at each step. In the most naïve and first attempt, we just require identical inputs and compare the outputs of simulating both circuits using (propositional) equality.

[5] More specifically, VecI$_2$ only handles type constructors with *two* indices.

$$\begin{aligned}&\_\!\equiv\!\mathsf{c}\_ : \forall\ \{i\ o\}\ (c_1\ c_2 : \mathbb{C}\ i\ o) \to \mathsf{Set}\\ &c_1 \equiv\!\mathsf{c}\ c_2 = \forall\ w \to [\![\, c_1 \,]\!]\ w \equiv [\![\, c_2 \,]\!]\ w\end{aligned}$$

This definition is very unsatisfactory though, because it can only be used to compare circuits with *definitionally equal* indices, i.e, we cannot compare $(c_1 : \mathbb{C}\ 1\ n)$ with $(c_2 : \mathbb{C}\ 1\ (n + 0))$. The first improvement over this definition is to use *vector equality* to compare the outputs. The notion of *semi-heterogeneous* vector equality ($\_\!\approx\!\_$) is defined in Agda's standard library and it considers two vectors equal whenever the elements are *pointwise* propositionally equal. The new definition of circuit equivalence looks as follows:

$$\begin{aligned}&\_\!\cong\!\_ : \forall\ \{i\ o_1\ o_2\} \to \mathbb{C}\ i\ o_1 \to \mathbb{C}\ i\ o_2 \to \mathsf{Set}\\ &c_1 \cong c_2 = \forall\ w \to [\![\, c_1 \,]\!]\ w \approx [\![\, c_2 \,]\!]\ w\end{aligned}$$

While now the problem of word size ($n + 0$ vs. n) has been solved for the outputs, the same issue remains for the input: we cannot yet compare $(c_1 : \mathbb{C}\ n\ 1)$ with $(c_2 : \mathbb{C}\ (n + 0)\ 1)$. Ultimately, what we want for circuit equivalence is to ensure that, when given "vector equal" inputs, both circuits will generate "vector equal" outputs:

$$\begin{aligned}&\_\!\approxeq\!\_ : \forall\ \{i_1\ o_1\ i_2\ o_2\} \to \mathbb{C}\ i_1\ o_1 \to \mathbb{C}\ i_2\ o_2 \to \mathsf{Set}\\ &\_\!\approxeq\!\_\ \{i_1\}\ \{\_\}\ \{i_2\}\ \{\_\}\ c_1\ c_2 =\\ &\quad \forall\ \{w_1 : \mathsf{W}\ i_1\}\ \{w_2 : \mathsf{W}\ i_2\}\\ &\quad \to w_1 \approx w_2 \to [\![\, c_1 \,]\!]\ w_1 \approx [\![\, c_2 \,]\!]\ w_2\end{aligned}$$

This is the definition that *almost* gets us there. It has a big problem, though: it's unsound. Very easily we can construct a term of type $(c_1 \approxeq c_2)$ simply by making sure the hypothesis is false. A simple example of a term that should be banned by $\_\!\approxeq\!\_$ but is allowed is the following:

$$\begin{aligned}&\approxeq\!\!-\!\mathsf{unsound} : (c_1 : \mathbb{C}\ 0\ 0)\ (c_2 : \mathbb{C}\ 1\ 1) \to c_1 \approxeq c_2\\ &\approxeq\!\!-\!\mathsf{unsound}\ c_1\ c_2\ ()\end{aligned}$$

To solve this issue, we make an extra requirement for two circuits to be considered equal. Now, not only vector equal inputs must lead to vector equal outputs, but also there must be a proof that the sizes of the input words are propositionally equal.

$$\begin{aligned}&\mathsf{data}\ \_\!\mathrel{\text{≋}}\!\_\ \{i_1\ o_1\ i_2\ o_2\} : \mathbb{C}\ i_1\ o_1 \to \mathbb{C}\ i_2\ o_2 \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{refl}\!\mathrel{\text{≋}} : \ \{c_1 : \mathbb{C}\ i_1\ o_1\}\ \{c_2 : \mathbb{C}\ i_2\ o_2\}\ (i\!\equiv : i_1 \equiv i_2)\\ &\qquad\qquad \to c_1 \approxeq c_2 \to c_1 \mathrel{\text{≋}} c_2\end{aligned}$$

This is the definition of circuit equivalence that we use, to state algebraic properties of circuit constructors and combinators, and also in the case study discussed in Section 6. For extra convenience, we packed up ($\mathbb{C}$, $\_\!\mathrel{\text{≋}}\!\_$) into an indexed setoid structure, and we added to Agda's standard library some facilities for *equational reasoning* with indexed setoids. All in all, this allows proofs about circuit equivalence to be written in a very nice-looking style. A good example of such a proof can be seen in Listing 11 of Section 6.

6 Case study: parallel prefix circuits

In order to put in practice the definitions of Π-Ware we decided to perform a case study involving *parallel prefix circuits*. Parallel prefix circuits are a wide family of circuit architectures that compute *scans*, that is, given a binary operator $\oplus$ and a vector of inputs $[x_0, x_1, x_2, ..., x_n]$, it will calculate the output vector $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), ..., (x_0 \oplus ... \oplus x_n)]$.

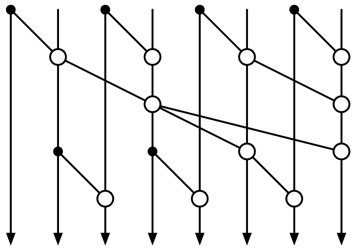

Figure 2 Example of an 8-input parallel prefix circuit.

When talking about parallel prefix circuits, we always assume that the binary operator $\oplus$ is *associative*, thus allowing different parts of the output to be calculated *in parallel*. In Figure 2 we show an example of a circuit utilizing maximal parallelism to calculate a scan with 8 inputs.

In this style of diagram, the data flows from top to bottom, each black dot is a forking point for wires and each white circle is an occurrence of the binary operator. Our case study was heavily influenced by the paper "An Algebra of Scans" [11]. In this paper, the author defines a set of primitives and combinators from which any scan circuit can be built, then states and proves algebraic properties of these combinators.

Our work consisted of formalizing the same primitives and combinators using Π-Ware, and proving the same basic algebraic facts about these combinators. Also, we formalized what exactly means to be a scan circuit, and proved that applying *scan combinators* to scan circuits will result in a scan.

Several of the primitives and combinators defined in the original paper [11] match exactly those present in Π-Ware, amongst them sequential (_⟫_) and parallel composition (_||_) along with the identity plug (id⤨). This coincidence makes the case study especially fruitful, as several of the basic algebraic properties assumed in the original paper could be proved in Π-Ware. For example, sequential combination (_⟫_) forms a monoid of circuits, with id⤨ as identity:

```
⟫−left−identity : ∀ {i o} (c : ℂ i o) → id⤨ ⟫ c ≋ c
⟫−left−identity c = ≅⇒≋ (from−≡ ∘ cong ⟦ c ⟧ ∘ id⤨−id)
```

```
⟫−assoc :  ∀ {i m n o} (c₁ : ℂ i m) (c₂ : ℂ m n) (c₃ : ℂ n o) → (c₁ ⟫ c₂) ⟫ c₃ ≋ c₁ ⟫ (c₂ ⟫ c₃)
⟫−assoc c₁ c₂ c₃ = ≅⇒≋ (from−≡ ∘ λ _ → refl)
```

In fact, the need to state and prove these basic algebraic laws was what led us to develop the notion of circuit equivalence (Section 5.4) in the first place. We predict that such algebraic structures over circuits will be important when reasoning about circuit transformations and synthesis. For example, the identity laws for id⤨ allow us to *remove* such plugs from any circuit, while being *certain* that the functional behaviour will not change.

The concept of scan circuit itself was formalized by defining a "prototype" scan, which was assumed to be correct. This definition is very inefficient (in terms of gate usage and also in depth), but has a very simple inductive definition:

```
scan : ∀ n → ℂ n n
scan zero =      id⤨₀
scan (suc n) =  id⤨₁ || scan n ⟫ fan (suc n)
```

Besides the parts already mentioned (_⟫_, _||_, id⤨), here we also use the fan primitive. A term "fan n" has type ℂ n n, and calculates $[x_0, (x_0 \oplus x_1), (x_0 \oplus x_2), ..., (x_0 \oplus x_n)]$. The diagram in Figure 3 illustrates the structure of "scan 3".

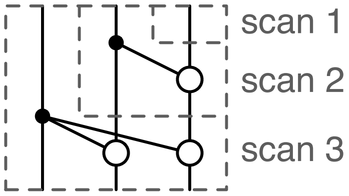

Figure 3 Structure of the prototype scan of size 3.

Using this specification, we could prove that several different architectures all indeed compute a scan. The proofs rely on the fact that all of these architectures are built by combining smaller scans into bigger ones.

Namely, we defined in Π-Ware the *sequential scan combinator* (called _▱_) and the *parallel scan combinator* (called _▯_). The sequential scan combinator connects the last output of its first argument into the first input of the second argument. For the parallel combinator both scan circuits are put side-by-side, and an extra `fan` connects the last output of the first argument to all inputs of the second.

Having defined those combinators, we then proved that their definition indeed satisfy their namesake property: whenever given scans as arguments, they produce a scan as output:

```
▱−law  : scan (suc m) ▱  scan (suc n)  ≈ scan (m + suc n)
▯−law  : scan (suc m) ▯  scan n        ≈ scan (suc m + n)
```

As an example of a proof involving lots of these algebraic properties, we show the correctness of a *serial scan*. A serial scan is a parallel prefix circuit of maximal depth, as it makes no use of parallelism at all, and it has the structure shown in Figure 4.

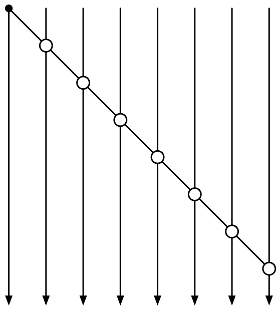

Figure 4 Structure of a serial scan.

The Π-Ware description for serial is pretty simple and makes essential use of the parallel scan combinator (_▯_). The code for serial is shown in Listing 10

```
serial : ∀ n → ℂ n n
serial zero            = id⤨ 0
serial (suc zero)      = id⤨ 1
serial (suc (suc n))   = serial (suc n) ▯ id⤨ 1
```

10 Π-Ware description of a serial scan.

Finally, we can then prove that serial does indeed compute a scan. The proof (Listing 11) relies on the key fact that _▯_ preserves scans. Furthermore, it also relies on the fact that _▯_ is a congruence with regards to circuit equivalence, and that two calls of scan with equal arguments will be equivalent (scan−cong).

```
serial−is−scan : ∀ n → serial n ≈ scan n
serial−is−scan zero              = ≈−refl
serial−is−scan (suc zero)        = id⤨₁≈scan₁
serial−is−scan (suc (suc n))     = begin
      serial (suc (suc n))
   ≈⟨⟩ - definition of serial (suc (suc n))
      serial (suc n) ▯ id⤨ 1
   ≈⟨ serial−is−scan (suc n) ▯−cong id⤨₁≈scan₁ ⟩
      scan (suc n) ▯ scan 1
   ≈⟨ ▯−law n 1 ⟩
      scan (suc n + 1)
   ≈⟨ scan−cong (cong suc (+−comm n 1)) ⟩
      scan (suc (suc n))
   ∎
```

11 Proof that `serial` computes a scan.

7 Discussion

Related work

There are numerous languages for hardware description; there is a wide variety of techniques that may be used for hardware verification, including the usage of automatic theorem provers, SAT solvers, model checking, and interactive theorem provers, notably HOL [6].

Systems such as ACL2 have been used to prove correctness of entire microprocessors [12], and the maturity of the ACL2 and HOL ecosystems is clearly visible in the highly optimized engines and large scale of some of the formalization efforts done using these languages. One of the key differences with our approach is the use of a typed higher-order host language, with which we can also have *higher-order specifications* for *connection patterns*. For example, the behaviour of the parsN pattern is equivalent to a functorial *map* over vectors.

The field of formal methods and functional programming applied to hardware design is indeed a crowded one, thus rather than attempt to survey these fields here, we will restrict ourselves to the most closely related work. There has been a great deal of work in the last thirty years marrying functional programming and hardware design, leading to languages such as Lava [1], Hawk [15] and ForSyDe [19]; Sheeran [20] gives an excellent overview. None of these, however, combine *all* of higher-order type safety, inductive proofs, synthesis and an executable semantics for both combinational and sequential circuits.

The idea of using dependent types to describe circuits is not new and can be traced back as far as Hanna [10], The paper “Constructing Correct Circuits” [3] gives a clear example of how dependent types can *tie together* specification and implementation. In this paper, the authors give a mapping between Peano naturals and binary numbers, then used to build a (ripple-carry) binary adder which is *correct by construction*. The approach taken in Π-Ware is significantly different. Rather than carry the functional specification of a circuit *in its type*, we clearly separate the construction, testing, and verification of circuits. This means that, for example, a designer can first simulate some instances of a design and get confidence in its correctness before trying to *prove* it. This greater degree of freedom may be particularly useful when exploring the design space, deferring the testing and verification effort until a satisfactory candidate design has been found.

Some of the most complete EDSLs for hardware (Coquet [4] and Fe-Si [5]) are hosted in the Coq theorem prover. Our design and implementation has been particularly inspired by Coquet. Both Coquet and Π-Ware use a similar *structural* and *nameless* description of circuits, parameterized by the type of gates. The most important difference between Π-Ware and Coquet, however, is that Π-Ware defines a *functional* semantics for circuits, while Coquet uses a *relational* semantics, i.e., the semantics are specified by defining a suitably indexed inductive data type. The choice of semantics style is crucial: Π-Ware circuits can be tested, simulated, and verified as any other Agda function.

Where Coq's richer language for proof tactics may provide a great deal of automation, the functional semantics presented here reduces *for free*, without relying on the invocation of tactics or proof search. We expect to reap the benefits of a functional semantics while combining them with some proof-by-reflection techniques [22, 14]. Furthermore, we can use Agda features such as goal and context reflection, as well as solvers for algebraic structures (monoids, semirings, etc.) [2].

Future work

Equality plugs

When explaining the behaviour of Plugs in Section 3, we said that they perform no computation. But further than that, some plugs in fact have also *no structural effect*. By this we mean plugs whose mapping is the identity function. They usually are an expression of arithmetic equalities over circuit indices, such as associativity:

```
assoc⤪ : ∀ {a b c} → ((a + b) + c) ⤪ (a + (b + c))
assoc⤪ {a} {b} {c} = eq⤪ (+-assoc a b c)
```

The need to place such a Plug between two circuits is essentially an artifact of Intensional Type Theory (ITT). In the sequence constructor (_⟫_), the output index of the first parameter and input index of the second must be *definitionally* equal, that is, they must have the same normal form. If Agda had the *equality reflection rule*, then equalities involving indices could be used during type checking, and we would not need to insert *equality plugs*.

Right now we are investigating two approaches to make this issue less inconvenient. Firstly, we can use the *ring solver* from Agda's standard library (coupled with reflection) to automatically solve index equalities and introduce the corresponding plugs whenever needed. Secondly, there was a recent addition to Agda of a language pragma called `REWRITE`, which allows for user-defined equalities to be added to Agda's typing rules, essentially turning Agda into an Extensional Type Theory (ETT). We will investigate how the use of this pragma affects our library and examples.

Functional language

Even though we are using a functional language to model our circuits, the circuit description themselves are very low-level. In particular, we need to rewire intermediate results explicitly using our Plug constructors. While our style closely resembles the netlist representation of circuits, we would like to provide circuit designers with a more high-level, applicative interface.

One problem that we must address to do so, however, is that of observable sharing [7]. Any domain specific language for hardware description embedded in a general purpose functional language must, at some point, ensure that the sharing and recursion of the circuit definitions are not lost. Although various solutions do exist, these typically place a higher burden on the programmer through the necessity of explicit fixed-point and sharing combinators or rely on specific compiler support. We hope to find a satisfactory solution to this problem in the context of dependently typed programming languages such as Agda, and use this to define a more "functional" layer on top of the definitions presented here.

Typed circuits

While Π-Ware rules out certain errors, such as short-circuits, we would like to investigate how to provide stronger static guarantees. So far, we have parameterized the type of circuits by the *size* of their inputs and outputs; we have started investigating how to parameterize circuits by their *type*.

For example, the type of a 2-input multiplexer would then become "ℂ (Bool × (Bool × Bool)) Bool", rather than the less informative "ℂ 3 1". To add extra type information to our circuits, we define a record wrapper for typed circuits (Listing 12).

```
record ℂ {s : IsComb} (α β : Set) {i j : ℕ} : Set where
  constructor Mkℂ
  field ⦃ α ⦄  : ⇓W⇑ α {i}
        ⦃ β ⦄  : ⇓W⇑ β {j}
        base  : ℂ {s} i j
```

12 Typed Circuit type.

Here we require that the input and output types of our circuits are synthesizable – that is, they can indeed be represented in our simulation semantics (as vectors of atoms). By adding a series of *smart constructors* that produce and combine such typed circuits, we can provide a more convenient and type-safe interface to our library. We are currently extending our library with such type-safe definitions, including the use of reflection to generate the required serializer/deserializer and proof that they are inverses.

8 Conclusion

With Π-Ware we have only started to explore the benefits that dependent types offer to digital circuit design. Π-Ware and the wider Agda ecosystem may not be mature enough yet to compete with some of the existing commercial tools and more mature prover technology; nonetheless we believe that the combination of the executable circuits, static types, and compositional proofs that Π-Ware offers form a novel point in the design space.

All the examples we have developed up to now, especially the case study on scan circuits, lead us to believe that this is indeed a fruitful avenue of study. By treating circuits as first-class objects in a dependently-typed language, we can reason about their behaviour and prove algebraic properties of relations, operators *over* circuits, and circuit generators. At the same time, we can simulate our designs and synthesize netlist descriptions. It should come as no surprise that type theory, a language of both computation and proof, provides a perfect setting for hardware verification and simulation.

Acknowledgments

We would like to thank the helpful comments and suggestions of the attendants of the TYPES2015 workshop in Tallinn where we presented our initial results on Π-Ware. The participation in other venues such as for instance the Midlands Graduate School 2015 in Sheffield was also very fruitful in allowing discussions about the type-theoretical underpinnings of this work. This work was supported by the Netherlands Organization for Scientific Research (NWO) project on *A Dependently-Typed Language for Verified Hardware*.

References

1 Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999.

2 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer Berlin Heidelberg, 2009.

3 Edwin Brady, James Mckinna, and Kevin Hammond. Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types. In *Trends in Functional Programming 2007*, 2007.

4 Thomas Braibant. Coquet: A Coq Library for Verifying Hardware. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, number 7086 in Lecture Notes in Computer Science, pages 330–345. Springer Berlin Heidelberg, January 2011.

5 Thomas Braibant and Adam Chlipala. Formal Verification of Hardware Synthesis. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, number 8044 in Lecture Notes in Computer Science, pages 213–228. Springer Berlin Heidelberg, January 2013.

6 Albert Camilleri, Mike Gordon, and Tom F Melham. *Hardware verification using higher-order logic*. University of Cambridge, Computer Laboratory, 1986.

7 Koen Claessen and David Sands. Observable sharing for functional circuit description. In *In Asian Computing Science Conference*, pages 62–73. Springer Verlag, 1999.

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

9 Morteza Fayyazi and Laurent Kirsch. Efficient Simulation of Oscillatory Combinational Loops. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 777–780, New York, NY, USA, 2010. ACM.

10 F. K. Hanna and N. Daeche. Dependent Types and Formal Synthesis. *Philosophical Transactions: Physical Sciences and Engineering*, 339(1652):121–135, April 1992.

11 Ralf Hinze. An algebra of scans. In Dexter Kozen, editor, *Mathematics of Program Construction*, number 3125 in Lecture Notes in Computer Science, pages 186–210. Springer Berlin Heidelberg, January 2004.

12 Warren A Hunt. *FM8501: A verified microprocessor*, volume 795. Springer, 1994.

13 IEEE. *Standard VHDL Language Reference Manual*, 1988.

14 Pepijn Kokke and Wouter Swierstra. Auto in agda. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 276–301. Springer International Publishing, 2015.

15 John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. *ACM SIGPLAN Notices*, 34(9):60–69, September 1999.

16 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

17 Nicolas Oury and Wouter Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.

18 Atze van der Ploeg and Oleg Kiselyov. Reflection Without Remorse: Revealing a Hidden Sequence to Speed Up Monadic Reflection. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 133–144, New York, NY, USA, 2014. ACM.

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

20 M Sheeran. Hardware Design and Functional Programming: a Perfect Match. 2005.

21 Mary Sheeran. muFP, a language for VLSI design. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 104–112. ACM Press, 1984.

22 Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer Berlin Heidelberg, 2013.