Abstract. Synchronous hardware can be modelled as a mapping from input and state to output and a new state. Functions in this form are referred to as transition functions. It is natural to use a functional language to implement transition functions. The CλaSH compiler is capable of translating Haskell code written in this form to VHDL. Modelling hardware using multiple components is convenient. Components can be considered as instantiations of functions. To avoid packing and unpacking state when composing components, functions are lifted to arrows. By using arrows the chance of making errors will decrease as it is not required to manually (un)pack the state. Furthermore, the Haskell do-syntax for arrows is a pleasant notation for describing hardware designs.

1 Introduction

In synchronous digital hardware, a clock signal is used as the heartbeat of the digital circuit. Synchronous digital hardware can be modelled using a Mealy machine. In a Mealy machine, current inputs \((i)\) and the current state \((s)\) which is stored inside registers are mapped, using a transition function, to a new state \((s')\) and output of the circuit \((o)\), see figure 1.

This transition function can be seen as a mathematical function, which is applied to the inputs and state at every clock cycle. However, when designing digital hardware, viewing the transition function as a pure mathematical function is not enough. For instance, \((a + b) \cdot c\) and \(a \cdot c + b \cdot c\) are mathematically equivalent since \(\cdot\) is distributive over +. However, in a hardware implementation the
latter expression would yield a bigger circuit. When designing hardware, some of the design steps result in different hardware with exactly the same functionality, but with different non-functional properties such as speed or area. Mostly, digital logic is implemented as an Application-Specific Integrated Circuits (ASICs) or instantiated in a Field Programmable Gate Arrays (FPGAs). FPGAs are reconfigurable logic devices which are mainly used for prototyping and decreasing the time to market. Since manual translation of a transition function to descriptions that can be used to produce the physical hardware is very cumbersome, this is often automated using software. The hardware is described using a Hardware Description Language (HDL), for which synthesis tools exists to translate the source code to a low level hardware description. This description specifies which components are used and how they are connected. The physical place of components and their interconnect inside the ASIC or FPGA can be assigned using a Place And Route (PAR) tool. A particularly nice feature of many HDLs is, that they allow modularization, which enable engineers to separately design relatively small pieces of hardware. The HDLs have syntax to compose components and instantiate components multiple times.

Two popular HDLs are VHDL and Verilog. Although it is possible to use these languages and the respective tools to design hardware and synthesise it, the source code descriptions are far from the mathematical function we started with. Furthermore, it is hard to prove that functionality remains the same after some design steps. In our experience, the hardware descriptions written in a (modified) Haskell subset are very compact and well readable when compared to their equivalent VHDL descriptions. Because of these reasons, it is natural to use functional programming languages to design synchronous digital hardware. In [2, 9] we have introduced a modified subset of Haskell, together with a compiler called CλaSH, which is based on the Glasgow Haskell Compiler (GHC).

In [9] the streaming reduction circuit, introduced in [6], was used as an example of how to describe hardware in a functional language. When using CλaSH, the user has to pass the state and current input of the circuit to the transition function. Since most circuits consist of many components, which in turn can consist of subcomponents, the user has to thread the state through the functions by packing and unpacking it. This might seem unnatural when viewing the circuit as a connection of components. The states are local to their corresponding instance of a component and only the inputs and outputs of components are connected. Furthermore, distributing and collecting the state manually increases the chance of making errors.

In this paper, we describe how to hide the state from the user by using an automata arrow [11]. Each arrow describes a component, which can be combined with other arrows (components). It is possible to combine multiple arrows to a single arrow, which is similar to combining many subcomponents to a single component. The main contribution of this article is showing how to deal with state when designing synchronous hardware using Haskell and presenting this using a nontrivial example.
Section 2 introduces related work and compares this to our own work. Arrows will be shortly discussed in section 3. Section 4 explains how to deal with the hardware state, when designing synchronous hardware in Haskell. In section 5, the streaming reduction circuit\cite{2} is introduced and implementations with and without arrows are compared.

2 Related work

Where the CλaSH compiler takes Haskell code as input, Lava\cite{3} and ForSyDe\cite{13} are domain specific embedded languages defined in Haskell. Both languages are stream processing languages. They operate on infinite streams. The state of synchronous hardware can be modelled using a delay function. By using this function, the state of the hardware is introduced. Instead of defining mappings from streams to streams, CλaSH defines a mapping from current input and current state to the next state and output, which corresponds to a Mealy machine. Since the input of CλaSH is not a domain specific language, all choice constructs in Haskell (if, guards, pattern matching, etc) are available. Lava has only the “mux” primitive, ForSyDe supports the if-then-else and case-expressions. Like Kansas Lava\cite{7} and ForSyDe, CλaSH has support for integer types and primitive operations; Chalmers Lava has only support for the bit type and related primitives. CλaSH, Lava and ForSyDe support polymorphic, higher-order functions. ForSyDe requires explicit wrapping of functions and processes and also explicit component instantiations, making descriptions in ForSyDe more cluttered than those in CλaSH.

In VHDL\cite{1}, all components are created using component declarations and connected using port maps. From variable and signal declarations in VHDL it is not clear if these variables and signals will become state. This depends on the actual code, not on the declarations. When using CλaSH, this is more transparent, as the current and next states are explicitly defined. Higher-level abstractions are cumbersome in VHDL, functional languages are better suited when high-level abstractions are desired.

In\cite{11}, arrows are introduced and circuits using delay functions are taken as an example. In section 5, it is shown that arrows can also be used for functional hardware modelled with Mealy machines whereas examples in\cite{11} do not make the state explicit in the function arguments and use a delay function instead. In the examples in\cite{11}, only relatively small hardware designs were explored. We will show it is possible, using Haskell and CλaSH, to create relatively big hardware designs. In our approach we will use the automata arrow as introduced in\cite{11}.

3 Arrows

This section briefly discusses arrows in Haskell, enough to understand the remainder of this article. For an elaborate discussion we refer to\cite{8} or\cite{11} which both contain an excellent introduction to arrows in Haskell.
Arrows can be used to describe components, which have several inputs and several outputs, and their compositions. An arrow can be depicted graphically as in figure 2, where $\beta$ and $\gamma$ denote the types of the input and output data, respectively, and where $f$ denotes the associated function from $\beta$ to $\gamma$. Since arrows provide a general structure, Haskell libraries [8, 11] have been defined with functions common to many arrows. Examples of arrows are functions, stream processors, state transformers, etc.

Arrows give a uniform interface for composition. Every arrow in Haskell is an instance of the type class $\text{Arrow}$ and of the type class $\text{Category}$. However, in this article we will use the notation as used in [11] and introduce arrows as a single type class $\text{Arrow}$.

For every arrow, sequential and parallel composition can be defined. To create an arrow from a regular function, the function $\text{pure}$ is used. The function $\ggg$ takes two arrows and composes them such that the output of the first arrow is connected to the input of the second arrow. For parallel composition, the function $\text{first}$ is used. The function $\text{first}$ takes an arrow with input type $\beta$ and output type $\gamma$ and creates a new arrow with input and output types respectively $(\beta, \delta)$ and $(\gamma, \delta)$. The arrow that is its argument is only applied to the first element of the tuple. The second element in the tuple will not be modified. The
function `second` is similar to `first`, except that it applies the arrow to the second element of the tuple. This function can be defined by using `first`. In figure 3, the expression `(first f) >>> (second g)` is shown graphically. The type class `Arrow` is defined as in listing 1.

```haskell
class Arrow α where
  pure :: (β → γ) → α β γ
  (>>>) :: α β γ → α γ δ → α β δ
  first :: α β γ → α (β, δ) (γ, δ)

Listing 1. The Arrow type class
```

![Diagram of an arrow loop](image)

Fig. 4. An arrow loop

Using these operators all parallel and sequential structures can be created. To create feedback loops, another type class called `ArrowLoop` is required, see figure 4. This type class is defined as in listing 2.

```haskell
class Arrow α ⇒ ArrowLoop α where
  loop :: α (β, δ) (γ, δ) → α β γ

Listing 2. The ArrowLoop type class
```

To model hardware, we use one specific arrow, namely the automata arrow. The automata arrow, described in [11], takes an input and produces an output together with a new automata arrow. We will use this to store the state inside the new arrow. In the next section a function is defined to lift a transition function to an automata arrow. The reason why we use the automata arrow together with the lifting function, instead of the circuit arrow from [11] is that the new arrow will have strong correspondence to the transition function. When using the form we propose, the arrow (which contains the state) receives an input and produces a new arrow (which contains state) together with an output. The automata arrow is defined in listing 3.
newtype Aut i o = A {
    apply :: i -> (o, Aut i o)
}

instance Arrow Aut where
    pure f = A (\b -> (f b, pure f))
    (A f) >>> (A g) = A (\b -> let (c, f') = f b
                          (d, g') = g c
                          in (d, f' >>> g'))

instance ArrowLoop Aut where
    loop (A f) = A (\i -> let ((c, d), f') = f (i, d)
                     in ((c, d), first f'))

Listing 3. Definition of the Automata Arrow

4 State

To model a Mealy machine, a transition function maps the input and the state to output and a new state, as was explained in section 1. In C\lambda aSH, this state is a function argument to the transition function. All transition functions in C\lambda aSH are defined as

\text{transition} :: \text{state} \to \text{input} \to (\text{state}, \text{output})

The input state and output state have the same type (state), as both correspond to the register contents. The types input, output and state can be freely constructed using a predefined set of types, under the restriction that their sizes are finite and fixed at compile time. Both restrictions are natural since using these types, hardware registers and buses are created.

The automata arrow will be used to hide state inside the arrow. Instead of using the transition function, a new function of type \text{Aut} is defined which maps input to an output and a new function of type \text{Aut}. The function of type \text{Aut} is an automata arrow. The function itself contains the state. The type of the state cannot be observed from the type \text{Aut}. Because of this, the state is not required as an argument to this function and is effectively hidden. Writing transition functions is natural when designing hardware, writing the transition functions directly as automata arrows is not. However, the correspondence between the two is clear and a mapping from a transition function to a automata arrow is defined using the lifting function in listing 4.
\[
\begin{align*}
\&(\text{..}) \colon (s \to i \to (s, o)) \to s \to \text{Auto} i o \\
\&(\text{..}) \ f \ \text{init} = A \ \text{applyS} \\
&\text{where} \ \text{applyS} = \ i \ \to \ \text{let} \ (s, o) = f \ \text{init} \ i \\
&\text{in} \ (o, f \ \text{..} \ s)
\end{align*}
\]

Listing 4. Function to lift a transition function to an automata arrow

This function requires the transition function and an initial state as function arguments. The initial state is used when the system is reset, which for instance occurs after power on. Since the creation of a new arrow cannot be implemented in actual hardware, the CλaSH compiler recognises the arrow, extracts the state and creates registers that represent the state.

Fig. 5. Multiply Accumulate transition function

One of the simplest synchronous circuits is the multiply accumulate (MAC). The accumulator adds the product of the inputs to its state and uses the result as new state and sends it to the output. The corresponding transition function is visualised in figure 5. It is defined in listing 5.

\[
\text{mac acc} \ (x, y) = (\text{acc}', \ \text{acc}')
\]

\text{where}

\[
\text{acc}' = \text{acc} + x \ast y
\]

Listing 5. A multiply accumulate

When arrows are not used, the initial state is passed as an extra argument to CλaSH when the circuit is simulated or when the circuit is synthesised. When the circuit is lifted to an arrow, the initial state is an argument to the lifting function \text{..}, which hides the state inside the function. To lift the function \text{mac} to the arrow \text{macA} using the initial state 0, the following definition is used:

\[
\text{macA} = \text{mac} \ \text{..} \ 0
\]

In CλaSH the do-notation to compose arrows is available, which was introduced in [12]. Using this do-notation, the arrows are automatically composed using first, >>= and pure. If the keyword rec is specified, loop is automatically used to compose arrows which require feedback. Section 5 shows an example which uses rec. In listing 6 it is shown how to define a circuit which contains two MACs, of which the results are added to produce an output.
macsum = proc (a, b, c, d) -> do
r1 ← mac °°° 0 ← (a, b)
r2 ← mac °°° 0 ← (c, d)
returnA ← (r1+r2)

Listing 6. Composing MAC components

In this example, the two components appear at lines 2-3. At the right, the inputs of the components are specified. When a component has multiple inputs, tuples are used. Between the two arrows, the transition function mac is shown, lifted to an arrow using the initial state 0. The output appears at the left side of the lines describing the components. The arrow macsum receives the inputs (a, b, c, d) at line 1 and returns r1 + r2 as output at line 4.

5 Reduction Circuit

The small example in the previous section does not yet show the full strength of C\(\lambda\)aSH, nor why arrows are useful. A more elaborate example of a circuit is the streaming reduction circuit [6], which is introduced below.

When solving the matrix equation \(Ax = b\) for a big sparse positive definite matrix \(A\), often the conjugate gradient algorithm is used. The conjugate gradient algorithm is time consuming, while for some applications a fast response is required. One method to enable a fast execution of this algorithm is by implementing this algorithm in hardware, for instance using an FPGA. A kernel operation of the conjugate gradient algorithm is the sparse matrix vector multiplication (SM\(\times\)V). When calculating a matrix vector multiplication, dot products can be used to calculate the elements of the result vector. For an SM\(\times\)V, the number of multiplications and additions required for an element in the result vector depends on the number of non-zeros in the respective row of the matrix. In most FPGA implementations, a pipelined floating point adder is used to calculate the additions. Every clock cycle an addition can be scheduled, however it will take several clock cycles before the result is available because the adder is pipelined.

This causes some difficulties when summing a row of numbers, as is required for an SM\(\times\)V. Take for instance a row of three values summed using a pipelined adder of 14 stages. An ideal situation would be using a single pipelined floating point adder, which receives the rows of floating point values at its input. Some logic is used to identify the rows and to make sure the correct values are added using minimal buffering, as buffers are relatively expensive. It is trivial to add the first two values. However, it will take 14 clock cycles before the result is available and can be added to the third value. Meanwhile, values of other rows are available for reduction. This illustrates the pipeline can be scheduled to reduce values of multiple rows simultaneously.

Various circuits which can sum rows of floating point values exists. These are called reduction circuits. Since these reduction circuits use pipelining and because of varying row lengths, it is hard to design a reduction circuit. Reduction circuits are an active area of research. Many other reduction circuits with
different properties are available [4–6, 10, 14]. Several designs rely on either a minimal or a maximal row length, where some require multiple adders, while others schedule a single floating point adder.

In [6] the streaming reduction circuit is introduced, together with an algorithm to schedule the pipeline and a proof to show that the defined buffer sizes are sufficient. An implementation of the streaming reduction circuit in both VHDL and C\text{aSH} was compared in [2]. In the streaming reduction circuit, values appear sequentially at the input, one value at every clock cycle. The streaming reduction circuit uses a single floating point adder with $\alpha$ pipeline stages. This pipeline is denoted by $P$. If two values of the same row are available at the input, they can be summed by inserting them into the pipeline. Since intermediate results which appear at the output of the pipeline have to be further reduced, they have to be temporarily stored. For the streaming reduction circuit, this is done in the partial result buffer (denoted by $R$). This partial result buffer has an additional task: it will reorder the final results, such that the results are sent to the output of the reduction circuit in the order of their arrival. When two intermediate results are reduced, it is not possible to simultaneously reduce values which appear at the input. Therefore, the values at the input must be buffered and their order of arrival must be preserved. To this end, we use a FIFO input buffer (denoted by $I$). To determine if either values from the input buffer, from the end of the pipeline and/or from the partial result buffer are used, five rules are checked. The rules can determine which values to use, i.e. the top two values from $I$ (denoted as $I_1$ and $I_2$), the output of the the adder pipeline (denoted as $P_\alpha$) or values from $R$.

![Fig. 6. Reduction circuit](image-url)
The five rules, in descending order of priority, are:

1. If there is a value available in $R$ with the same row index as $P_\alpha$, then this value from $R$ enters the pipeline together with $P_\alpha$.
2. If $I_1$ has the same index as $P_\alpha$, then $I_1$ and $P_\alpha$ enter the pipeline.
3. If there are at least two elements in $I$, and $I_1$ and $I_2$ have the same index, then they enter the pipeline.
4. If there are at least two elements in $I$, but $I_1$ and $I_2$ have different indexes, then $I_1$ enters the pipeline together with the unit element of the operation dealt with by the pipeline (thus for example, 0 in case of addition, 1 in case of multiplication).
5. In case there are less than two elements available in $I$, no elements enter the pipeline.

The rules are schematically shown in figure 7. The datapath of the reduction circuit is shown in figure 6. The components $I$, $R$, and $P$, together with the controller ($C$) are shown in this figure. To identify rows within the reduction circuit, discriminators are used as identification. They are assigned to new rows which enter the reduction circuit and are released when a row is fully reduced and leaves the reduction circuit, after which the discriminator is reused. Discriminators require fewer bits than the row index.

Although figure 6 makes clear how data flows through the reduction circuit, it neglects the control signals. Figure 8 shows the entire circuit including control signals, as it was implemented in Haskell. The controller, denoted by $C$, checks which rule has to be executed and routes all signals. The discriminators are assigned by $D$.

To connect the components in the case the automata arrow is not used, the code from listing 7 was used.
Hiding State in CΛaSH Hardware Descriptions

As described before, the user has to manually extract the substates from the reduction circuit state, apply the transition functions and then repack it. In line 3, all states (denoted by \( \sigma_{\text{red}} \)) are unpacked. Lines 4-8 show how the components are instantiated and how the state is distributed to all components. The new states (denoted by \( \sigma'_{\text{x}} \)) are collected and packed at the lines 9-11. Besides risking possible errors and making the code hard to read, this requires work from the user which can be automated by CΛaSH. When arrows are used, the function which connects the components looks as shown in listing 8.

**Listing 7. Reduction circuit without arrows**

\[
\begin{align*}
\text{reduction} & \quad \sigma_{\text{red}} (x,i) = (\sigma'_{\text{red}}, y) \\
\text{where} & \\
\text{State Reduction} & \quad \{ ... \} = \sigma_{\text{red}} \\
(\sigma'_D, (\text{new}, d)) & = D \sigma_D i \\
(\sigma'_I, (i_1, i_2)) & = I \sigma_I (x,d,\delta) \\
(\sigma'_P, \rho) & = P \sigma_P (a_1, a_2) \\
(\sigma'_R, (r,y)) & = R \sigma_R (\text{new}, d, \rho, r) \\
(\sigma'_C, (a_1, a_2, \delta, r')) & = C \sigma_C (i_1, i_2, \rho, r) \\
\sigma'_{\text{red}} & = \text{State Reduction} \{ \sigma_D = \sigma'_D, \sigma_I = \sigma'_I, \\
& \quad \sigma_P = \sigma'_P, \sigma_R = \sigma'_R, \\
& \quad \sigma_C = \sigma'_C \} 
\end{align*}
\]

**Fig. 8. Reduction circuit signals**
\textbf{Listing 8.} Reduction circuit with arrows

Listing 8 shows that manually unpacking and packing the state is not required anymore when using arrows. The transition functions are now lifted using an initial state (denoted by the calligraphic letters with subscript zero) to arrows (lines 3-7). Only the composition of the components is shown, the state is only visible through the initial state. Since it is natural to define the initial state where the component is instantiated, this is a desired notation.

When arrows are used to implement the reduction circuit, it is clear an \textit{ArrowLoop} is required. In line 2 of listing 8 this is enabled using \texttt{rec}. The component (or function) \(I\) requires a result from the controller, while the controller requires a result from \(I\), i.e. the functions depend on each other’s results. In figure 8, this is shown using the signals \(\delta, i_1\) and \(i_2\). These same signals are shown in listing 8. Because the result produced by the input buffer does not depend on the signal sent by the controller, Haskell’s lazy evaluation will make sure this functional dependency will not be a problem.

6 Conclusions and Future Work

Functional languages are well suited for hardware design. The well known Mealy machine can be described using a function from input and state to output and a new state. This can be modelled in a functional language using a single function, called the transition function. Dealing with state when not using arrows is very cumbersome. The notation of arrows yields both a pleasant notation and a method to hide the state inside the arrow.

Our approach was tested by modelling the streaming reduction circuit, a nontrivial circuit, in Haskell and compile it using C\texttt{\lambda}aSH. From this example, it is clear that it is possible to design nontrivial hardware using Haskell. Using arrows, packing and unpacking the substates can be avoided, which makes the code easier to read. ArrowLoop is used since loops are often required for digital hardware design. Since for such hardware, not all outputs depend directly on the input values. Because of this, lazy functional languages are desirable for hardware design.

Although using this extension it becomes easy to design and implement synchronous hardware using C\texttt{\lambda}aSH, it is not yet possible to design hardware with multiple clock domains. Only synchronous hardware is supported by C\texttt{\lambda}aSH. In the future, support for asynchronous hardware will be considered. Further research is required in these directions.
References