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 [22, 2, 20, 5]. 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 [9]. 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 [18, 17], a dependently-typed programming language. Π-Ware provides a single, strongly-typed language

\(^1\) Project hosted at http://piware.alvb.in
Π-Ware: Hardware Description and Verification in Agda

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 [5]), 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 [12].

## 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: \{NOT, AND, 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.

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 1** Truth table specification of mux.
From the truth table we can (straightforwardly) derive a boolean formula:

\[ Z = (A \land \neg S) \lor (B \land S) \]  

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.](image)

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

\[
\text{mux} \; : \; \mathbb{C}^{3 \mid 1}
\]
\[
\text{mux} = \text{forkx} \times
\]
\[
\quad (\neg C \parallel \text{fst}\{X_1\} \parallel \land C) \parallel (\text{id}\{X_1\} \parallel \text{snd}\{X_1\} \parallel \land C) \parallel C
\]

Listing 1 Π-Ware model of mux.

In this description, we use three kinds of fundamental gates: AND (\(\land C\)), OR (\(\lor C\)) and NOT (\(\neg C\)). 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 forkx\times outputs two exact copies of its input bus side-by-side, while \text{fst}\{X_1\} and \text{snd}\{X_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 variable 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 give the type and definition of the aforementioned forkx\times.

\[
\text{forkx} \times : \forall \{n\} \rightarrow \mathbb{C} \; n \; (n + n)
\]
\[
\text{forkx} \times \{n\} = \text{Plug} \; (\text{tabulate} \; \{n\} \; \text{id} \; \text{++} \; \text{tabulate} \; \{n\} \; \text{id})
\]

Listing 2 Π-Ware model of the forkx\times generator.
Even though circuit semantics is only given in Section 4, we can already see what would be possible given a functional (simulation) semantics for our examples. For now, let’s assume the semantic function for circuits has the following type:

\[ \langle \_ \rangle : \mathbb{C} \rightarrow (\mathbb{V} \mathbb{B} \rightarrow \mathbb{V} \mathbb{B}) \]

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. A first possibility to “reason” about circuit behaviour is to just test a circuit with some inputs and observe the produced outputs to gain confidence in its correctness. In the following snippet we give some test cases for \( \text{mux} \). As using dependent types implies 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:

\[
\begin{align*}
test1 : \langle \text{mux} \rangle (\text{false} :: (\text{true} :: \text{false} :: [])) \equiv (\text{true} :: []) \\
test2 : \langle \text{mux} \rangle (\text{true} :: (\text{true} :: \text{false} :: [])) \equiv (\text{false} :: [])
\end{align*}
\]

This approach works fine to check the correctness of the simple \( \text{mux} \), just write one test for each line of the truth table. However, one cannot simply test circuit generators (such as \( \text{fork} \times \times \)), as that would entail running the simulation over an infinite number of inputs. To definitely convince ourselves of the correctness of \( \text{fork} \times \times \) we will want to prove a more general statement, such as:

\[
\begin{align*}
\text{fork} \times \times \mathbb{C}^++ & : \forall n \ (w : \mathbb{V} \mathbb{B}) \rightarrow [\text{fork} \times \times \{n\}] \ w \equiv w ++ w
\end{align*}
\]

Here we can regard the concatenation function \((+_+)_\) as a formal specification of \( \text{fork} \times \times \), and \( \text{fork} \times \times \mathbb{C}^++ \) as a proof that \( \text{fork} \times \times \) complies with its specification. What the statement of \( \text{fork} \times \times \mathbb{C}^++ \) intuitively means is that the circuit has the effect of duplicating its inputs into its outputs. The proof of this statement is written by induction on \( w \), and relies on auxiliary lemmas involving vector functions such as \( _++_ \) and \( \text{tabulate} \).

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 \( \mathbb{C} \)). Our DSL distinguishes between combinational and sequential circuits. In summary, sequential circuits can have internal state, while combinational ones do not. We denote this distinction by indexing the \( \mathbb{C} \) type with an element of \( \text{IsComb} \):

\[
\begin{align*}
data \text{IsComb} & : \text{Set} \\
\text{where} & \quad \sigma \omega : \text{IsComb}
\end{align*}
\]

We consider circuits indexed with the \( \omega \) value to be sequential, and those with \( \sigma \) to be combinational. The choice of name for these (one-letter) constructors is only partially arbitrary: we were motivated by the usage of \( \Sigma^\omega \) in mathematics to represent the set of all infinite sequences over a given alphabet \( \Sigma \). This distinction between combinational and sequential circuits results in some
convenience: with the knowledge that a circuit has no state, the type (and definition) of its simulation semantics becomes simpler – just a function between appropriately-sized vectors. Also the proofs involving statically known-to-be stateless circuits are simpler.

Listing 3 shows the circuit type (\( \mathbb{C} \)). The handling of the IsComb tag in each of the constructors tells us which sort of semantics (stateful or not) we need to have from the subparts in order to get the semantics of the whole circuit.

\[
\begin{align*}
\text{data } \mathbb{C} : \{ s : \text{IsComb} \} & \to \mathbb{N} \to \mathbb{N} \to \text{Set} \; \text{where} \\
\text{Gate} & : \forall (g : \text{Gate}) \{ s \} \to \mathbb{C} \{ s \} \{ \#\text{in} \, g \} \{ \#\text{out} \, g \} \\
\text{Plug} & : \forall \{ i o s \} \to \mathbb{I} \times \mathbb{O} \to \mathbb{C} \{ s \} \{ i \} \\
\_ \rangle \_ & : \forall \{ i m o s \} \to \mathbb{C} \{ s \} \{ i \} \times \mathbb{C} \{ s \} \{ o \} \times \mathbb{C} \{ s \} \{ o \} \\
\_ \parallel \_ & : \forall \{ i_1 o_1 i_2 o_2 s \} \to \mathbb{C} \{ s \} \{ i_1 \} \times \mathbb{C} \{ s \} \{ o_1 \} \times \mathbb{C} \{ s \} \{ o_2 \} \to \mathbb{C} \{ s \} \{ i_1 + i_2 \} \times \mathbb{C} \{ s \} \{ o_1 + o_2 \} \\
\text{DelayLoop} & : \forall \{ i o l \} \to \mathbb{C} \{ s \} \{ i \} \times \mathbb{C} \{ s \} \{ o \} \to \mathbb{C} \{ s \} \{ o \} \{ i \} \\
\end{align*}
\]

Listing 3 The circuit datatype (\( \mathbb{C} \)).

The constructors for sequential (\( \_ \rangle \_ \)) and parallel (\( \_ \parallel \_ \)) composition, for example, preserve the \( s \) tag. This means that, to evaluate a circuit (\( c_1 \rangle c_2 \)) in a stateless way, both \( c_1 \) and \( c_2 \) need to be stateless (combinational). Equivalently, if any of the parts is stateful, only a stateful evaluation of the whole is allowed.

Besides IsComb, the circuit datatype (\( \mathbb{C} \)) 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 [5], especially in the usage of dependent types in the (\( \_ \rangle \_ \)) and (\( \_ \parallel \_ \)) constructors to enforce sizing constraints.

In order to facilitate discussion of the constructors of \( \mathbb{C} \), we categorize them as either primitive or composite: composite constructors take arguments of type \( \mathbb{C} \), 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 \( i \times o \), and this is a synonym for a mapping from output wires (indices) to input wires (indices).

\[
\begin{align*}
\_ \times \_ : \mathbb{N} & \to \mathbb{N} \to \text{Set} \\
i \times o & = \text{Vec} \{ \text{Fin} \, i \} \times \text{Fin} \, o
\end{align*}
\]

Using such a mapping, no Plug can ever be built containing any information other than the origin of each output wire. An intuitive definition for \( (i \times o) \) would be \( (\text{Fin} \, o \to \text{Fin} \, i) \), but we opted for the (first-order) Vec representation to get easier combination of plugs and easier proofs. Also, the first-order representation will make synthesis more straightforward.
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 [10]. The remaining composite constructors of \_\_ 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.

Careful indexing of sequential and parallel composition, together with the type of \_\_ ▷ \_, ensure that some design mistakes are prevented by construction. Floating wires are forbidden by \_\_ ▷ \_: in a term “\_ \_ ▷ \_ \_”, the output size of \_ \_ needs to equal the input size of \_ \_. 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 [5]. 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 Atomic types

Hardware descriptions in VHDL or Verilog, often model the information carried on the wires as bits. This 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, an 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 ports and wires are called atomic types. Elements of such types are considered to have no parts and cannot be inspected by Π-Ware. Some 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 Bool. When using Π-Ware’s interface, we can enumerate the elements of Bool, and we can test whether 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 inhabited. We pack the type itself together with these requirements in the Atomic record (Listing 4), and all circuit descriptions must be parameterized by an instance of such record.

```
record Atomic : Set, where
  field Atom : Set
    enum : FiniteInhabited Atom

open FiniteInhabited enum public
W = Vec Atom
```

**Listing 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 FiniteInhabited record (shown in Listing 5).

```agda
class FiniteInhabited {ℓ} (α : Set ℓ) : Set ℓ where
  field finite : Finite α
  default : α
open Finite finite public
```

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.

A last detail to note is that the bijection between the Atom type and Fin n implies the existence of a decidable equality (and decidable setoid structure) for Atom. There is a function in the Agda standard library (called eq?) that gives a decidable equality for any type A, provided there is an injection from A into Fin n. In our case, we pass to eq? the injection obtained as a consequence of the bijection between Atom and Fin n. The relevant definitions are shown in Listing 6.

```agda
_≟_ : ∀ {α : Set} → fa : Finite α → Decidable {A = α} _≡_
_≟_ = let open Finite fa in eq? injection

_\_≡A_ : Decidable {A = Atom} _≡_
_\_≡A_ = \_≟_ \_≡Inhabited enum 

decSetoidA : DecSetoid _≡_
decSetoidA = decSetoid _\_≡A_
```

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.

A last detail to note is that the bijection between the Atom type and Fin n implies the existence of a decidable equality (and decidable setoid structure) for Atom. There is a function in the Agda standard library (called eq?) that gives a decidable equality for any type A, provided there is an injection from A into Fin n. In our case, we pass to eq? the injection obtained as a consequence of the bijection between Atom and Fin n. The relevant definitions are shown in Listing 6.

```agda
Listing 5 The FiniteInhabited record.
```

3.2 Fundamental Gates

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

To define a particular choice of fundamental gate library, users must define a suitable Agda record specifying the interface and semantics of each gate. This record (Gates) is shown in Listing 7.

First of all, the whole Gates module is parameterized by an instance of Atomic, thus fixing W and defining the type of elements that appear in the inputs and outputs of our gates.
record Gates : Set₁ where
  field Gate : Set
  #in #out : Gate → ℕ
  spec : ∀ g → (W (#in g) → W (#out g))

Listing 7 The Gates record.

The type of gate identifiers is stored in the Gate field, and there are functions that assign to each gate identifier a 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 its 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 each gate defined by the Gates record is 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. We impose no requirements on a type to satisfy this role, but here NandGate is a simple enumeration type. 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 ($\mathbb{C}$). 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 lemmas 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 $\mathbb{C}$ type with an algebra.

The module *PiWare.Circuit.Algebra* defines the *algebra type* for $\mathbb{C}$ (as a *record*), along with the associated *catamorphism* (fold). There are two algebra types: one for combinational circuits ($\mathbb{C}_\sigma A$) and one for (possibly) sequential ones ($\mathbb{C} A$). The only difference between them is that a case for *DelayLoop* is absent from $\mathbb{C}_\sigma A$.

We use Agda’s feature of *sections* (parameterized anonymous modules) to avoid repetition of parameters used in several definitions. Firstly, the algebra record type ($\mathbb{C}_\sigma A$) is parameterized by the *carrier* type (called $T$). Also, the catamorphism for combinational circuits (called $\text{cata}_{\mathbb{C}\sigma}$) is parameterized by an instance of the record type $\mathbb{C}_\sigma A$. Here we show the algebra type for combinational circuits ($\mathbb{C}_\sigma A$):

```agda
core \mathbb{C}_\sigma A : Set where
field GateA : \forall g# \to T (\#in g#) (\#out g#)
PlugA : \forall \{ i o \} \to i \times o \to T i o
\_\_A : \forall \{ i m o \} \to T i m \to T m o \to T i o
\_\_\_A : \forall \{ i_1 o_1 i_2 o_2 \} \to T i_1 o_1 \to T i_2 o_2 \to T (i_1 + i_2) (o_1 + o_2)
```

We use Agda’s feature of *sections* (parameterized anonymous modules) to avoid repetition of parameters used in several definitions. Firstly, the algebra record type ($\mathbb{C}_\sigma A$) is parameterized by the *carrier* type (called $T$). Also, the catamorphism for combinational circuits (called $\text{cata}_{\mathbb{C}\sigma}$) is parameterized by an instance of the algebra record.

Listing 8 shows the catamorphism for combinational circuits ($\text{cata}_{\mathbb{C}\sigma}$). Notice how this definition, *by itself*, takes only the circuit to be interpreted as parameter. However, $\text{cata}_{\mathbb{C}\sigma}$ is part of a section, so there might be situations where it takes an extra parameter (an element of the record type $\mathbb{C}_\sigma A$). Notice also how (due to the $\sigma$ index) there is no need to define a clause for *DelayLoop*.

```agda
cata\text{C}\sigma : \forall \{ i o \} \to \mathbb{C} \{ \sigma \} i o \to T i o
cata\text{C}\sigma (\text{Gate} g) = \text{GateA} g
cata\text{C}\sigma (\text{Plug} f) = \text{PlugA} f
cata\text{C}\sigma (c_1 \_\_ c_2) = \text{cata}\text{C}\sigma c_1 \_\_ \text{cata}\text{C}\sigma c_2
cata\text{C}\sigma (c_1 \_\_\_ c_2) = \text{cata}\text{C}\sigma c_1 \_\_\_ \text{cata}\text{C}\sigma c_2
```

**Listing 8** Catamorphism for combinational circuits.
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 \( \mathbb{C} \) should result in a function of type \( W \to W \). Thus the carrier type for the combinational simulation algebra is:

\[
W \to W : \mathbb{N} \to \mathbb{N} \to \text{Set}
\]

\[
W \to W : m \times n = W \to W
\]

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

\[
\llbracket \_ \rrbracket : \forall \{ i, o \} \to \mathbb{C} \{ \sigma \} \, i \circ o \to W \to W
\]

\[
\llbracket \_ \rrbracket = \text{cata} \mathbb{C} \sigma \text{simulation} \sigma
\]

Notice how the type of the semantic function requires the interpreted circuit to be combinational (it must be indexed by \( \sigma \)). In this way, the algebra used (\text{simulation} \sigma) does not have a field for the \text{DelayLoop} case. We show on Listing 9 the definitions for each of the fields in the algebra (\text{simulation} \sigma)^2.

\[
\text{simulation} \sigma : \mathbb{C} \sigma \text{A} \\
\text{simulation} \sigma = \text{record} \{ \text{Gate} \sigma = \text{spec} \\
\text{Plug} \sigma = \lambda p \, \text{ins} \to \text{tabulate} \left( \text{flip lookup} \, \text{ins} \cdot \text{flip lookup} \, p \right) \\
\llbracket \_ \rrbracket \llbracket \_ \rrbracket = \text{flip} \cdot \text{s' } \cdot \\
\llbracket \_ \rrbracket \llbracket \_ \rrbracket = \lambda f, f' \to \text{uncurry'} \cdot + + \cdot \text{map} \cdot f, f' \cdot \text{splitAt'} \cdot \}
\]

\textbf{Listing 9} Simulation semantics algebra for combinational circuits.

The cases for sequential \( \llbracket \_ \rrbracket \llbracket \_ \rrbracket \) and parallel composition \( \llbracket \_ \rrbracket \llbracket \_ \rrbracket \) rely, respectively, on function composition and \text{map} \, \, \text{over products}. Sequential circuit composition is evaluated simply as flipped composition of the functions obtained from evaluating the subcircuits. For parallel composition, the simulation behaviour is to split the input at the appropriate index, pass each of the parts to the functions obtained from evaluating the subcircuits, and concatenate the results. In the case of fundamental gates, we simply rely on that gate’s specification function. This leaves the most interesting definition to be explained: \text{Plug} \sigma.

In the case of a \text{Plug}, we build the output word pointwise by using \text{tabulate}. The \text{tabulate} function from Agda’s standard library “fills” a \( \text{(Vec} \, a \, n) \) by evaluating a given function of type \( \text{(Fin} \, n \to a) \) on all points of its domain. In our case, each of these points is an output index (element of \( \text{Fin} \, o) \). First, we \text{lookup} the output index in the plug mapping \( p \), obtaining the corresponding input index. Then we use this index to \text{lookup} the input word and place the correct \text{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 \text{NAND} gate \( \overline{\mathbb{C}} \), with the following type and definition:

\[
\text{2 It is useful to note Agda’s convention of adding primes to the names of non-dependent functions (uncurry’, _s’_)}
\]
The gate is described using two-input conjunction (\(\land\)) and one-input negation (\(\lnot\)) as building blocks, and these pieces come from the library of fundamental gates that we are using (BoolTrio). By evaluating \(\top\) we then obtain the following function:

\[
\left\llbracket \top \right\rrbracket : W \rightarrow W \ 2 \ 1
\]

\[
\left\llbracket \top \right\rrbracket = \text{spec } \lnot C' \ast \text{spec } \land C'
\]

To simulate \(\top\) we rely on the specification functions of both gates and on function composition. The names \(\lnot C'\) and \(\land C'\) are just the gate identifiers. If we reduce the expression above further (expanding spec and \(W \rightarrow W\)), we obtain the following:

\[
\left\llbracket \top \right\rrbracket : W \rightarrow W \ 2 \ 1
\]

\[
\left\llbracket \top \right\rrbracket = \lambda \{ (x :: y :: []) \rightarrow \{ \text{not} (x \land y) \} \}
\]

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 is piecewise constant (as well as its output signal). Because of this characteristic, we can model both input and output signals as Streams\(^4\), 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:

\[
\text{shift} : C \{\omega\} \ 1 \ 1
\]

\[
\text{shift} = \text{DelayLoop} \ swap \chi_1
\]

The expected behaviour of shift exemplifies why the type of Atoms (values that can be carried over Π-Ware wires) needs to be not only finite but also inhabited. The shift circuit will put out the value present at its input on the previous clock cycle. On the zeroth clock cycle, however, there is no previous cycle, and thus there must be some default value to be put out.

To discuss the semantics of shift, we first note that the type of shift is tagged by \(\omega\) (omega), and for circuits with \(\omega\) in their type, we must use the sequential simulation semantics \([\left[\_\right]\omega]\). In the specific case of shift, the function obtained via the sequential simulation semantics will have the following type:

\[
[\text{shift}]\omega : \text{Stream} (W \ 1) \rightarrow \text{Stream} (W \ 1)
\]

---

3. By signal we mean a function over the time domain.

4. A stream is an infinite list, i.e., a list without the \texttt{Nil} constructor.
The function obtained via $\llbracket \_ \rrbracket_\omega$ consumes and produces a \textit{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 $\text{Stream} \ a \rightarrow \text{Stream} \ b$) can (possibly) \textit{look into the future}. Considering \textit{Streams} over the discrete time domain, one simple example of stream function that “looks into the future” is $\text{tail}$.

\begin{verbatim}
\text{tail} : \forall \{ \alpha \} \rightarrow \text{Stream} \ \alpha \rightarrow \text{Stream} \ \alpha
\text{tail} (\text{in}_0 :: \text{in}_{1_\text{x}}) = \downarrow \text{in}_{1_\text{x}}
\end{verbatim}

The element at position 0 in the output of $\text{tail}$ 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 \textit{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 \textit{causal step functions}, and they are defined as follows:

\begin{verbatim}
\_ \Rightarrow_c \_ : \forall \{ \ell_1 \ell_2 \} (\alpha : \text{Set} \ \ell_1) (\beta : \text{Set} \ \ell_2) \rightarrow \text{Set} (\ell_1 \uplus \ell_2)
\alpha \Rightarrow_c \beta = \Gamma_c \alpha \rightarrow \beta
\end{verbatim}

The symbol $\Gamma_c$ means \textit{causal context}, and it is defined simply as a non-empty list ($\text{List}^+$), that is, a pair of the head (current value) with a possibly-empty tail (past values).

\begin{verbatim}
\Gamma_c : \forall \{ \ell \} (\alpha : \text{Set} \ \ell) \rightarrow \text{Set} \ \ell
\Gamma_c = \text{List}^+
\end{verbatim}

Coming back to the definition of simulation semantics for sequential circuits, we can now establish the \textit{carrier} type for the algebra of sequential circuits as being a \textit{causal step function} between words of the appropriate length. Then, the \textit{causal} simulation of a circuit is defined as a catamorphism over $\mathbb{C}$ with the $\text{simulation}_c$ algebra:

\begin{verbatim}
W \Rightarrow c W : \forall i o \rightarrow \text{Set}
W \Rightarrow c W i o = W i \Rightarrow c W o
\llbracket \_ \rrbracket_c : \forall \{ i o \} \rightarrow \mathbb{C} i o \rightarrow W \Rightarrow c W i o
\llbracket \_ \rrbracket_c = \text{cat a}_{\mathbb{C}} \{ \sigma \Rightarrow c \text{ simulation}_c \} \text{ simulation}_c
\end{verbatim}

Listing 10 shows the $\text{simulation}_c$ record. Note how the the definitions packed inside of $\text{simulation}_c$ are made available for use by open-ing the corresponding module in the where block.

\begin{verbatim}
\text{simulation}_c : \text{CA} \{ W \rightarrow W \} \{ W \Rightarrow c W \}
\text{simulation}_c = \text{record} \{ \text{GateA} = \lambda g \rightarrow \text{GateA}_\sigma g \ast \text{head} \\
\text{PlugA} = \lambda f \rightarrow \text{PlugA}_\sigma f \ast \text{head} \\
\_ \Rightarrow c \_ = \lambda f_1 f_2 \rightarrow f_2 \ast \text{map}^+ f_1 \ast \text{pasts} \\
\_ \Rightarrow c \_ = \lambda f_1 f_2 \rightarrow \text{uncurry'} \_ \Rightarrow c \_ \Rightarrow c \_ \ast \text{map} \ast \text{unzip}^+ \ast \text{splitAt}^+ \_ \\
\text{DelayLoopA} = \lambda \{ \_ \} \left( \alpha \right) f \rightarrow \text{take}_c \alpha \ast \text{delay} \_ o f \}
\text{where open \text{CA} \text{ simulation}_c using () renaming (\text{GateA} to \text{GateA}_\sigma; \text{PlugA} to \text{PlugA}_\sigma)}
\end{verbatim}

\textbf{Listing 10} Simulation semantics algebra for sequential circuits.
In the case of GateA and PlugA we simply take the present value from the causal context and pass it to the corresponding combinational field (GateAσ and PlugAσ), while the sequence (⟩⟩A_) and parallel (∥A_) cases are a bit more involved.

To understand the ⟩⟩A_ case, first notice that each of the parameters f₁ and f₂ is a causal step function. The pasts function takes the causal context and produces a (non-empty) list of all of its tails: essentially, each element from this list is a causal context viewed from a given moment. We then map⁺ the f₁ function over each of these pasts, producing a list in which each element is the next output considering that given past. Finally, the result of mapping is fed as the causal context of f₂.

The definition for the parallel case (∥A_) is somewhat similar to the combinational one. We also split the input at the appropriate point and use map× to apply f₁ and f₂ to each part of the product. However, splitAt⁺ works pointwise over the causal context, thus the need to use unzip⁺ in order to make the list of pairs into a pair of lists.

Perhaps the most important case in the sequential simulation algebra is DelayLoopA. The delay function transforms the regular word function (which takes l extra wires) into a causal step function, which depends 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:

\[ \text{⟦}_o : \forall \{i:o\} \rightarrow C \rightarrow (\text{Stream} \ i \rightarrow \text{Stream} \ (W \ o)) \]
\[ \text{⟦}_o = \text{runc} \circ \text{⟦}_c \]

In contrast with the type of (⟦_⟧)(the combinational semantics), the type of (⟦_⟧o) 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 (⟦_⟧o), 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 (⟦⟧_{c} and ⟦⟧_{c}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):

\[
\text{test} \text{-} \text{mux} \ 1 \ : \ \text{⟦} \text{mux} \ \text{⟧} (\text{false} :: (\text{true} :: \text{false :: []})) \equiv [ \text{false} ] \\
\text{test} \text{-} \text{mux} \ 1 \ = \ \text{refl} \\
\text{test} \text{-} \text{mux} \ 2 \ : \ \text{⟦} \text{mux} \ \text{⟧} (\text{true} :: (\text{true} :: \text{false :: []})) \equiv [ \text{true} ] \\
\text{test} \text{-} \text{mux} \ 2 \ = \ \text{refl}
\]

Unit testing is useful, but we can do better, thus 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:

\[
\text{⟦} \text{mux} \ \text{⟧} : \text{W}_3 \rightarrow \text{W}_1
\]

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:

\[
\text{ite} : \text{W}_3 \rightarrow \text{W}_1 \\
\text{ite} (s :: a :: b :: []) = [ \text{if} \ s \ \text{then} \ b \ \text{else} \ a ]
\]

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 11.

\[
\text{record} \ \text{Finite} \ \{ \ell \} \ (\alpha : \text{Set } \ell) : \text{Set } \ell \ \text{where} \\
\text{field} \ \#\alpha : \mathbb{N} \\
\text{mapping} : \alpha \leftrightarrow \text{Fin } \#\alpha \\
\text{open Inverse'} \ \text{mapping public}
\]

**Listing 11** 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 \text{⟦} \text{mux} \text{⟧} 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: this vector will contain proofs \{P(x_1), P(x_2), \ldots, P(x_n)\} for all the elements \(x_n\) of a type \(\alpha\) given a certain predicate \(P : \alpha \rightarrow \text{Set}\).

By using its bijection with Fin n, we can have a \(\forall\)-introduction rule for any Finite type.
The $\forall$–Finite function takes an implicit parameter $(ps)$ containing the aforementioned heterogeneous vector of proofs. This parameter can be implicit because each of the vector’s elements reduces to $(\text{tt} : \text{T})$ (if the predicate holds), and the whole “vector” reduces to a nested pair of units. Agda’s definition of both pairs and the unit type as a record, combined with the η-rule for records, ensure that the value of $ps$ can be guessed.

Now, our final goal is to have exhaustive checking of circuit behaviour, so we need a $\forall$-introduction rule for the type of circuit inputs and outputs – Vec Atom $n$, abbreviated as $W$ $n$. We know that $\text{Vec} \; \alpha$ $n$ is Finite whenever $\alpha$ also is (proof omitted here), and combining this knowledge with the previous definition of $\forall$–Finite we arrive at the desired $\forall$–W rule ($\forall$-introduction for words):

$$\forall$–W : $\forall$ \{ $\{ $\forall$ \{ $\forall$–W \}$ \}
\rightarrow ($\forall$ \{ $\forall$–W \} $\rightarrow P$ $w$)

In particular, we can then simply use $\forall$–W to prove properties involving all possible inputs of a circuit. The property in which we are interested is whether a circuit and a given specification function agree on a certain input.

$$\check\equiv?_\text{at}_\text{w} : \forall$ \{ $\forall$ \{ $\check\equiv?_\text{at}_\text{w}$ \} \}
\rightarrow ($\check\equiv?_\text{at}_\text{w}$ $\rightarrow \text{Set}$)

The statement $\check\equiv?_\text{at}_\text{w}$ $f$ $w$ can be read as “$c$ complies with $f$ at input $w$”. This relation relies on a decidable equality over output words of the checked circuit $(\check\equiv?_\text{w})$, and uses it to compare the results obtained by running the circuit simulation and the specification function. In the definition of $\check\equiv?_\text{at}_\text{w}$, the call to $[\_]$ serves only to transform the value returned by $\check\equiv?_\text{w}$ into a $\text{Bool}$, which is then further transformed by $\text{T}$ into a $\text{Set}$ (either $\text{T}$ or $\perp$).

To put the last pieces of the puzzle together we call $\forall$–W, passing the relation just defined (partially applied) as the predicate to be exhaustively checked. In this way we obtain the function we ultimately wanted: $\text{check}\equiv?$.

$$\check\equiv? : \forall$ \{ $\forall$ \{ $\check\equiv? \}$ \}
\rightarrow ($\check\equiv? \rightarrow \text{Set}$)

This function performs automatic exhaustive checking, in order to verify that a circuit complies with a given specification function. It is feasible to use $\text{check}\equiv?$ 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 [2] EDSL, but the idea of parameterized definitions is at least as old as VHDLC’s generics [14]. 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 \texttt{muxN}, that selects between two inputs of size \(n\) each, has the following type and definition:

\[
\texttt{muxN} : \forall n \{s\} \rightarrow \mathbb{C} \{s\} ((1 + (n + n)) n) \\
\texttt{muxN zero} = \texttt{nilX} \\
\texttt{muxN (suc n)} = \texttt{adaptX} n \parallel \texttt{mux || muxN n}
\]

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 (\texttt{nilX}). In the recursive case, we connect the 2-input \texttt{mux} and the recursive call (\texttt{muxN n}) in parallel, and we need a \texttt{Plug} (called \texttt{adaptX}) to make the right wires meet the right ports.

\[
\texttt{adaptX} : \forall n \{s\} \rightarrow \mathbb{C} \{s\} ((1 + ((1 + n) + (1 + n))) ((1 + 1 + 1) + (1 + (n + n))))
\]

The first 3 bits in the output size of \texttt{adaptX} \(1 + 1 + 1\) are exactly those needed by the \texttt{mux}, while the remaining ones are consumed by \texttt{muxN n}. The diagram on Figure 2 shows exactly how \texttt{adaptX} forks the selection bit and rearranges the remaining wires appropriately.

![Figure 2 Architecture of the adaptX plug.](image)

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 \texttt{muxN}, the specification is \texttt{iteN}:

\[
\texttt{iteN} : \forall n \rightarrow W (1 + (n + n)) \rightarrow W n \\
\texttt{iteN zero} _ = [] \\
\texttt{iteN (suc n)} (_ : \texttt{ab}) = \texttt{splitAt (suc n) ab} \\
\texttt{iteN (suc n)} (s : : (a ++ b)) | a , b , \texttt{refl} = \texttt{if s then b else a}
\]

In \texttt{iteN}, the tail of the input is split into two equal parts and we use \texttt{if\_then\_else\_} to choose (based on the selection bit), which of the two parts will be the output.

The functional correctness property that we are interested in is the \textit{pointwise equality} of the specification function for a circuit and the function obtained via the simulation semantics, that is, both functions have to agree on all inputs. This relation is abbreviated with the name \_\_\_\_\_, where a statement \(c \subseteq f\) can be read as “\(c\) complies with \(f\)”.

In the case of our \texttt{muxN} example, the proof of compliance will need to follow the same induction pattern used to define the specification (\texttt{iteN}) itself. Namely, we need to pattern match on \(n\) and do case analysis on the result of splitting the tail of the input.
\[
\begin{align*}
\text{muxN} \sqsubseteq \text{iteN} : \forall n \to \text{muxN} n \sqsubseteq \text{iteN} n \\
\text{muxN} \sqsubseteq \text{iteN} \text{zero} & \quad (_{\_ :: []}) = \text{refl} \\
\text{muxN} \sqsubseteq \text{iteN} \text{ succ} n & \quad (_{\_ :: ab}) \quad \text{with splitAt} \ (\text{succ} n) \ ab \\
\text{muxN} \sqsubseteq \text{iteN} \text{ succ} n & \quad (s :: (a ++ b)) \ | \ a, b, \text{refl} = \text{muxN} \sqsubseteq \text{iteN}'
\end{align*}
\]

Unfortunately, the full proof of \( \text{muxN} \sqsubseteq \text{iteN} \) is a bit too long to be completely analyzed here (we abbreviate at \( \text{muxN} \sqsubseteq \text{iteN}' \)). The proof relies of course on the proof of correctness for \( \text{mux} \) (the basic circuit with type \( \mathbb{C}_3 \)). It also relies on properties of the \( \text{adaptX} \) 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\(^5\). Connection patterns are just functions taking circuits as inputs and producing circuits as outputs. Typically they use the constructors of \( \mathbb{C} \) to connect their arguments in a certain fashion, thus the name.

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

\[
\begin{align*}
\text{parsN} : \forall \{k \ i o s\} \to \mathbb{C} \{s\} \ i o \to \mathbb{C} \{s\} (k \ast i) (k \ast o) \\
\text{parsN} (k) (i) (o) c = \text{subst}_2 \mathbb{C} (\ast \text{--sum--replicate} \ k \ i) (\ast \text{--sum--replicate} \ k \ o) \\
& \quad \text{(pars (replicateI}_{2} \{n = k\} \ c))
\end{align*}
\]

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 (\( \text{parsN} \)) is a special case of a more general pattern: instead of \( \text{replicating} \) the same circuit \( n \) times, we can connect a whole vector of (different) circuits in parallel. This is achieved by \( \text{pars} \):

\[
\begin{align*}
\text{pars} : \forall \{n \ i o s\} \{\text{is os} : \text{VecI}_{2} \ \mathbb{C} \{s\} \ i o\} \to \mathbb{C} \{s\} \ (\text{sum is}) (\text{sum os}) \\
\text{pars \{is = []\} \{[]\} \{[]\} _2 \ = \text{nil}X} \\
\text{pars \{is = _{_{::_}}\} \{_{::_}\} \{c :: _2 \ cs\} \ = c \parallel \text{pars cs}
\end{align*}
\]

As the parameter of \( \text{pars} \) 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 \( \text{VecI}_{2} \), what we call an index-heterogeneous vector\(^6\). 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 \( \text{seqsN} \), taking a circuit and connecting \( n \) copies of it in sequence:

\[
\begin{align*}
\text{seqsN} : \forall k \ {s \ io} \to \mathbb{C} \{s\} \ io \ io \to \mathbb{C} \{s\} \ io \ io \\
\text{seqsN} k = \text{seqs} \ast \text{replicate} \ \{n = k\}
\end{align*}
\]

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 \( \text{seqsN} \) 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

---

\(^5\) This name comes from Lava as well.

\(^6\) More specifically, \( \text{VecI}_{2} \) only handles type constructors with two indices.
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 [19] to a dependently-typed setting.

We are currently working on establishing lemmas 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:

$$\forall \ w \rightarrow \ \llbracket \ \text{seqsN} \ k \ c \ \rrbracket \ w \equiv (\text{iterate} \ k \ \llbracket \ c \ \rrbracket) \ w$$

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 $\equiv c'$”, 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.

$$\_\equiv\_ : \forall \ \{i \ a\} \ (c_1, c_2 : \ C \ i \ a) \rightarrow \text{Set}$$

$$c_1 \equiv c_2 = \forall \ w \rightarrow \ \llbracket \ c_1 \ \rrbracket \ w \equiv \ \llbracket \ c_2 \ \rrbracket \ w$$

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 : C 1 n) with (c_2 : 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:

$$\_\approx\_ : \forall \ \{i \ a_1, a_2\} \rightarrow \text{C} \ i \ a_1 \rightarrow \text{C} \ i \ a_2 \rightarrow \text{Set}$$

$$c_1 \approx c_2 = \forall \ w \rightarrow \ \llbracket \ c_1 \ \rrbracket \ w \equiv \ \llbracket \ c_2 \ \rrbracket \ w$$

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 : C n 1) with (c_2 : 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:

$$\_\approx\_ : \forall \ \{i_1 \ a_1, i_2 \ a_2\} \rightarrow \text{C} \ i_1 \ a_1 \rightarrow \text{C} \ i_2 \ a_2 \rightarrow \text{Set}$$

$$\_\approx\_ \ {\{i_1\}} \ {\{i_2\}} \ c_1 \ c_2 =$$

$$\forall \ \{w_1 : W i_1\} \ {\{w_2 : W i_2\}} \rightarrow \ w_1 \approx w_2 \rightarrow \ \llbracket \ c_1 \ \rrbracket \ w_1 \approx \ \llbracket \ c_2 \ \rrbracket \ w_2$$

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 $\approx$ c_2) simply by making sure the hypothesis is false. A simple example of a term that should be banned by $\_\approx\_$ but is allowed is the following:
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.

Besides the requirement over the sizes of input words, we also implement this relation as a data-type (instead of a Set-valued function). This allows us to pattern-match on the arguments of the refl≋ constructor, which is needed when proving several lemmas related to _≋_ (for example, symmetry and transitivity).

With _≋_, we arrive at the definition of circuit equivalence used 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 (C, _≋_) 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 13 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 \([x_0, (x_0 \oplus x_1), (x_0 \oplus x_1 \oplus x_2), ..., (x_0 \oplus ... \oplus x_n)]\).

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 3 we show an example of a circuit utilizing maximal parallelism to calculate a scan with 8 inputs.

![Figure 3](image-url) Example of an 8-input parallel prefix circuit.

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” [12]. 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 [12] match exactly those present in Π-Ware, amongst them sequential (\( \parallel \parallel \)) and parallel composition (\( \parallel \square \)) along with the identity plug (\( \text{id} \square \)). 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 (\( \parallel \parallel \)) forms a monoid of circuits, with \( \text{id} \square \) as identity:

\[
\parallel \parallel \text{left-identity} : \forall \{ i o \} (c : \mathbb{C} i o) \rightarrow \text{id} \square c \equiv c
\]

\[
\parallel \parallel \text{left-identity} c = \equiv \equiv \left( \text{from-\equiv} \circ \cong \right) \left( c \parallel \text{id} \square \text{id} \right)
\]

\[
\parallel \parallel \text{assoc} : \forall \{ i m n o \} (c_1 : \mathbb{C} i m) (c_2 : \mathbb{C} m n) (c_3 : \mathbb{C} n o) \rightarrow (c_1 \parallel c_2) \parallel c_3 \equiv c_1 \parallel (c_2 \parallel c_3)
\]

\[
\parallel \parallel \text{assoc} c_1 c_2 c_3 = \equiv \equiv \left( \text{from-\equiv} \circ \lambda \rightarrow \text{refl} \right)
\]

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 \( \text{id} \square \) 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:

\[
\begin{align*}
\text{scan} & : \forall n \rightarrow \mathbb{C} n n \\
\text{scan zero} & = \text{id} \square 0 \\
\text{scan} (\text{suc} n) & = \text{id} \square 1 \parallel \text{scan} n \parallel \text{fan} (\text{suc} n)
\end{align*}
\]

Besides the parts already mentioned (\( \parallel \parallel \), \( \parallel \square \), \( \text{id} \square \)), here we also use the \( \text{fan} \) primitive. A term “\( \text{fan} n \)” has type \( \mathbb{C} n n \), and calculates \([x_0, (x_0 \oplus x_1), (x_0 \oplus x_2), ..., (x_0 \oplus x_n)]\). The diagram in Figure 4 illustrates the structure of “scan 3”.

![Figure 4](structure_of_prototype_scan.png)

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 \( \parallel \parallel \square \square \)) and the **parallel scan combinator** (called \( \parallel \square \square \)). 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 \( \text{fan} \) connects the last output of the first argument to all inputs of the second.

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

\[
\begin{align*}
\square \square \text{law} & : \text{scan} (\text{suc} m) \square \square \text{scan} (\text{suc} n) \equiv \text{scan} (m + \text{suc} n) \\
\parallel \parallel \text{law} & : \text{scan} (\text{suc} m) \parallel \parallel \text{scan} n \equiv \text{scan} (m + n)
\end{align*}
\]
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 5.

![Figure 5](structure-of-a-serial-scan.png)  

**Figure 5** Structure of a serial scan.

The Pi-Ware description for serial is pretty simple and makes essential use of the parallel scan combinator (\(\parallel\)). The code for serial is shown in Listing 12.

```
serial : \forall n \to \mathbb{C} n n
serial zero = \text{id} \times 0
serial (suc zero) = \text{id} \times 1
serial (suc (suc n)) = serial (suc n) \parallel \text{id} \times 1
```

**Listing 12** Pi-Ware description of a serial scan.

Finally, we prove that serial does indeed compute a scan. The proof (Listing 13) relies on the key fact that \(\parallel\) preserves scans. Furthermore, it relies on the fact that \(\parallel\) 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 : \forall n \to serial n \equiv scan n
serial-is-scan zero = \equiv-refl
serial-is-scan (suc zero) = id \times 1 \equiv scan 1
serial-is-scan (suc (suc n)) = begin
  serial (suc (suc n))
\equiv (\equiv-refl definition of serial (suc (suc n)))
  serial (suc n) \parallel id \times 1
\equiv (serial-is-scan (suc n) \parallel-cong id \times 1 \equiv scan 1)
  scan (suc n) \parallel id \times 1
\equiv (\parallel-law n \equiv scan (suc n + 1))
\equiv (scan-cong (cong suc (+--comm n 1)))
\equiv (scan (suc (suc n)))
```

**Listing 13** 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 [7].

Systems such as ACL2 have been used to prove correctness of entire microprocessors [13], 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 [2], Hawk [16], Wired [1] and ForSyDe [20]; Sheeran [21] gives an excellent overview. When comparing Π-Ware to these other DSLs, the overarching theme is the use by Π-Ware of a host language in which stronger static guarantees are provided and in which circuits and proofs live side-by-side.

Specifically when comparing Π-Ware and Lava, we can say:

- Lava uses observational sharing as a binding technique, while Π-Ware is combinator-based.
- Lava verification uses external tools, while Π-ware circuits and proofs share a language.
- In Lava, only specific instances of a circuit generator can be verified, while Π-ware exploits the inductive structure of generators for verifying the generators themselves.

ForSyDe is still closely related to Π-Ware in terms of its goals, but much less closely than Lava. It offers similar verification facilities to those of Lava (using external model checking tools), so the same comparison applies. However, ForSyDe offers a different “front-end” to the user, by using reflection features of Haskell (namely quasi-quoters). Another considerable restriction on ForSyDe is that the set of types supported as inputs and outputs of circuits is very restricted (booleans, sequences, tuples), and cannot easily be extended, while Π-Ware is designed to be as general as possible regarding these choices.

With regards to Π-Ware and Wired, both languages describe the architecture of circuits, but while Wired definitions specify the exact geometry and placement of components, Π-Ware only talks about topology. We recognize the usefulness of geometric descriptions, but we have chosen to focus on topology as our initial goal is to prove properties involving (the preservation of) functional correctness and circuit transformations with a space vs. time trade-off.

When looking at the literature for hardware verification methods, we note that the idea of using dependent types for circuit description is not new and can be traced back as far as Hanna [11]. The paper “Constructing Correct Circuits” [4] 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 [5] and Fe-Si [6]) 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 [23, 15]. Furthermore, we can use Agda features such as goal and context reflection, as well as solvers for algebraic structures (monoids, semirings, etc.) [3].

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:

\[
\text{assoc} : \forall \{a \ b \ c\} \rightarrow ((a + b) + c) \cong (a + (b + c))
\]

\[
\text{assoc} \{a\} \{b\} \{c\} = \text{eq} \left(\leftrightarrow \text{assoc} \ a \ b \ c\right)
\]

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 [8]. 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 "\( C (\text{Bool} \times (\text{Bool} \times \text{Bool})) \) \text{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 14).

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

Listing 14 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.

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


