Verified Timing Transformations in Synchronous Circuits with λπ-Ware

João Paulo Pizani Flor, Wouter Swierstra

Utrecht University
{J.P.PizaniFlor,W.S.Swierstra}@uu.nl

Abstract. We define a DSL for hardware description, called λπ-Ware, embedded in the dependently-typed language Agda, which makes the DSL well-scoped and well-typed by construction. Other advantages of dependent types are that circuit models can be simulated and verified in the same language, and properties can be proven not only of specific circuits, but of circuit generators describing (infinite) families of circuits. This paper focuses on the relations between circuits computing the same values, but with different levels of statefulness. We define common recursion schemes, in combinational and sequential versions, and express known circuits using these recursion patterns. Finally, we define a notion of convertibility between circuits with different levels of statefulness, and prove the core convertibility property between the combinational and sequential versions of our vector iteration primitive. Circuits defined using the recursion schemes can thus have different architectures with a guarantee of functional equivalence up to timing.

1 Introduction

Modelling electronic circuits has been a fertile ground for functional programming (Sheeran, 2005) and theorem proving (Hanna and Daeche, 1992). There have been numerous efforts to describe, simulate, and verify circuits using functional languages such as MuFP (Sheeran, 1984) and more recently CλaSH (Baaij, 2015) and ForSyDe (Sander and Jantsch, 2004).

Functional languages have also been used to *host* an Embedded Domain-Specific Language (EDSL) for hardware description. Some of these EDSLs, such as Wired (Axelsson et al., 2005), capture low-level information about the layout of a circuit; others aim to use the host language to provide a higher-level of abstraction to describe the circuit's intended behaviour. A notable example of the latter approach is Lava (Bjesse et al., 1999) and its several variants (Gill et al., 2009; Singh, 2004).

Also interactive theorem proving and programming with dependent types have been fruitfully used to support hardware verification efforts, with some based on HOL (Melham, 1993; Boulton et al., 1992), some on Coq (Braibant, 2011; Braibant and Chlipala, 2013) and some on Martin-Löf Type Theory (Brady et al., 2007) Following this line of research, we utilize a dependently-typed programming language (Agda) as the *host* of our hardware EDSL, for its proving capabilities and convenience of embedding.

In particular, this paper focuses on verification related to *timing*, that is, the behaviour of a circuit in terms of its inputs *over time*. When designing hardware, a compromise must be made between the *area* occupied by a circuit and the *number of clock cycles* it takes to produce its results.

A *combinational* (stateless) architecture better harnesses potential parallelism but might negatively influence other constraints such as frequency and power consumption. A more *sequential* circuit (stateful), on the other hand, will occupy less area but might be a bottleneck in computational throughput and impact other parts of the design that depend on its outputs.

There are many different ways to implement any specific functional behaviour, and it can be difficult to find the right spot in the design space upfront. Timing-related circuit transformations are quite invasive and error-prone – making it difficult to correct bad design decisions *a posteriori*. With this paper, we attenuate some of these issues by defining a language for circuit description that facilitates the exploration of different points in the timing design space. More concretely, this paper makes the following contributions:

- We show how to embed a typed hardware DSL, λπ-Ware, in the general purpose dependently typed programming language Agda (Section 3), together with an executable semantics based on state transitions (Section 4).
- Next, we define common recursion patterns to build circuits in both combinational and sequential architectures (Section 5). We show how some well-known circuits can be expressed in terms of these recursion patterns.
- Finally, we define a precise relation between the combinational and sequential versions of circuits that *exhibit equivalent behaviour* (Section 5.1). By proving that different versions of our recursion schemes are convertible, we allow hardware designers to enable different levels of parallelism while being certain that semantics are being preserved up to timing.

Altogether, these contributions help to *separate the concerns* between the *values* a circuit must produce and the *timing* with which they are produced. In this way, timing decisions can more easily be modified later in the design process.

The codebase in which the ideas exposed in this paper are developed is available online.[1] For the sake of presentation, code excerpts in this paper may differ slightly from the corresponding ones in the repository.

2 Overview

We begin by shortly demonstrating the usage of λπ-Ware. Although inspired by our previous work (*Π*-Ware (Pizani Flor et al., 2016)), λπ-Ware uses variable binding for sharing and loops, instead of pointfree combinators. Furthermore, λπ-Ware has a universe of (simply-)structured types, whereas the types of *Π*-Ware were vectors only. In this section, we illustrate the language by means of two variations on a simple circuit. Later sections cover the syntax and semantics of λπ-Ware in greater detail.

[1] `https://gitlab.com/joaopizani/lambda1-hdl/tree/paper-2017-comb-seq`

Example: Horner's method We look at two circuits for calculating the value of a polynomial at a given point, one with a combinational architecture and another sequential, both based on Horner's method.

For any coefficients $a_0, \ldots, a_n$ in $\mathbb{N}$, we can define a polynomial as follows:

$$p(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n,$$

In order to compute the value of the polynomial at a specific point x_0 of its domain, Horner's method proceeds by using the following sequence of values:

$$\begin{aligned} b_n &:= a_n \\ b_{n-1} &:= a_{n-1} + b_n x_0 \\ &\vdots \\ b_0 &:= a_0 + b_1 x_0. \end{aligned}$$

Then b_0 is the value of of our polynomial at x_0, that is, $p(x_0)$. By iteratively expanding definitions for each of the b_i in the equations above, one arrives at a factorized form of the polynomial clearly equivalent to the usual series of powers.

Combinational version Horner's method is easily expressed as a *fold*, and in λπ-Ware we can build a combinational (stateless) circuit to compute this fold, for any given degree n. When reading the signature of the horner-comb definition below, one must note that *only* the parameters with the type former λH are circuit inputs, and the others are synthesis parameters.

```
horner-comb : ∀ n (x₀ : λH ℕ) (aₙ : λH ℕ) (as : Vec (λH ℕ) n) → λH ℕ
horner-comb x₀ = foldl-comb (λ s a → a :+: x₀ :*: s)
```

This circuit computes the value of a polynomial of degree n at a given point. It has three inputs: the point at which to evaluate the polynomial (x_0), the coefficient of highest degree (a_n) and the remaining coefficients (as). Later in Section 4 we present the detailed semantics of circuits, but for now we can say that horner-comb n behaves similarly to foldl from Agda's standard library.

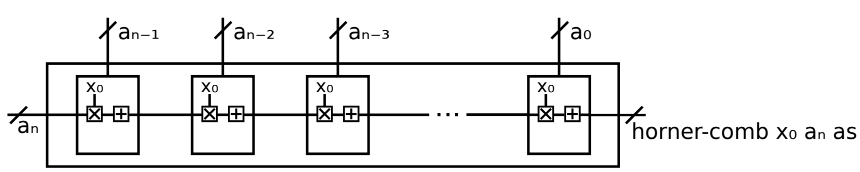

Fig. 1. Block diagram of the horner-comb circuit.

Figure 1 shows the architecture of horner-comb, where we can clearly see that the circuit contains no loops nor memory cells and that the *body* of the foldl is *replicated* n times. In the horner-comb model, area is linearly proportional to the degree of the polynomial, and if we want to reduce area occupation, we need to introduce *state* into the picture somehow.

Sequential version Next, we describe a fully *sequential* circuit to do the same calculation, using internal state to produce a *sequence of outputs*. With this architecture the area is constant (independent of the degree of the polynomial). The output value of the circuit at clock cycle i corresponds to the sum of all polynomial terms with degree smaller than or equal to i, evaluated at point x_0.

```
horner-seq : ∀ (x0 : λH ℕ) (a : λH ℕ) → λH ℕ
horner-seq x0 = foldl-seq (λ s a → a :+: x0 :*: s)
```

The circuit takes two inputs: x_0, the point at which we desire to evaluate the polynomial; and a, a *single* input containing the n- (i+1)-th coefficient at the i-th clock cycle. The circuit is defined using the foldl-seq combinator, that iterates its argument function. This function corresponds to the loop body, mapping the current approximation, s, and the current value of the input a to a new approximation. As we shall see, to execute this *sequential* circuit, we will need to provide an initial value for the state, s.

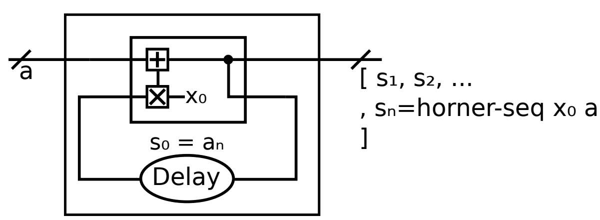

Fig. 2. Block diagram of the horner-seq circuit.

Figure 2 shows the architecture of horner-seq, where we see that the body of the foldl is the same as in the combinational version. But now instead of n instances of the body we have a single instance, with one of its outputs *tied back* in a loop with a memory cell (shift register).

We have seen that the combinational and sequential definitions are syntactically similar, but have very different timing behaviour and generate very different architectures. First of all, the coefficients input of horner-comb is a vector (a *bus* in hardware parlance), while the corresponding input of horner-seq is a single number. Also, all the coefficients are consumed by horner-comb in a single clock cycle, while horner-seq consumes the sequence of coefficients over n clock cycles. It is only after these n cycles that the results of the two circuits will coincide.

3 λπ-Ware

We begin by fixing the universe of types, U, for the elements that circuits may produce or consume. This type is parameterized by the type of data carried over the circuit's wires (B). A typical choice of B would be bits or booleans, with other choices possible when modelling a higher-level circuit, such as integers or a datatype representing assembly instructions for a microprocessor.

```
data U (B : Set) : Set where
  unit  : U B
  ι     : U B
  _⇒_  : (σ τ : U B) → U B
  _⊗_  : (σ τ : U B) → U B
  _⊕_  : (σ τ : U B) → U B
  vec   : (τ : U B) (n : ℕ) → U B
```

The collection of type codes consists of a unit ($\mathbb{1}$) and base (ι) types, closed under function space (⇒), products (⊗), coproducts (⊕) and homogeneous arrays of fixed size (vec). Each element of U B is mapped to the corresponding Agda type, in particular the code ι is mapped to B, the base type in our type universe.

Core datatype As mentioned before, our language is a deep-embedding in Agda, and circuits are elements of the λB datatype. Let us start by discussing the most fundamental constructors of λB, shown below. Additional constructors are discussed further ahead.

```
data λB : (Γ : Ctxt B) (τ : U B) → Set where
  ⟨_⟩  : (g : Gate τ) → λB Γ τ
  var  : (i : Γ ∋ τ)   → λB Γ τ
  _$_  : (f : λB Γ (σ ⇒ τ)) (x : λB Γ σ) → λB Γ τ
  let′  : (x : λB Γ σ) (b : λB (σ :: Γ) τ) → λB Γ τ
  loop : (c : λB (σ :: Γ) (σ ⊗ τ))          → λB Γ τ
```

We use *typed De Bruijn* indices for variable binding, however, there is a convenience layer on top of λB, called λH, as seen in the overview section. Definitions using λH are essentially a shallow embedding of circuits into Agda (using Higher-Order Abstract Syntax (HOAS)), offering a more convenient programming interface by having named variables. The *unembedding* technique (Atkey et al., 2009) guarantees that it is always possible to go from a circuit definition using λH to an equivalent one using λB.

Returning to the λB datatype itself, it is indexed by a context (Γ : Ctxt B) representing the arguments to the circuit or any free variables currently in scope. The datatype is also indexed by the circuit's output type, (τ : U B).

The whole development is parameterized by a type of primitive gates, Gate : U B → Set, and the ⟨_⟩ constructor creates a circuit from such a fundamental gate. One example of such type of gates is the usual triple ($\{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}\}$)

with Bool as the chosen base type; circuit designers, however, are free to choose the fundamental gates that best fit their domain.

Our language does have an eliminator (_$_) for arrow types, but no introduction form. Arrow types can only be introduced by using gates, and this is by design, as we target synthesizability and circuits must be first-order to be synthesized. Using arrow types for gates allows for convenient partial application, while for general abstraction we use host language definitions as metaprograms.

While the constructors shown above form the heart of the λB datatype, there are also constructors for products, coproducts and vectors:

```
_,_                 : λB Γ τ₁  →  λB Γ τ₂ → λB Γ (τ₁ ⊗ τ₂)
case⊗_of_           : λB Γ (σ₁ ⊗ σ₂)  →  λB (σ₁ :: σ₂ :: Γ) τ → λB Γ τ
inl                 : λB Γ τ₁ → λB Γ (τ₁ ⊕ τ₂)
inr                 : λB Γ τ₂ → λB Γ (τ₁ ⊕ τ₂)
case⊕_either_or_    : λB Γ (σ₁ ⊕ σ₂)  →  λB (σ₁ :: Γ) τ  →  λB (σ₂ :: Γ) τ
                    → λB Γ τ
nil                 : λB Γ (vec τ zero)
cons                : λB Γ τ → λB Γ (vec τ n) → λB Γ (vec τ (suc n))
mapAccumL-comb      : λB (σ :: ρ :: Γ) (σ ⊗ τ)  →  λB Γ σ  →  λB Γ (vec ρ n)
                    → λB Γ (σ ⊗ vec τ n)
```

We give the elimination forms for both products and coproducts uniformly as case constructs, instead of projections that matches on its argument and introduces newly bound variables to the context. For vectors, λB has the two usual introduction forms: one to produce an empty vector of any type (nil) and to extend an existing vector with a new element (cons). Finally, the *accumulating map*, mapAccumL-comb, performs a combination of map and foldl: The input vector with elements of type ρ is pointwise transformed into one with elements of type τ, all the while threading an accumulating parameter of type σ from left to right.

This eliminator is less general than the usual type theoretic elimination principle for vectors; embedding this more general eliminator would require dependent types and higher-order functions in our circuit language. To keep our object language simple, however, we chose a more simple elimination principle capable of expressing the most common hardware constructs.

4 Semantics and properties

Where the previous section defined the *syntax* of our circuit language, we now turn our attention to its *semantics*. Although there are many different interpretations that we could assign to our circuits, for the purpose of this paper we will focus on describing a circuit's input/output behaviour.

State transition semantics Circuits defined in λB can be classified in two ways. *Combinational* circuits do not have any loops; *sequential* circuits may contain loops. To define the semantics of sequential circuits, we will need to define the type of state associated with a particular circuit. To do so, we define the inductive family λs:

```
data λs : (c : λB Γ τ) → Set where
  _s,_  : (sx : λs x) (sy : λs y)      → λs (x , y)
  sLoop : {c : λB (σ :: Γ) (σ ⊗ τ)} → (si : El σ) → (sc : λs c) → λs (loop c)
  ...
```

This family has a constructor for each constructor of λB. Most of these constructors either contain no significant information, or simply follow the structure of the circuit, like in the clause for pairs, _s,_, shown above. The most interesting case is sLoop, in which the state required to simulate a circuit of the form loop c consists of a value of type El σ — where σ is the type of the state that the circuit produces — together with any additional state that may arise from the loop body.

One other constructor of λs deserves special attention: sMapAccumL-comb. A circuit built with mapAccumL-comb consists of n *copies* of a subcircuit f connected in a row. Hence, the state of such a circuit consists of a *vector* of states, one for each of the copies of f. Correspondingly, we define the state associated with such an accumulating map as follows:

```
sMapAccumL-comb : (sf : Vec (λs f) n) (se : λs e) (sxs : λs xs)
                → λs (mapAccumL-comb f e xs)
```

With this definition of state in place, we turn our attention to the semantics of our circuits. We will sketch the definition of our single step semantics, ⟦_|⟧s, mapping a circuit, initial state and environment to a new state and the value produced by the circuit.

```
⟦_|⟧s : (c : λB Γ τ) (m : λs c) (γ : Env El Γ) → λs c × El τ
```

The environment γ assigns values to any free variables in our circuit definition. The base cases for our semantics are as follows:

```
⟦ ⟨ g ⟩ |⟧s m γ  =  m , (⟦–⟧g g)
⟦ var i |⟧s m γ  =  m , lookup i γ
```

In the case for gates, we apply the semantics of our atomic gates, described by the auxiliary function ⟦–⟧g; in the case for variables, we lookup the corresponding value from the environment. Both these cases do not refer to the circuit's state. This state becomes important when simulating loops. In the clauses for application, let′ and loop, shown in Listing 1, we do need to consider the circuit's state.

In the cases of application and **let**, each subcircuit simply "takes a step" independently and the next state of the whole circuit is a combination of the next states of each subcircuit. The case for loop is slightly more interesting: the

```
⟦ f $ x |⟧s (mf s$ mx) γ  =  let (mx′ , rx)  =  ⟦ x |⟧s mx γ
                                 (mf′ , rf)   =  ⟦ f |⟧s mf γ
                             in ((mf′ s$ mx′) , (rf rx))
⟦ let′ x b |⟧s (sLet mx mb) γ  =  let (mx′ , rx)  =  ⟦ x |⟧s mx γ
                                      (mb′ , rb)  =  ⟦ b |⟧s mb (rx :: γ)
                                  in ((sLet mx′ mb′) , rb)
⟦ loop f |⟧s (sLoop ml mf) γ  =  let (mf′ , (ml′ , rl))  =  ⟦ f |⟧s mf (ml :: γ)
                                 in  ((sLoop ml′ mf′) , rl)
```

Listing 1: State-combining clauses of the single-step state transition semantics.

loop body,f, takes an additional input, namely the current state given by the ml parameter of sLoop constructor.

The further clauses of the transition function handle the introduction and elimination forms of products, coproducts and vectors. They are all defined simply by recursive evaluation of the subcircuits, and are straightforward enough to omit from the presentation here. For example, the clause for coproduct elimination is shown below:

```
⟦ case⊕ x∨y either f or g |⟧s (sCase⊕ mxy mf mg) γ  =
   let (mxy′ , rx∨ry)  =  ⟦ x∨y |⟧s mxy γ
   in  [ map× (flip (sCase⊕ mxy′) mg) id ∘ (⟦ f |⟧s1 mf γ)
       , map× (      (sCase⊕ mxy′) mf)  id ∘ (⟦ g |⟧s1 mg γ)
       ] rx∨ry
```

First the coproduct value (x∨y) is evaluated, computing a result value and its next state. The result of the evaluation (rx∨ry) is then fed to Agda's coproduct eliminator ([_,_]); the functions that process the left and right injections proceed accordingly. In either case, the value is fed into evaluation of the appropriate body (either f or g), and the result is then used as the result of the whole coproduct evaluation.

Similarly our elimination principle for vectors, mapAccumL-comb, is worth highlighting:

```
⟦ mapAccumL-comb f e xs |⟧s (sMapAccumL-comb mfs me mxs) γ  =
   let (me′ , re)          =  ⟦ e |⟧s me γ
       (mxs′ , rxs)        =  ⟦ xs |⟧s mxs γ
       (rz , mfs′ , rys)  =  mapAccumL2 (transformF ⟦ f |⟧s2 γ) re mfs rxs
   in (sMapAccumL-comb mfs′ me′ mxs′ , (rz , rys))
```

The above clause is key in the relation that we later establish (Section 5.1) between combinational and sequential versions of circuits. The three key substeps involved in this clause are: evaluation of the left identity element (e), the evaluation of the row of inputs (xs) and the row of step function copies (f).

The first two steps are as expected: both the identity and row of inputs take a step, and we thus obtain the next state and result values of each. The core step is then evaluating the row of copies of f, and its semantics are given using the auxiliary function mapAccumL2.

The mapAccumL2 function is simply a two-input version of an accumulating map, which works by simply zipping the pair of input vectors and calling the mapAccumL function from Agda's standard library.

```
mapAccumL : (σ → α → (σ × β)) → σ → Vec α n → σ × Vec β n
mapAccumL f s []         = s , []
mapAccumL f s (x :: xs)  = let s′ , y   = f s x
                               s″ , ys  = mapAccumL f s′ xs
                           in s″ , (y :: ys)
mapAccumL2 : (σ → α → γ → (σ × β × δ)) → σ → Vec α n → Vec γ n
             → σ × Vec β n × Vec δ n
mapAccumL2 f s xs ys
             = map× id unzip $ mapAccumL (uncurry ∘ f) s (zip xs ys)
```

In the semantics of mapAccumL-comb, we apply mapAccumL2 to the vector with the result of xs (called rxs) as well as the vector with states for the copies of f (called mfs). Then, as the result of the application we obtain the final accumulator value and vector of result values, *together with the vector of next state values* (mfs′).

Multi-step semantics To describe the behaviour of a circuit *over time*, we need to define another semantics. More specifically, in this work we consider only discrete-time synchronous circuits, and thus we will show how to use ⟦_|⟧s to define a multi-step state-transition semantics.

```
⟦_|⟧n : (c : λB Γ τ) (m : λs c) n (γ : Vec (Env El Γ) n) → λs c × Vec (El τ) n
⟦_|⟧n c m n = mapAccumL ⟦ c |⟧s m
```

When simulating a circuit for n cycles, we need to take not *one* input environment but n, and instead of producing a single value, the simulation returns a vector of n values. Just as we saw for mapAccumL-comb, we ensure that the newly computed state is threaded from one simulation cycle to the next.

This is exactly the behaviour of an accumulating map, thus the use of mapAccumL here. The use of mapAccumL here is the key to the connection between the multi-cycle of circuits using loop and the single-cycle behavior of circuits using mapAccumL-comb.

5 Combinational and sequential combinators

With λπ-Ware we intend to give a hardware developer more freedom to explore the trade-offs between area, frequency and number of cycles that a circuit might take to complete a computation. This freedom comes from the proven guarantees of convertibility between combinational and sequential versions of circuits.

To make it easier to explore this design space, we provide some *circuit combinators* for common patterns. Each of these patterns comes in a pair of sequential and combinational versions, with a lemma relating the two. If a circuit is defined using one of these combinators, changing between architectures is as easy as changing the combinator version used. The associated lemma guarantees the relation between the functional behaviour of the versions.

All combinators in this section are derived from the two primitive constructors loop and mapAccumL-comb. By appropriate partial application and the use of "wrappers" to create the loop body, all sequential combinators are derived from loop. Similarly, using the same wrappers but with mapAccumL-comb, we derive all combinational combinators.

Of notice is also the fact that, in this section, we present the combinators in *De Bruijn* style, as this is the most useful representation to use when evaluating circuit (generators), which is covered in 5.1

The map combinators For example, we might want to easily build circuits that *map* a certain function over its inputs. We will define both the sequential and combinational map combinators in terms of a third circuit, mapper. The sequential version is given by map-seq:

$$
\begin{aligned}
&\mathsf{mapper} \;:\; (\mathsf{f} \;:\; \lambda\mathsf{B}\ (\rho :: \Gamma)\ \tau) \to \lambda\mathsf{B}\ (\sigma :: \rho :: \Gamma)\ (\sigma \otimes \tau)\\
&\mathsf{mapper}\ \mathsf{f} \;=\; \#_0\ ,\ \mathsf{K}_1\ \mathsf{f}\\
&\mathsf{map\text{-}seq} \;:\; (\mathsf{f} \;:\; \lambda\mathsf{B}\ (\rho :: \Gamma)\ \tau) \to \lambda\mathsf{B}\ (\rho :: \Gamma)\ \tau\\
&\mathsf{map\text{-}seq}\ \mathsf{f} \;=\; \mathsf{loop}\ \{\sigma \;=\; \mathbb{1}\}\ (\mathsf{mapper}\ \mathsf{f})
\end{aligned}
$$

We define map-seq by applying loop to the mapper f circuit. In mapper, the next state (first projection of the pair) is a copy of its first input ($\#_0$), whereas the second projection is made by the *weakened* f, which discards its first input.

The combinational version of the same combinator (map-comb) is defined in terms of mapAccumL-comb and mapper:

$$
\begin{aligned}
&\mathsf{map\text{-}comb} \;:\; (\mathsf{f} \;:\; \lambda\mathsf{B}\ (\rho :: \Gamma)\ \tau)\ (\mathsf{xs} \;:\; \lambda\mathsf{B}\ \Gamma\ (\mathsf{vec}\ \rho\ \mathsf{n})) \to \lambda\mathsf{B}\ \Gamma\ (\mathsf{vec}\ \tau\ \mathsf{n})\\
&\mathsf{map\text{-}comb}\ \mathsf{f}\ \mathsf{xs} \;=\; \mathsf{snd}\ (\mathsf{mapAccumL\text{-}comb}\ (\mathsf{mapper}\ \mathsf{f})\ \mathsf{unit}\ \mathsf{xs})
\end{aligned}
$$

In the above definition we note that we are free to choose the type of the "initial element" (2nd argument), but we use $\mathbb{1}$ (value unit), as units can always be used regardless of the base type chosen in the development. Furthermore, we use snd to extract only the second element of the pair (the output vector), and discard the "final element" outputted.

The foldl-scanl combinators Perhaps even more useful than mapping is *scanning* and *folding* over a vector of inputs. To obtain the sequential and combinational versions of such combinators, we again apply the loop and mapAccumL-comb primitives to a special body which wraps the binary operation (f) of the scan/fold.

$$\mathsf{folder} : (\mathsf{f} : \lambda\mathsf{B}\ (\sigma :: \rho :: \Gamma)\ \sigma) \rightarrow \lambda\mathsf{B}\ (\sigma :: \rho :: \Gamma)\ (\sigma \otimes \sigma)$$
$$\mathsf{folder\ f} = \#_0 \ , \mathsf{f}$$
$$\mathsf{foldl\text{-}scanl\text{-}seq} : (\mathsf{f} : \lambda\mathsf{B}\ (\sigma :: \rho :: \Gamma)\ \sigma) \rightarrow \lambda\mathsf{B}\ (\rho :: \Gamma)\ \sigma$$
$$\mathsf{foldl\text{-}scanl\text{-}seq\ f} = \mathsf{loop\ (folder\ f)}$$

The wrapper called folder makes the next state equal to the first input of the binary operator, and the output be the result of applying the binary operator. In the above definition of foldl-scanl-seq, we get the behaviour of scanl and foldl *combined*: The circuit outputs from clock cycle 0 to n form the result of the scanl operation, and the last one at cycle n+1 is the value of the foldl.

The combinational version also has such a combined behaviour:

$$\begin{aligned}\mathsf{foldl\text{-}scanl\text{-}comb} &: (\mathsf{f} : \lambda\mathsf{B}\ (\sigma :: \rho :: \Gamma)\ \sigma)\ (\mathsf{e} : \lambda\mathsf{B}\ \Gamma\ \sigma)\ (\mathsf{xs} : \lambda\mathsf{B}\ \Gamma\ (\mathsf{vec}\ \rho\ \mathsf{n})) \\ &\rightarrow \lambda\mathsf{B}\ \Gamma\ (\sigma \otimes \mathsf{vec}\ \sigma\ \mathsf{n})\end{aligned}$$
$$\mathsf{foldl\text{-}scanl\text{-}comb\ f\ e\ xs} = \mathsf{mapAccumL\text{-}comb\ (folder\ f)\ e\ xs}$$

In foldl-scanl-comb, we obtain a pair as output, of which the first element is the foldl component, and the second element is the scanl (vector) component. Thus by simply applying the fst and snd functions we can obtain the usual foldl and scanl.

Whereas these combinators capture some common patterns in hardware design, their usefulness also depends on lemmas relating their combinational and sequential versions.

5.1 Convertibility of combinational and sequential versions

In this section we make precise the relation between circuits with different levels of statefulness. For conciseness, only the extreme cases are handled: completely stateless (combinational) versus completely sequential. However, nothing in the following treatment precludes it from being used for *partial unrolling*.

We will show that when two circuits are deemed “convertible up to timing”, they can be substituted for one another with minor interface changes in the surrounding context but no alteration of the values ultimately produced.

The relation of convertibility relies on the fact that any sequential circuit will have an occurrence of the loop constructor. As such, a less stateful variant of such a circuit can be obtained by substituting the occurrence of loop with one of mapAccumL-comb, thereby unrolling the loop. The fundamental relation between loop and mapAccumL-comb is what we now establish. First, recall the types of the single- and multi-step semantic functions:

$$\llbracket\_ | \rrbracket\mathsf{s} : (\mathsf{c} : \lambda\mathsf{B}\ \Gamma\ \tau)\ (\mathsf{m} : \lambda\mathsf{s}\ \mathsf{c})\ \ (\gamma : \mathsf{Env\ El}\ \Gamma) \rightarrow \lambda\mathsf{s}\ \mathsf{c} \times \mathsf{El}\ \tau$$
$$\llbracket\_ | \rrbracket\mathsf{n} : (\mathsf{c} : \lambda\mathsf{B}\ \Gamma\ \tau)\ (\mathsf{m} : \lambda\mathsf{s}\ \mathsf{c})\ \mathsf{n}\ (\gamma : \mathsf{Vec\ (Env\ El}\ \Gamma)\ \mathsf{n}) \rightarrow \lambda\mathsf{s}\ \mathsf{c} \times \mathsf{Vec\ (El}\ \tau)\ \mathsf{n}$$

Now, to establish the desired relation, we apply both the single and multi-cycle semantics. The *step function* subcircuit (called f) is equal in both cases, and the mapAccumL-comb case takes 2 extra parameters besides f.

$$
\begin{array}{l}
[\![\ \mathsf{mapAccumL\text{-}comb\ f\ (val\ m)\ xs}\ |]\!]\mathsf{s}\ (\mathsf{sMapAccumL\text{-}comb\ (replicate\ mf)})\ \gamma\ :\ \mathsf{Tpar} \\
[\![\ \mathsf{loop\ f}\ |]\!]\mathsf{n}\ (\mathsf{sLoop\ m\ mf})\ \mathsf{n}\ (\mathsf{map}\ (\_ :: \gamma)\ \mathsf{xs})\ :\ \mathsf{Tloop} \\
\quad \textbf{where}\ \mathsf{Tpar} \ = \ \lambda\mathsf{s}\ (\ldots) \times \mathsf{Tp}\ \sigma \times \mathsf{Vec}\ (\mathsf{Tp}\ \tau)\ \mathsf{n} \\
\quad\quad\quad\quad \mathsf{Tloop} \ = \ \lambda\mathsf{s}\ (\ldots) \times \mathsf{Vec}\ (\mathsf{Tp}\ \tau)\ \mathsf{n}
\end{array}
$$

The second parameter of mapAccumL-comb must be a circuit whose value is the same as the first parameter of sLoop, and we use here the simplest possible such circuit: (val m). The third parameter (xs) is the input vector of size n, and is used to build the vector of environments used by the multi-cycle semantics (map (_:: γ) xs).

Finally, the state of the mapAccumL-comb case is built by simply replicating one state of f by n times. Stating the convertibility property in this way makes it be valid only for a state-independent f, that is, when the input/output semantics of f is independent of the state.

$$
\begin{array}{l}
\mathsf{state\text{-}independent}\ :\ \forall\ (\mathsf{c}\ :\ \lambda\mathsf{B}\ \Gamma\ \tau) \rightarrow \mathsf{Set} \\
\mathsf{state\text{-}independent\ c}\ =\ \forall\ \mathsf{sa\ sb}\ \gamma \rightarrow [\![\ \mathsf{c}\ |]\!]\mathsf{s\ sa}\ \gamma \equiv [\![\ \mathsf{c}\ |]\!]\mathsf{s\ sb}\ \gamma
\end{array}
$$

This restriction on f could be somewhat further loosened (as is discussed in Section 6.2), but we work here with state-independent loop bodies to simplify the presentation.

As we have seen, the results from applying each semantic function have different types (Tpar and Tloop), so the relation comparing these results is more subtle than just equality. We define this relation, called _=*_, as follows:

$$
\begin{array}{l}
\_{=}{*}\_\ :\ \mathsf{Tpar} \rightarrow \mathsf{Tloop} \rightarrow \mathsf{Set} \\
(\_\ ,\ \mathsf{s}'\ ,\ \mathsf{xs}')\ {=}{*}\ (\mathsf{sm}''\ ,\ \mathsf{xs}'')\ =\ (\mathsf{s}' \equiv \mathsf{gets}_0\ \mathsf{sm}'') \times (\mathsf{xs}' \equiv \mathsf{xs}'')
\end{array}
$$

Both sides of (_=*_) consist of a pair of next state and circuit outputs. In the mapAccumL-comb case, the next state can be ignored in the comparison, but in the loop case, the *value* stored in the loop state (obtained by gets_0) must be equal to the first output of evaluating mapAccumL-comb. With the comparison function defined, we can finally completely express the relation we desire:

$$
\begin{array}{l}
\quad [\![\ \mathsf{mapAccumL\text{-}comb\ f\ (val\ m)\ xs}\ |]\!]\mathsf{s}\ (\mathsf{sMapAccumL\text{-}comb\ (replicate\ mf)})\ \gamma \\
{=}{*}\ [\![\ \mathsf{loop\ f}\ |]\!]\mathsf{n}\ (\mathsf{sLoop\ m\ mf})\ \mathsf{n}\ (\mathsf{map}\ (\_ :: \gamma)\ \mathsf{xs})
\end{array}
$$

Proof of the basic relation The proof of the basic convertibility relation between mapAccumL-comb and loop proceeds by induction on the input vector xs. Due to the deliberate choice of semantics for both constructors involved, and the choice of the right parameters for the application of each, a considerable part of the proof is achieved by just the built-in reduction behaviour of the proof assistant (Agda).

The only key lemma involved is shown below. Namely, the state-independence principle is shown to hold for a whole vector, assuming that it holds for the body circuit f.

```
state-independent-vec : ∀ (mfas mfbs : Vec (λs f) n) (p : state-independent f)
                          →   mapAccumL2 (transformF ⟦ f |⟧s2 γ) e mfas xs
                            ≡ mapAccumL2 (transformF ⟦ f |⟧s2 γ) e mfbs xs
```

This lemma is useful because both left-hand side and right-hand side of the convertibility relation can be transformed into applications of mapAccumL2 simply by reduction, but with different state vector parameters. Thus the lemma is used to bring the sub-goals to a state where they can be closed by using the induction hypothesis.

```
mapAccumL-comb-seq :
       ⟦ mapAccumL-comb f (val m) xs |⟧s (sMapAccumL-comb (replicate mf)) γ
  =* ⟦ loop f                        |⟧n (sLoop m mf) n (map (_:: γ) xs)
mapAccumL-comb-seq f mf m (x :: xs) γ p = g:m , g:ys where
  m′ , mf′ , y = (transformF ⟦ f |⟧s2 γ) m mf x   -- take one step
  ih:m , ih:ys = mapAccumL-comb-seq f mf′ m′ xs γ p   -- ind. hyp.
  lemma = state-independent-vec f xs (replicate mf) (replicate mf′) p
  g:m  : p₁ (p₂ (⟦ mapAccumL-comb f ... |⟧s ...)) ≡ gets₀ (p₁ (⟦ loop f |⟧n ...))
  g:ys : p₂ (p₂ (⟦ mapAccumL-comb f ... |⟧s ...)) ≡        p₂ (⟦ loop f |⟧n ...)
  g:m  =            (cong ... lemma) ⟨ trans ⟩ ih:m
  g:ys = cong₂ ... ((cong ... lemma) ⟨ trans ⟩ ih:ys)
```

Convertibility of derived combinators When building circuits using the derived combinators (map, foldl-scanl, etc.), the convertibility between different (more or less stateful) variants of such circuits rely on the convertibility between the different variants of the combinators themselves.

The basic convertibility principle shown above between mapAccumL-comb and loop is the most general one, and can be directly applied to the derived combinators as well, as they are all just a specialized instance of mapAccumL-comb or loop. However, for the derived combinators, some more specific properties are useful.

With regards to the map combinators, for example, we wish that the vectors produced by the combinational and sequential versions be equal, without any regard for initial or final states. This can be succinctly expressed as:

```
  snd (⟦ map-comb f xs |⟧s units  γ)
≡ snd (⟦ map-seq f     |⟧n units′ (map (_:: γ) xs))
```

Where units and units′ are simply the states (composed of units) that need to be passed to the semantic function but are irrelevant for the computed vectors.

On the other hand, when comparing foldl-comb to foldl-seq, the intermediate values produced in the output of foldl-seq are disregarded, and only the final state matters.

```
  fst (⟦ foldl-comb f (val e) xs |⟧s m             γ)
≡ fst (⟦ foldl-seq f             |⟧n (sFoldl e m) (map (_:: γ) xs))
```

Both of these properties (for map and for foldl) can simply be proven by application of the general property shown above for mapAccumL-comb and loop. This is because the definition of the derived combinators is just a partial application of mapAccumL-comb and loop, along with projections.

5.2 Applications of the combinational and sequential combinators

In this section we describe several variants of circuit families that compute matrix multiplication, as a commonly used application of the aforementioned techniques.

The first design choice involved in this example application is *how to represent* matrices, i.e., the choice of the *matrix type*. Traditionally in computing contexts, matrices are mostly represented in two ways: *row major* (vector of rows) and *column major* (vector of columns). As it turns out, *both* representations are useful for our purposes, so we show both here:

```
RMat CMat : (r c : ℕ) → U ℕ
RMat r c = vec (vec ℕ c) r
CMat r c = vec (vec ℕ r) c
```

Here, RMat r c and CMat r c both represent matrices with r rows and c columns, the difference being only whether they are row- or column-major. Going further with the example, we need to define the basic ingredient of matrix multiplication: the *dot product* of two equally-sized vectors.

```
dp : λH (vec ℕ n) → λH (vec ℕ n) → λH ℕ
dp xs ys = foldl-comb _:+:_ (val 0) (zipWith-comb _:*:_ xs ys)
```

The dot product is simply defined as element-wise multiplication of the vectors and summing up the results. We can then use the dot product m times in order to multiply a vector by a compatibly-sized matrix.

```
vec×mat-comb : λH (vec ℕ n) → λH (CMat n m) → λH (vec ℕ m)
vec×mat-comb v m = map-comb (dp v) m
```

Here an important detail resides: as the dot product is done for *each column* of the matrix, the matrix argument of vec×mat must be in *column-major* representation. Also, here we start having choices: we may either have the computation done combinationally as above, or sequentially as below:

```
vec×mat-seq : λH (vec ℕ n) → λH (vec ℕ n) → λH ℕ
vec×mat-seq v m = map-seq (dp v) m
```

With the multi-step semantics in mind, we know that each of the m columns of the matrix will be present on the circuit's second input, one per clock cycle, and that collecting the output values for m cycles gives the same vector of results as the one from the combinational version.

For defining the multiplication of two matrices, we simply use vec×mat on each row of the left matrix. If using vec×mat-comb, we obtain a matrix multiplication circuit with area proportional to r * c, whereas by using vec×mat-seq the area is proportional to r * 1.

```
mat×mat-comb : λH (RMat n m) → λH (CMat m p) → λH (RMat n p)
mat×mat-comb mr mc = map-comb (flip vec×mat-comb mc) mr
mat×mat-seq : λH (RMat n m) → λH (vec ℕ m) → λH (vec ℕ n)
mat×mat-seq mr mc = map-comb (flip vec×mat-seq mc) mr
```

In the combinational version (mat×mat-comb), all the rows in the resulting matrix are computed in parallel, with the column-positioned values inside each row computed also in parallel. In the sequential version, at each clock cycle one whole *column* is produced, with the row-positioned values inside each column computed in parallel.

Matrix multiplication as defined here has two nested recursion blocks, and thus four ways in which it could be sequentialized. Above we have shown two possible such choices, and the other two can simply be obtained by swapping map-comb for map-seq.

6 Discussion

6.1 Related work

There is a rich tradition of using functional programming languages to model and verify hardware circuits, Sheeran (2005) gives a good overview – we restrict ourselves to the most closely related languages here. Languages embedded in Haskell, such as Lava and Wired, typically rely on automated theorem provers and testing using QuickCheck for verification. In λπ-Ware, however, we can perform *inductive* verification of our circuits. Existing embeddings in most theorem provers, such as Coquet (Braibant, 2011) and *Π*-Ware (Pizani Flor et al., 2016), have a more limited treatment of variable scoping and types. More recent work by Choi et al. (2017) is higher level, but sacrifices the ability to be simulated directly (using denotational semantics) in the theorem prover.

6.2 Future work

Other timing transformations While our language easily lets you explore possible designs, trading time and space, there are several alternative transformations, such as *pipelining* that we have not yet tried to describe in this setting.

While we have a number of combinators for transforming between combinational and sequential circuits, these are mostly aimed at linear, list-like data. Even though these structures are the most prevalent in hardware design, we would like to explore related timing transformations on tree-structured circuits. To this end, it would be interesting to look into the formalization and verification of *flattening* transformations, and of the work done in the field of *nested data parallelism*.

Relaxed unrolling restriction In Section 5.1 we mention that the proof of semantics preservation for loop unrolling relies on the premise that the loop body is *state-independent*, that is, it has the same input/output behaviour for any given state. This premise can be relaxed somewhat, and proving that loop unrolling still preserves semantics under this relaxed premise is (near-)future work.

The relaxed restriction on the body f of a loop to be unrolled is as follows:

$$\begin{aligned}&\mathsf{state\text{-}input\text{-}independent} \;:\; \forall\, (\mathsf{c} \;:\; \lambda\mathsf{B}\;\Gamma\;\tau) \to \mathsf{Set}\\ &\mathsf{state\text{-}input\text{-}independent}\;\mathsf{c} \;=\; \mathsf{fst}\;(\llbracket\;\mathsf{c}\;\rrbracket\mathsf{s}\;\mathsf{sa}\;\gamma) \equiv \mathsf{fst}\;(\llbracket\;\mathsf{c}\;\rrbracket\mathsf{s}\;\mathsf{sa}\;\delta)\end{aligned}$$

That is, the next state (fst projection) is equal even with evaluation taking different input environments. This condition is necessary because when writing the combinational version of a loop construct we must give each copy of f its own initial state. As the desired initial state for each such copy must be known at verification time, it cannot depend on input.

Using the definitions from Section 5.1 along with the relaxed hypotheses above, we can show that not only total, but also *partial unrolling* preserves semantics up to timing.

7 Conclusion

There are several advantages to be gained by embedding a hardware design DSL in a host language with dependent types, such as Agda. Among these advantages are the easy enforcement of some well-formedness characteristics of circuits, the power given by the host's type system to express object language types and design constraints. The crucial advantage though, is the ability to have *modelling, simulation, synthesis and theorem proving* in the same language.

By using the host language's theorem-proving abilities, we are able not only to show properties of individual circuits, but of (infinite) classes of circuits, defined by using *circuit generators*. Particularly interesting is the ability to have *verified transformations*, preserving some semantics.

The focus of this paper lies on timing-related transformations, but we also recognize the promise of theorem proving for the formalization of other non-functional aspects of circuit design, such as power consumption, error correction, fault-tolerance and so forth. The formal study of all these aspects of circuit construction and program construction could benefit from mechanized verification.

Acknowledgments

We would like to thank the very helpful feedback gathered during the visit to Chalmers University of Technology, funded by COST Action EUTypes CA15123. Especially valuable were the meetings and discussions with Mary Sheeran, whose deep knowledge of the field oriented this work in its beginning stage. Also, we are very thankful to the feedback given during our presentation on this topic at the TFP2017 conference in Canterbury.

This work was supported by the Netherlands Organization for Scientific Research (NWO) project on *A Dependently Typed Language for Verified Hardware*.

Bibliography

Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding Domain-specific Languages. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 37–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi: 10.1145/1596638.1596644. URL `http://doi.acm.org/10.1145/1596638.1596644`.

Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-Aware Circuit Design. In *Correct Hardware Design and Verification Methods*, pages 5–19. Springer, Berlin, Heidelberg, October 2005. doi: 10.1007/11560548_4. URL `https://link.springer.com/chapter/10.1007/11560548_4`.

Christiaan Pieter Rudolf Baaij. *Digital circuits in ClaSH: functional specifications and type-directed synthesis.* info:eu-repo/semantics/doctoralThesis, University of Twente, Enschede, January 2015. URL `https://doi.org/10.3990/1.9789036538039`.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1):174–184, January 1999. ISSN 03621340. doi: 10.1145/291251.289440. URL `http://portal.acm.org/citation.cfm?doid=291251.289440`.

Richard J Boulton, Andrew D Gordon, Michael JC Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *TPCD*, volume 10, pages 129–156, 1992.

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.

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. ISBN 978-3-642-25378-2, 978-3-642-25379-9. URL `http://link.springer.com/chapter/10.1007/978-3-642-25379-9_24`.

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. ISBN 978-3-642-39798-1 978-3-642-39799-8. URL `http://link.springer.com/chapter/10.1007/978-3-642-39799-8_14`.

Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1 (ICFP):24:1–24:30, August 2017. ISSN 2475-1421. doi: 10.1145/3110268. URL `http://doi.acm.org/10.1145/3110268`.

Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. Introducing Kansas Lava. In *Proceedings of the Symposium on Im-*

plementation and Application of Functional Languages, volume 6041 of *LNCS*. Springer-Verlag, Sep 2009.

F. K. Hanna and N. Daeche. Dependent Types and Formal Synthesis. *Philosophical Transactions: Physical Sciences and Engineering*, 339(1652):121–135, April 1992. ISSN 0962-8428. URL `http://www.jstor.org/stable/54016`.

T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993. ISBN 0-521-41718-X. doi: 10.1017/CBO9780511569845. URL `http://www.cs.ox.ac.uk/tom.melham/pub/Melham-1993-HOL.html`.

Joao Paulo Pizani Flor, Yorick Sijsling, and Wouter Swierstra. π-ware : Hardware description and verification in agda. In Tarmo Uustalu, editor, *21th International Conference on Types for Proofs and Programs (TYPES 2015)*, Leibniz International Proceedings in Informatics (LIPIcs), 2016.

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. ISSN 0278-0070. doi: 10.1109/TCAD.2003.819898.

M Sheeran. Hardware Design and Functional Programming: a Perfect Match. 2005. URL `http://www.jucs.org/jucs_11_7/hardware_design_and_functional/jucs_11_7_1135_1158_sheeran.pdf`.

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. ISBN 0897911423. doi: 10.1145/800055.802026. URL `http://portal.acm.org/citation.cfm?doid=800055.802026`.

S. Singh. Designing reconfigurable systems in Lava. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 299–306, 2004. doi: 10.1109/ICVD.2004.1260941.