Higher-Order Abstraction in Hardware Descriptions with CλaSH

Marco Gerards, Christiaan Baaij, Jan Kuper, Matthijs Kooijman
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
m.e.t.gerards@utwente.nl

***Abstract*—Synchronous hardware can be straightforwardly modelled as a function from input and (current) state to an updated state and output. The CλaSH compiler can translate such a transition function, described in a functional language, to synthesisable VHDL. Taking a hardware-oriented viewpoint, components can then be seen as an instantiation of such a transition function. An abstraction called Arrows is used to directly model components by combining a transition function and its state. The abstraction also provides an uniform interface for composition, without losing the referential transparency offered by a functional description. Furthermore, readability of hardware designs is increased by the use of the γ-syntax, that automatically composes components according to the Arrow interface. The advantages of the Arrow abstraction and the γ-syntax are demonstrated by means of a realistic example circuit consisting of multiple components. This is a significant extension to CλaSH and enables many high level abstractions.**

***Keywords*-Functional Programming, Hardware description languages, Pipeline processing**

I. Introduction

Synchronous digital hardware can be modelled using a Mealy machine, where current inputs (i) and the current state (s) are mapped, using a transition function, to a new state (s') and output of the circuit (o), see figure 1. The state is stored inside registers.

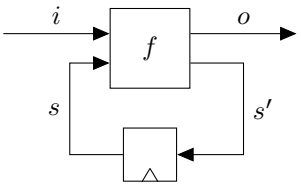

Figure 1: Mealy machine

This transition function can be seen as a mathematical function, which is applied to the inputs and state at every clock cycle. 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.

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 very different from the mathematical function we started with. For instance in a VHDL process, sequential statements are used to describe parallel hardware. 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 since the level of abstraction is raised and less syntactical overhead is required. Because of these reasons, it is natural to use functional programming languages to design synchronous digital hardware. In [1], [2] we have introduced a modified subset of Haskell, together with a compiler called CλaSH, which is based on the Glasgow Haskell Compiler (GHC).

In this paper we describe a substantial extension to CλaSH which makes it possible to describe components as a transition function together with state. In our implementation, the result remains functional (hence, a "normal" Haskell description), allows for extensions to CλaSH like multiple clock domains and it yields a pleasant notation for port mappings.

In this paper, we describe how to hide the state from the user in function compositions, i.e. the state is part of the function arguments but this is hidden while composing components, by using an *automata arrow* [3]. 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 CλaSH and presenting this using a nontrivial example.

Section II discusses related work and compares this to our own work. Section III will elaborate on CλaSH. Arrows will be shortly discussed in section IV. Section V explains how to deal with the hardware state, when designing synchronous hardware in CλaSH. In section VI, the streaming reduction circuit [1] is introduced as a non-trivial circuit and an implementation using arrows is elaborated upon.

II. Related work

Where the CλaSH compiler takes Haskell code as input, Lava [4] and ForSyDe [5] are domain specific embedded languages defined within Haskell. Both languages are stream processing languages, i.e. they operate on infinite streams. In

stream processing languages, the state of synchronous hardware can be modelled using a delay function. In CλaSH, the delay function is a special case and can be trivially written as a simple transition function. Instead of defining mappings from streams to streams, CλaSH defines a mapping from current input and current state to the next state and output, this mapping 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 [6] 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 verbose than those in CλaSH.

VHDL [7] components are created using component declarations and connected using port maps. In VHDL it is not clear from variable and signal declarations whether these variables and signals will become part of the 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 such as (but not limited to) using functions as function argument or functions returning a function as result are cumbersome in VHDL, functional languages are better suited when high-level abstractions are desired.

In [3], arrows are introduced and circuits using delay functions are taken as an example. In section V, we show that arrows can also be used for functional hardware modelled with Mealy machines whereas examples in [3] do not make the state explicit in the arguments of a function and use a delay function instead. In the examples in [3], only very small hardware designs were explored. We will show it is possible, using CλaSH, to create relatively large hardware designs. In our approach we will use the automata arrow as introduced in [3].

III. CλaSH

Using the CλaSH libraries one can simulate synchronous hardware designs written in Haskell using any Haskell interpreter or compiler. When describing hardware in CλaSH, it is possible to use Haskell choice constructs like if-then-else and pattern matching, higher order functions, etc. It is not trivial to compile all Haskell code to hardware, since not all Haskell constructs have a direct structural counterpart in hardware. For instance, Haskell types like Integer and lists do not have a size that can always be fixed at compile time. Therefore, there is no (direct) translation from such types to hardware, as in hardware the number of bits is fixed.

On the other hand, some types are important when designing hardware, while they are less important when designing software. In software mainly words which are a multiple of 8 bits are used, while in hardware it is common to let the designer choose the number of bits in a word. Furthermore, operations on bits and vectors of bits are crucial for hardware designs.

Several built-in types are available:

Bit	This is a predefined algebraic data type which is either Low or High
Bool	Values of this boolean type can either be True or False. This can, for instance, be used in if-then-else constructs.
Vector	Vectors have a static length, dependent types are used to fix their length.
Index	The index type is used to index elements in a Vector.
(Un)signed	Integer types in CλaSH have a fixed number of bits and can be either signed or unsigned, they will wrap around when an overflow occurs.

Besides these predefined types, tuples and algebraic data types are supported. Together these types are sufficient to describe most synchronous hardware. The way of describing synchronous functional hardware is the topic of the next section.

To translate the extended Haskell subset supported by CλaSH to a description of physical hardware, VHDL is used as intermediate language. Although it is certainly possible to directly translate Haskell to a network of hardware components, using VHDL as intermediate language makes it possible to use all existing tools that facilitate hardware design. For instance, a lot of optimisations, power simulations, etc. are enabled this way.

To translate Haskell to VHDL, CλaSH rewrites GHCs internal Core to a normal form using a set of rewrite rules [8]. This normal form is very close to a netlist, the actual transformation from Core in the normal form to VHDL is more or less trivial. Examples of these transformations are β-reduction and η-expansion, but there are also transformations to unfold higher-order functions to first order functions by repeated application of the appropriate function. In a similar fashion, CλaSH recognises the automata arrow in Core and knows how to extract state from these arrows.

IV. Arrows

This section briefly discusses arrows in CλaSH and Haskell, enough to understand the remainder of this article. For an elaborate discussion we refer to [9] or [3] which both contain an excellent introduction to arrows in Haskell.

Arrows give an uniform interface for composition, and is a well-known abstraction in the functional programming community. Every arrow is an instance of the type class $Arrow$. Type classes in CλaSH can be compared to interfaces in Java [1]. For every arrow, sequential and parallel

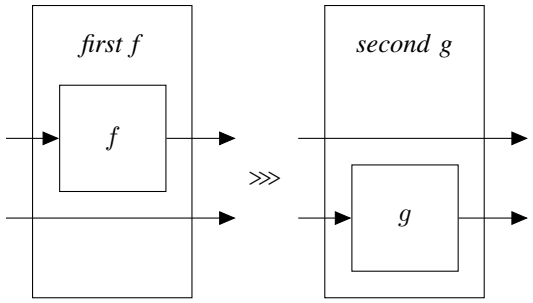

Figure 2: Composition of arrows using first, second and >>>

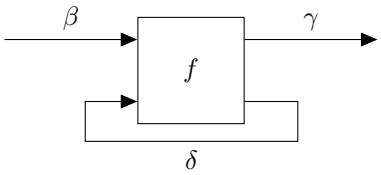

Figure 3: An arrow loop

composition can be defined. To create an arrow from a regular function we use the function $pure$. The function >>> 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 process is somewhat more elaborate and uses the functions *first* and $second$. The function *first* takes an arrow with input type β and output type γ and creates a new arrow with input and output types respectively (β,δ) and (γ,δ). The arrow that is used as the argument of *first* is only applied to the first element of the tuple (β,δ), 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. The expression $(first\ f) \ggg (second\ g)$ thus forms the parallel composition of the arrows f and g. Figure 2 shows this parallel composition graphically. The type class $Arrow$ is defined as in Listing 1.

```
class Arrow α where                                  1
  pure :: (β → γ) → α β γ                            2
  (>>>) :: α β γ → α γ δ → α β δ                     3
  first  :: α β γ → α (β,δ) (γ,δ)                    4
```

Listing 1: The arrow type class

Using these operators all parallel and sequential structures can be created. To create feedback loops (Figure 3) another type class, called *ArrowLoop*, is required. This type class is defined as in Listing 2.

```
class Arrow α ⇒ ArrowLoop α where                    1
  loop :: α (β,δ) (γ,δ) → α β γ                      2
```

Listing 2: The ArrowLoop type class

To model hardware, we use one specific arrow, namely the automata arrow. The automata arrow, described in [3] and shown in Listing 3, takes an input and produces an output together with a new automata arrow. The functions *pure*, >>>, *first* and *loop* are defined in [3] for *Comp*.

```
newtype Comp i o = C {                               1
   exec :: i → (o, Comp i o)                         2
 }                                                   3
```

Listing 3: Definition of the Automata Arrow

We will use this functionality of producing a new arrow to store the state. In that case the arrow is a function that contains the current state as a constant. In the next section we define a function that lifts a transition function to an automata arrow. The reason why we use the automata arrow together with this lifting function, instead of the circuit arrow from [3], is that our approach has a 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.

V. State

In a Mealy machine, the transition function maps the input and the current state to output and a new state, as was explained in section I. In CλaSH, the state is an argument of the transition function. All transition functions in CλaSH have the following type:

$$state \rightarrow input \rightarrow (state, output)$$

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 the types that were described in the previous section.

The automata arrow is used to hide state inside the arrow. Instead of using the transition function, a new function of type $Comp$ is defined which maps input to an output and a new function of type $Comp$. The function of type $Comp$ is an automata arrow and contains the state. The type of the state cannot be observed from the type $Comp$. Because of this, the state is not required as an argument to this function and is effectively hidden. A mapping from a transition function to an automata arrow is defined using the lifting function ⇑ in listing 4. This lifting function is recognised by CλaSH in Core expressions and is used to identify state.

```
(⇑) :: (s → i → (s, o)) → s → Comp i o               1
(⇑) f init = C applyS                                2
  where                                              3
    applyS = λi → let (s, o) = f init i              4
                  in (o, f ⇑ s)                      5
```

Listing 4: Lifting a transition function to a Component

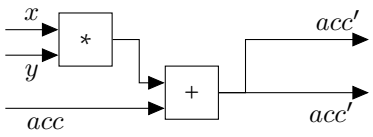

Figure 4: Multiply Accumulate transition function

This function requires the transition function and an initial state as 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 can not be implemented in actual hardware, the CλaSH compiler recognises the arrow, extracts the state and creates registers that represent the state.

The multiply accumulate (MAC) will be used as an example. The accumulator adds the product of the inputs to its state and uses the result as new state and also sends it to the output. The corresponding transition function is visualised in figure 4 and defined in listing 5.

```
mac acc (x, y) = (acc', acc')                1
  where                                      2
    acc' = acc + x * y                       3
```

Listing 5: Multiply accumulate

To simulate synchronous hardware described by transition functions, the *simulate* function is used, as defined in listing 6. The simulate function receives the transition function f and initial state s as argument, together with a list of input values $(x : xs)$.

```
simulate f s (x : xs) = y : simulate f s' xs   1
  where                                        2
    (s', y) = f s x                            3
```

Listing 6: The simulate function

When the circuit is *lifted* to an arrow, the initial state is an argument to the lifting function ⇑, which hides the state inside the function. To lift the function *mac* to the arrow *macA* using the initial state 0, the following definition is used:

```
macA = mac ⇑ 0
```

Because the state is now hidden in an arrow, the simulation function for arrows differs slightly from the simulation function described in Listing 6: instead of using the new state (s') in the recursive call of *simulate*, we would use a new function f'.

For the composition of arrows in CλaSH we introduce a slightly different notation as originally introduced in [10]. Using this component composition notation, indicated by γ, the arrows are automatically composed using *first*, ⋙ and *pure*. In this notation, first the inputs and outputs of the component are described, followed by a **where** statement after which the subcomponents are instantiated. The *loop*

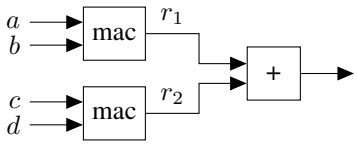

Figure 5: The arrow *macsum*

function is automatically used to compose arrows which require feedback. In listing 7 it is shown how to define a circuit (using the component composition notation) that contains two MACs, of which the results are added to produce an output. This arrow is visualised in figure 5.

```
macsum = γ (a, b, c, d) ⇒ (r1 + r2)          1
  where                                      2
    r1  ⇐  mac ⇑ 0  ⇐  (a, b)                3
    r2  ⇐  mac ⇑ 0  ⇐  (c, d)                4
```

Listing 7: Composing MAC components

In this example, the instantiations of the two components appear at lines 3 and 4. 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 component instantiations. The arrow *macsum* receives the inputs (a, b, c, d) and returns $r_1 + r_2$ as output (line 1). Note that arbitrarily deep nesting of components defined using arrows is possible, as the γ-notation results in a Comp arrow which again can be used for composition.

Using transition functions it becomes easy to define a *delay* function, which will be translated to a register.

```
delay s0 i = (i, s0)
```

Note that the *delay* function is polymorph, hence values of any type can be passed to this function. One example where this can be useful is in the definition of pipelines. Consider components $C_1, \ldots, C_N$, where the input ports of C_i (for $i > 1$) are connected to the output ports of C_{i-1} using the ⋙ operator defined for arrows. In CλaSH this can be written as

$$C_1 \ggg C_2 \ggg \cdots \ggg C_N$$

Suppose this circuit has to be pipelined by inserting registers between the components. In CλaSH this can be written as

$$C_1 \ggg delay \Uparrow s_2 \ggg C_2 \ggg \cdots \ggg delay \Uparrow s_N \ggg C_N$$

Two big advantages of CλaSH are shown here. Due to polymorphism, the delay function and compositions can always be used as long as the types match.

Parameterisation is possible when using the Comp arrow, for instance as in listing 8. Listing 8 shows how a complex

```
complexAddition f s0                                          1
  = γ (Cpx a1 b1, Cpx a2 b2) ⇒ Cpx a b                        2
  where                                                       3
    a  ⇐  f ⇑ s0  ⇐  (a1, a2)                                 4
    b  ⇐  f ⇑ s0  ⇐  (b1, b2)                                 5
```

Listing 8: Parameterisation

adder can be defined using a given adder. Note that it is possible to, for instance, instantiate the complex adder with a certain floating point adder but also with an integer adder due to the support for polymorphism in CλaSH. The argument f is a function that describes an adder, the argument s_0 the initial state of f. This makes it possible to replace the adder without changing the code of the complex multiplication.

Note that if the floating point adder has a certain delay due to the pipeline, the composition will have the same delay. In the next section another example of parameterisation is given.

VI. Reduction Circuit

The small example in the previous section does not yet show the full strength of CλaSH, nor why arrows are useful. A more elaborate example of a circuit is the streaming reduction circuit [11], which is introduced below.

When solving the matrix equation $A\mathbf{x} = \mathbf{b}$ for a big sparse positive definite matrix A, the conjugate gradient algorithm is often used. The conjugate gradient algorithm can be 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×V). When calculating a matrix-vector multiplication, dot products can be used to calculate the elements of the result vector. For an SM×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 binary pipelined floating point adder is used to calculate the additions. Pipelining enables a higher clock frequency at the cost of an increased delay (in clock cycles). 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. In figure 6 pipelining is demonstrated, where it is shown how values propagate through the pipeline. During the first clock cycle the calculation $a+b$ enters the pipeline, the next cycle the calculation $c + d$ enters the pipeline, etc. Note that in the input two values enter the pipeline, whereas inside the pipeline the values are step by step added. For brevity, in our notation we assume the addition takes place immediately, while the result propagates through the pipeline, this leads to an abstract notation for a pipeline. After α clock cycles, where α is the depth of the pipeline,

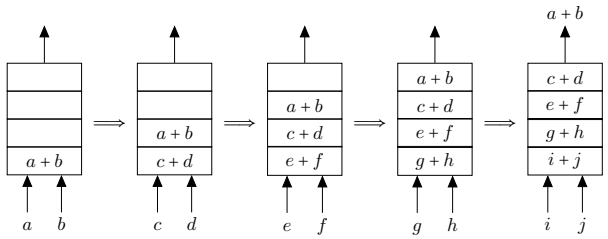

Figure 6: Pipelining, 5 consecutive clock cycles

the results appear at the output of the pipeline. Note that all pipeline stages operate in parallel.

Summing a row of numbers with a pipelined binary adder, as is required for an SM×V, is more complex than summing rows of values with a non-pipelined binary adder. Take for instance a row of three values summed using a pipelined binary adder of 14 stages. 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, hence this third value has to be buffered. Meanwhile, values of other rows might be available for reduction. This illustrates that the pipeline can be scheduled to reduce values of multiple rows simultaneously.

Various circuits which can sum variable length rows of floating point values exist, 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 reduction circuits with different properties are available [11], [12], [13], [14], [15]. Several designs rely on either a minimum or a maximum row length, where some require multiple adders, while others schedule a single floating point adder.

There are two popular methods to deal with complexities caused by pipelining. In the first method, values at the input and partial results at the output of the pipelined adder are placed in a buffer. During a clock cycle there can be multiple values from different rows in the buffer that holds input values and there can be multiple values from different rows in the buffer that holds partial results, a scheduler is used to choose which values will enter the pipeline. It has to be shown that the buffers are bounded, since in hardware the buffers are relatively expensive and have a fixed size. In the other method, it is assumed that rows have a maximum length n, in that case an adder with at least n input ports is created using multiple binary adders. The drawback is that this approach is less generic and requires a lot of parallel adders, such a design can become too big if one has to deal with long rows.

In [11] our *streaming reduction circuit* is introduced, together with an algorithm to determine the inputs for the pipeline and a proof to show that the defined buffer sizes

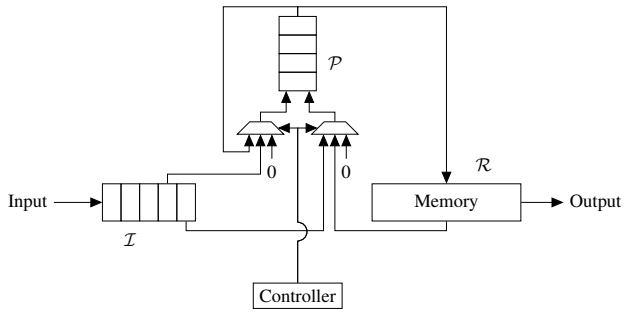

Figure 7: Streaming Reduction circuit

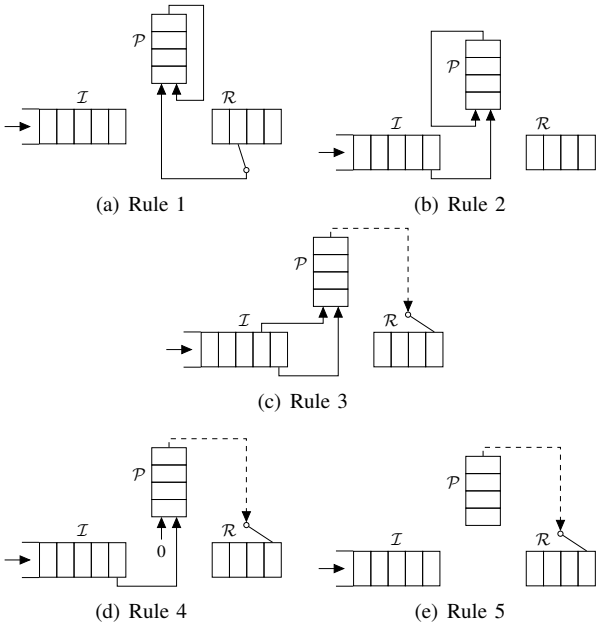

Figure 8: Rules.

are sufficient. In the streaming reduction circuit, values appear sequentially at the input, one value at every clock cycle. These values are a two tuple consisting of a floating point value (which has to be added) together with a row index which uniquely identifies the rows of values which have to be accumulated. The streaming reduction circuit uses a single floating point adder with α pipeline stages. However, this adder can in general be replaced by any binary commutative and associative operator. This pipelined operator is denoted by $\mathcal{P}$.

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 $\mathcal{R}$). 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 $\mathcal{I}$). To determine if either values from the input buffer, from the end of the pipeline and/or from the partial result buffer will be used, five rules are checked. The rules can determine which values to use, i.e. the top two values from $\mathcal{I}$ (denoted as $\mathcal{I}_1$ and $\mathcal{I}_2$), the output of the the adder pipeline (denoted as $\mathcal{P}_\alpha$) or values from $\mathcal{R}$.

The five rules, in descending order of priority, are:

1) If there is a value available in $\mathcal{R}$ with the same row index as $\mathcal{P}_\alpha$, then this value from $\mathcal{R}$ enters the pipeline together with $\mathcal{P}_\alpha$.
2) If $\mathcal{I}_1$ has the same index as $\mathcal{P}_\alpha$, then $\mathcal{I}_1$ and $\mathcal{P}_\alpha$ enter the pipeline.
3) If there are at least two elements in $\mathcal{I}$, and $\mathcal{I}_1$ and $\mathcal{I}_2$ have the same index, then they enter the pipeline.
4) If there are at least two elements in $\mathcal{I}$, but $\mathcal{I}_1$ and $\mathcal{I}_2$ have different indexes, then $\mathcal{I}_1$ 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 $\mathcal{I}$, no elements enter the pipeline.

The rules are schematically shown in figure 8. The datapath of the reduction circuit is shown in figure 7. The components $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{P}$, together with the controller 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 less bits than the row index, as the number of rows within the reduction circuit is bounded.

Although figure 7 makes it clear how data flows through the reduction circuit, it neglects the *control signals*. Figure 9 shows the entire circuit including control signals. The controller, denoted by $\mathcal{C}$, checks which rule has to be executed. The discriminators are assigned by $\mathcal{D}$.

All components of the streaming reduction circuit are modelled as a function in CλaSH. Taking the input buffer ($\mathcal{I}$) as an example, which has the type

$$\mathcal{I} :: ISt \rightarrow (DVal, Index3) \rightarrow (ISt, (DVal, DVal))$$

The type indicates that it has two inputs and one result, the first input is the current state, and the second input is a tuple containing the signals coming from other components. The output is a tuple which consists of the new/updated state and the output signals for other other components.

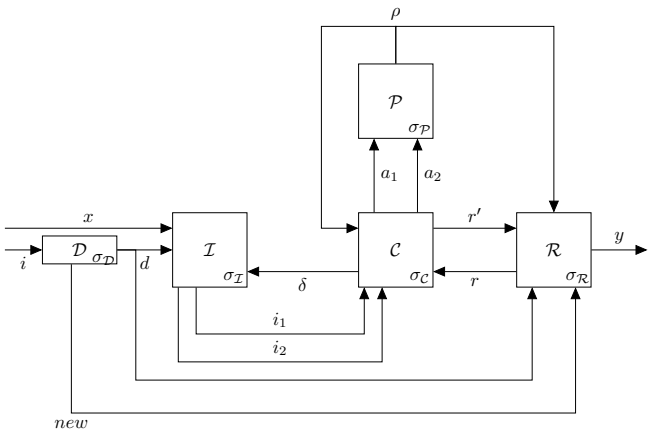

Figure 9: Reduction circuit signals

A value of type $DVal$ consists of a floating point value and its discriminator; the discriminator is used to determine to what row the values belong to. The signals coming from other components are thus the value (of type $DVal$) that has to be placed in the buffer, and a second signal indicating how many values will be consumed from the buffer. The second signal is an index of type $Index3$, an index with an exclusive upper bound of 3.

The state of the input buffer is an algebraic datatype (with constructor ISt) that contains a vector and two indices; together used to implement the FIFO as a circular buffer. The result of the function $\mathcal{I}$ is the tuple containing the new state, and the two values (of type $DVal$) that are at the top of the FIFO. In a similar fashion, the other components that are shown in figure 9 are written as a Haskell function. We connect these components to form the complete reduction circuit by using the code shown in Listing 9.

```
rc P = γ (x, i) → y                                                    1
  where                                                                2
    (new, d)        ⇐  D ⇑ D0  ⇐  i                                    3
    (i1, i2)        ⇐  I ⇑ I0  ⇐  (x, d, δ)                            4
    ρ               ⇐  P ⇑ P0  ⇐  (a1, a2)                             5
    (r, y)          ⇐  R ⇑ R0  ⇐  (new, d, ρ, r')                      6
    (a1, a2, δ, r') ⇐  C ⇑ C0  ⇐  (i1, i2, ρ, r)                       7
```

Listing 9: Reduction circuit with arrows

In listing 9 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 the component and its initial state belong together, it is natural to define the initial state where the component is instantiated. The floating point operator $\mathcal{P}$ is passed as a parameter to the reduction circuit, making the implementation generic for all kinds of pipelined reduction operations.

Table I: Design characteristics Reduction circuit

	CλaSH	**VHDL**
CLB Slices & LUT	4076	4734
Dffs or Latches	2467	2810
Operating Frequency (MHz)	159	171

When arrows are used to implement the reduction circuit, an *ArrowLoop* is required. In line 1 of listing 9 this is automatically enabled using the γ syntax. The component (or function) $\mathcal{P}$ requires a result from $\mathcal{C}$, while $\mathcal{C}$ requires a result from $\mathcal{P}$, i.e. the functions depend on each other's results. In figure 9, this is shown using the signals δ, i_1 and i_2. These same signals are shown in listing 9. Because the result (ρ) produced by the pipeline ($\mathcal{P}$) does not immediately depend on the signals (a_1, a_2) sent by the controller ($\mathcal{C}$) during the same clock cycle, Haskell's lazy evaluation will make sure this functional dependency will not be a problem in simulation since the data which is required is already available in the state and does not depend on the input. For exactly the same reason, this will not be a problem in the actual hardware produced using CλaSH.

Table I displays the design characteristics of both the CλaSH design and a hand-optimized VHDL design where the same global design decisions and local optimizations were applied to both designs. The figures in the table show that the results are comparable, but we remark that they only give a first impression.

VII. 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 the current state to output and a new state. This can be modelled in a functional language using a single function, called the transition function. The notation of arrows yields both a pleasant notation and a method to hide the state inside the arrow. This abstraction is well-known in the functional programming community, parameterisable and functional.

Our approach was tested by modelling and compiling the streaming reduction circuit, a nontrivial circuit, in CλaSH. From this example, it is clear that it is possible to design nontrivial hardware using Haskell. ArrowLoop is used since loops are often required for digital hardware design. Because such (non-combinational) loops occur frequently in digital designs it is desirable to use lazy functional languages to simulate hardware designs. The γ syntax automatically introduces the *loop* construct in descriptions when a looping dependency is discovered.

Only synchronous hardware is supported by CλaSH. In the future, support for asynchronous hardware will be considered. Further research is required in this direction.

REFERENCES

[1] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards, "CλaSH: Structural descriptions of synchronous hardware using Haskell," in *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France*. USA: IEEE Computer Society, September 2010, pp. 714–721. [Online]. Available: http://eprints.eemcs.utwente.nl/18376/

[2] J. Kuper, C. P. R. Baaij, M. Kooijman, and M. E. T. Gerards, "Exercises in architecture specification using CλaSH," in *Proceedings of Forum on Specification and Design Languages, FDL 2010, Southampton, England*. Gières, France: ECSI Electronic Chips & Systems design Initiative, September 2010, pp. 178–183.

[3] R. Paterson, "Arrows and computation," *The Fun of Programming*, pp. 201–222, 2003.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *Proceedings of the third ACM SIGPLAN international conference on Functional programming - ICFP '98*, 1998, pp. 174–184.

[5] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, 2004.

[6] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing Kansas Lava," November 2009, submitted to The International Symposia on Implementation and Application of Functional Languages (IFL)'09. [Online]. Available: http://ittc.ku.edu/ andygill/papers/kansas-lava-ifl09.pdf

[7] *IEEE Standard 1076-2008 VHDL Language Reference Manual*, 2009.

[8] M. Kooijman, "Haskell as a higher order structural hardware description language," Master's thesis, Univ. of Twente, December 2009. [Online]. Available: http://essay.utwente.nl/59381/

[9] J. Hughes, "Programming with arrows," *Advanced functional programming: 5th international school, AFP 2004, Tartu, Estonia, August 14-21, 2004: revised lectures*, pp. 73–129, 2005.

[10] R. Paterson, "A new notation for arrows," in *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming - ICFP '01*, 2001, pp. 229–240.

[11] M. E. T. Gerards, J. Kuper, A. B. J. Kokkeler, and E. Molenkamp, "Streaming reduction circuit," in *2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*. Los Alamitos: IEEE Computer Society Press, August 2009, pp. 287–292. [Online]. Available: http://eprints.eemcs.utwente.nl/17041/

[12] M. R. Bodnar, J. P. Durbano, J. R. Humphrey, P. F. Curt, and D. W. Prather, "FPGA-based, floating-point reduction operations," in *MATH'06: Proceedings of the 10th WSEAS International Conference on APPLIED MATHEMATICS*. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2006, pp. 5–9.

[13] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *2008 International Conference on Field-Programmable Technology*, 2008, pp. 33–40.

[14] K. K. Nagar, Y. Zhang, and J. D. Bakos, "An integrated reduction technique for a double precision accumulator," in *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications - HPRCTA '09*, 2009, pp. 11–18.

[15] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, pp. 1377–1392, 2007.