CλaSH: Structural Descriptions of Synchronous Hardware using Haskell

Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, Marco Gerards
Computer Architecture for Embedded Systems (CAES)
Department of EEMCS, University of Twente
P.O. Box 217, 7500 AE, Enschede, The Netherlands
c.p.r.baaij@utwente.nl, matthijs@stdin.nl, j.kuper@utwente.nl

Abstract—CλaSH is a functional hardware description language that borrows both its syntax and semantics from the functional programming language Haskell. Polymorphism and higher-order functions provide a level of abstraction and generality that allow a circuit designer to describe circuits in a more natural way than possible with the language elements found in the traditional hardware description languages.

Circuit descriptions can be translated to synthesizable VHDL using the prototype CλaSH compiler. As the circuit descriptions, simulation code, and test input are also valid Haskell, complete simulations can be done by a Haskell compiler or interpreter, allowing high-speed simulation and analysis.

I. INTRODUCTION

Hardware description languages (HDLs) have not allowed the productivity of hardware engineers to keep pace with the development of chip technology. While traditional HDLs, like VHDL [1] and Verilog [2], are very good at describing detailed hardware properties such as timing behavior, they are generally cumbersome in expressing the higher-level abstractions needed for today’s large and complex circuit designs. In an attempt to raise the abstraction level of the descriptions, a great number of approaches based on functional languages have been proposed [3]–[10]. The idea of using functional languages for hardware descriptions started in the early 1980s [3], [4], a time which also saw the birth of the currently popular HDLs, such as VHDL. Functional languages are especially well suited to describe hardware because combinational circuits can be directly modeled as mathematical functions and functional languages are very good at describing and composing these functions.

In an attempt to reduce the effort involved with prototyping a new language, such as creating all the required tooling like parsers and type-checkers, many functional HDLs [6]–[9] are embedded as a domain specific language (DSL) within the functional language Haskell [11]. This means that a developer is given a library of Haskell functions and types that together form the language primitives of the DSL. The primitive functions used to describe a circuit do not actually process any signals, they instead compose a large graph (which is usually hidden from the designer). This graph is then further processed by an embedded circuit compiler which can perform e.g. simulation or synthesis. As Haskell’s choice elements (case-expressions, pattern-matching, etc.) are evaluated at the time the graph is being build, they are no longer visible to the embedded compiler that processes the graph. Consequently, it is impossible to capture Haskell’s choice elements within a circuit description when taking the embedded language approach. This does not mean that circuits specified in an embedded language can not contain choice, just that choice elements only exist as functions, e.g. a multiplexer function, and not as syntactic elements of the language itself.

This research uses (a subset of) the Haskell language itself for the purpose of describing hardware. As a result, certain language constructs, like all of Haskell’s choice elements, can now be captured within circuit descriptions. Advanced features of Haskell, such as polymorphic typing and higher-order functions, are also supported.

Where descriptions in a conventional HDL have an explicit clock for the purposes of state updates and synchronicity, the clock is implicit for the descriptions and research presented in this paper. A circuit designer describes the behavior of the hardware between clock cycles, as a transition from the current state to the next. Many functional HDLs model signals as a stream of values over time, state is then modeled as a delay on this stream of values. Descriptions presented in this research make the current state an additional input and the updated state a part of their output. This abstraction of state and time limits the descriptions to synchronous hardware. However, work is in progress to add an abstraction mechanism that allows the modeling of asynchronous and multi-clock systems.

Likewise as with the traditional HDLs, descriptions made in a functional HDL must eventually be converted into a netlist. This research also features a prototype compiler, which has the same name as the language: CλaSH1 (pronounced: clash). This compiler converts the Haskell code to equivalently behaving synthesizable VHDL code, ready to be converted to an actual netlist format by a standard VHDL synthesis tool.

To the best knowledge of the authors, CλaSH is the only (functional) HDL that allows circuit specification to be written in a very concise way and at the same time support such advanced features as polymorphic typing, user-defined higher-order functions and pattern matching.

1CλaSH: CAES Language for Synchronous Hardware.

Supported through the FP7 project: S(o)OS (248465)
The next section will describe the language elements of CλaSH, and Section III gives a high-level overview of the CλaSH compiler. Section IV discusses two use-cases, a FIR filter, and a higher-order CPU design. The related work section (Section V) is placed towards the end, as the features of CλaSH should be presented before comparing CλaSH to existing (functional) HDLs. Conclusions are presented in Section VI, and future work is discussed in Section VII.

II. HARDWARE DESCRIPTION IN HASKELL

This section describes the basic language elements of CλaSH and the support of these elements within the CλaSH compiler. In various subsections, the relation between the language elements and their eventual netlist representation is also highlighted.

A. Function application

Two basic elements of a functional program are functions and function application. These have a single obvious translation to a netlist format: 1) every function is translated to a component, 2) every function argument is translated to an input port, 3) the result value of a function is translated to an output port, and 4) function applications are translated to component instantiations. The result value can have a composite type (such as a tuple), so the fact that a function has just a single result value does not pose any limitation. The actual arguments of a function application are assigned to signals, which are then mapped to the corresponding input ports of the component. The output port of the function is also mapped to a signal, which is used as the result of the application itself. Since every function generates its own component, the hierarchy of function calls is reflected in the final netlist.

The short example below (1) gives a demonstration of the conciseness that can be achieved with CλaSH when compared to other (more traditional) HDLs. The example is a combinational multiply-accumulate circuit that works for any word length (this type of polymorphism will be further elaborated in Section II-D). The corresponding netlist is depicted in Figure 1.

\[
\text{mac } a \ b \ c = \text{add} (\text{mul } a \ b) \ c \quad (1)
\]

The use of a composite result value is demonstrated in the next example (2), where the multiply-accumulate circuit returns not only the accumulation result, but also the intermediate multiplication result (see Figure 2, where the double arrow suggests the composite output).

\[
\text{mac } a \ b \ c = (z, \text{add } z \ c) \quad \text{where} \quad z = \text{mul } a \ b \quad (2)
\]

B. Choice

In Haskell, choice can be achieved by a large set of syntactic elements, consisting of: case expressions, if – then – else expressions, pattern matching, and guards. The most general of these are the case expressions (if expressions can be directly translated to case expressions). When transforming a CλaSH description to a netlist, a case expression is translated to a multiplexer. The control value of the case expression is fed into a number of comparators, and their combined output forms the selection port of the multiplexer. The result of each alternative in the case expression is linked to the corresponding input port of the multiplexer.

A code example (3) that uses a case expression and if – then – else expressions is shown below. The function counts up or down depending on the direction variable, and has a bound variable that determines both the upper bound and wrap-around point of the counter. The direction variable is of the following, user-defined, enumeration datatype:

\[
\text{data Direction} = \text{Up} \mid \text{Down}
\]

The naive netlist corresponding to this example is depicted in Figure 3. Note that the direction variable is only compared to Up, as an inequality immediately implies that direction is Down (as derived by the compiler).

\[
\text{counter bound direction } x = \text{case direction of} \\
\text{Up} \rightarrow \text{if } x < \text{bound then} \\
\quad x + 1 \quad \text{else} \\
\text{Down} \rightarrow \text{if } x > 0 \quad \text{then} \\
\quad x - 1 \quad \text{else} \quad \text{bound} \quad (3)
\]

A user-friendly and also powerful form of choice that is not found in the traditional HDLs is pattern matching. A function can be defined in multiple clauses, where each clause corresponds to a pattern. When an argument matches a pattern, the corresponding clause will be used. Expressions can also contain guards, where the expression is only executed if the guard evaluates to true, and continues with the next clause if the guard evaluates to false. Like if – then – else expressions, pattern matching and guards have a (straightforward) translation to case expressions and can as such be mapped to multiplexers. A second version (4) of the earlier example, now using both pattern matching and guards, can be seen on the next page. The guard is the expression that follows the
vertical bar (|) and precedes the assignment operator (=). The otherwise guards always evaluate to true.

The second version corresponds to the same naive netlist representation (Figure 3) as the earlier example.

\[
\begin{align*}
\text{counter bound Up} & \quad x \mid x < \text{bound} = x + 1 \\
& \quad \text{otherwise} = 0 \\
\text{counter bound Down} & \quad x \mid x > 0 = x - 1 \\
& \quad \text{otherwise} = \text{bound}
\end{align*}
\] (4)

C. Types

Haskell is a statically-typed language, meaning that the type of a variable or function is determined at compile-time. Not all of Haskell’s typing constructs have a clear translation to hardware, therefore this section only deals with the types that do have a clear correspondence to hardware. The translatable types are divided into two categories: built-in types and user-defined types. Built-in types are those types for which a fixed translation is defined within the CλaSH compiler. The CλaSH compiler has generic translation rules to translate the user-defined types, which are described later on.

Type annotations (entities in VHDL) are optional, since the CλaSH compiler can derive them when the top-level function is annotated with its type.

1) Built-in types: The following types have fixed translations defined within the CλaSH compiler:

- **Bit**: the most basic type available. It can have two values: Low or High.
- **Bool**: this is a basic logic type. It can have two values: True or False. Supporting the Bool type is required in order to support the if – then – else expression.
- **Signed, Unsigned**: these are types to represent integers, and both are parametrizable in their size. The overflow behavior of the numeric operators defined for these types is wrap-around.
- **Vector**: this type can contain elements of any type and has a static length. The Vector type constructor takes two arguments: the length of the vector and the type of the elements contained in it. The short-hand notation used for the vector type in the rest of paper is: [a | n], where a is the element type, and n is the length of the vector.

Index: the main purpose of the Index type is to be used as an index into a Vector, and has an integer range from zero to a specified upper bound. If a value of this type exceeds either bounds, an error will be thrown during simulation.

2) User-defined types: A designer may define a completely new type by an algebraic datatype declaration using the data keyword. Type synonyms can be introduced using the type keyword. Type synonyms do not need any particular translation, as a synonym will use the same representation as the original type.

Algebraic datatypes can be categorized as follows:

Single constructor: datatypes with a single constructor with one or more fields allow values to be packed together in a record-like structure. Haskell’s built-in tuple types are also defined as single constructor algebraic types (using some syntactic sugar). An example of a single constructor type with multiple fields is the following pair of integers:

\[
\text{data IntPair} = \text{Pair} \text{ Int Int}
\]

Multiple constructors, No fields: datatypes with multiple constructors, but without any fields are enumeration types. An example of an enumeration type definition is:

\[
\text{data TrafficLight} = \text{Red} \mid \text{Orange} \mid \text{Green}
\]

Multiple constructors with fields: datatypes with multiple constructors, where at least one of these constructors has one or more fields are currently not supported. Additional research is required to optimize the overlap of fields belonging to the different constructors.

D. Polymorphism

A powerful feature of some programming languages is polymorphism, it allows a function to handle values of different data types in a uniform way. Haskell supports parametric polymorphism, meaning that functions can be written without mentioning specific types, and that those functions can be used for arbitrary types.

As an example of a parametric polymorphic function, consider the type of the first function, which returns the first element of a tuple:²

\[
\text{first} :: (a, b) \rightarrow a
\]

This type is parameterized in a and b, which can both represent any type that is supported by the CλaSH compiler. This means that first works for any tuple, regardless of what elements it contains. This kind of polymorphism is extremely useful in hardware designs, for example when routing signals without knowing their exact type, or specifying vector operations that work on vectors of any length and element type. Polymorphism also plays an important role in most higher order functions, as will be shown in the next subsection.

²The :: operator is used to annotate a function with its type, where the annotation z :: x → y indicates that z is a function with an argument of type x and a result of type y.
Another type of polymorphism is \textit{ad-hoc polymorphism}, which refers to functions that can be applied to arguments of a limited set to types. Furthermore, how such functions work may depend on the type of their arguments. For instance, multiplication only works for numeric types, and it works differently for e.g. integers and complex numbers.

In Haskell, ad-hoc polymorphism is achieved through the use of \textit{type classes}, where a class definition provides the general interface of a function, and class \textit{instances} define the functionality for the specific types. For example, all numeric operators are gathered in the \textit{Num} class, so every type that wants to use those operators must be made an instance of \textit{Num}.

By prefixing a type signature with class constraints, the constrained type parameters are forced to belong to that type class. For example, the arguments of the \textit{add} function must belong to the \textit{Num} type class because the \textit{add} function adds them with the (+) operator:

\[
\text{add :: Num } a \Rightarrow a \rightarrow a \rightarrow a
\]
\[
\text{add } a \ b \Rightarrow a + b
\]

\text{C abduction supports both parametric polymorphism and ad-hoc polymorphism. A circuit designer can specify his own type classes and corresponding instances. The C abduction compiler will infer the type of every polymorphic argument depending on how the function is applied. There is however one constraint: the top level function that is being translated cannot have polymorphic arguments. The arguments of the top-level cannot be polymorphic as there is no way to infer the specific types of the arguments.}

With regard to the built-in types, it should be noted that members of some of the standard Haskell type classes are supported as built-in functions. These include: the numeral operators of \textit{Num}, the equality operators of \textit{Eq}, and the comparison (order) operators of \textit{Ord}.

\text{E. Higher-order functions & values}

Another powerful abstraction mechanism in functional languages is the concept of \textit{functions as a first class value} and \textit{higher-order functions}. These concepts allow a function to be treated as a value and be passed around, even as the argument of another function. The following example clarifies this concept:

\[
\text{negateVector } xs = \text{map not } xs
\]

The code above defines the \textit{negateVector} function, which takes a vector of booleans, \textit{xs}, and returns a vector where all the values are negated. It achieves this by calling the \textit{map} function, and passing it another \textit{function}, boolean negation, and the vector of booleans, \textit{xs}. The \textit{map} function applies the negation function to all the elements in the vector.

The \textit{map} function is called a higher-order function, since it takes another function as an argument. Also note that \textit{map} is again a parametric polymorphic function: it does not pose any constraints on the type of the input vector, other than that its elements must have the same type as the first argument of the function passed to \textit{map}. The element type of the resulting vector is equal to the return type of the function passed, which need not necessarily be the same as the element type of the input vector. All of these characteristics can be inferred from the type signature of \textit{map}:

\[
\text{map :: (a } \rightarrow \text{ b) } \rightarrow [a \mid n] \rightarrow [b \mid n]
\]

In Haskell, there are two more ways to obtain a function-based value: partial application and lambda abstraction. Partial application means that a function that takes multiple arguments can be applied to a single argument, and the result will again be a function, but takes one argument less. As an example, consider the following expression, that adds one to every element of a vector:

\[
\text{map (add 1) } xs
\]

Here, the expression \textit{(add 1)} is the partial application of the addition function to the value 1, which is again a function that adds 1 to its \textit{(next)} argument.

A lambda expression allows a designer to introduce a function in any expression without first defining that function. Consider the following expression, which again adds 1 to every element of a vector:

\[
\text{map (λx \rightarrow x + 1) } xs
\]

Finally, not only built-in functions can have higher-order arguments (such as the \textit{map} function), but any function defined in C abduction may have functions as arguments. This allows the circuit designer to apply a large amount of code reuse. The only exception is again the top-level function: if a function-typed argument is not instantiated with an actual function, no hardware can be generated.

An example of a common circuit where higher-order functions and partial application lead to a very concise and natural description is a crossbar. The code (9) for this example can be seen below:

\[
\text{crossbar inputs selects } = \text{map (max inputs) selects where}
\]
\[
\text{max inp } x = (\text{inp}!x)
\]

The \textit{crossbar} function selects those values from \textit{inputs} that are indicated by the indexes in the vector \textit{selects}. The crossbar is polymorphic in the width of the input (defined by the length of \textit{inputs}), the width of the output (defined by the length of \textit{selects}), and the signal type (defined by the element type of \textit{inputs}). The type-checker can also automatically infer that \textit{selects} is a vector of \textit{Index} values due to the use of the vector indexing operator \((\cdot)!\).

\text{F. State}

In a stateful design, the outputs depend on the history of the inputs, or the state. State is usually stored in registers, which retain their value during a clock cycle.

An important property in Haskell, and in many other functional languages, is \textit{purity}. A function is said to be \textit{pure} if it satisfies two conditions: 1) given the same arguments twice, it should return the same value in both cases, and 2) that
the function has no observable side-effects. Pure functions are a perfect match for combinational circuits, where the output solely depends on the inputs. When a circuit has state however, it can no longer be described by a pure function. CλaSH deals with the concept of state by making the current state an additional argument of the function, and the updated state part of the result. In this sense the descriptions made in CλaSH are the combinational parts of a Mealy machine.

A simple example is adding an accumulator register to the earlier multiply-accumulate circuit, of which the resulting netlist can be seen in Figure 4:

\[
\text{mac}S (\text{State } c) (a, b) = (\text{State } c', c') \\
\text{where} \\
c' = \text{mac } a \ b \ c
\]  

(10)

Note that the \text{mac}S function returns both the new state \((\text{State } c')\) and the value of the output port \((c')\). The State wrapper indicates which arguments are part of the current state, and what part of the output is part of the updated state. This aspect will also be reflected in the type signature of the function. Abstracting the state of a circuit in this way makes it very explicit: which variables are part of the state is the function. Abstracting the state of a circuit in this way is only an \text{SH} phase, exhaustively applies a set of meaning-preserving transformations on the \text{Core} description until this description is in a normal form. This set of transformations includes transformations typically found in reduction systems and lambda calculus, such as β-reduction and η-expansion. It also includes transformations that are responsible for the specialization of higher-order functions to ‘regular’ first-order functions, and specializing polymorphic types to concrete types.

The complete simulation can be compiled to an executable binary by a Haskell compiler, or executed in a Haskell interpreter. Both simulation paths require less effort from a circuit designer than first translating the description to VHDL and then running a VHDL simulation; it is also very likely that both simulation paths are much faster.

III. THE C\text\{a\}SH COMPILER

The prototype C\text\{a\}SH compiler translates descriptions made in the C\text\{a\}SH language as described in the previous section to synthesizable VHDL.

The Glasgow Haskell Compiler (GHC) [12] is an open source Haskell compiler that also provides a high level API to most of its internals. Furthermore, it provides several parts of the prototype compiler for free, such as the parser, the semantics checker, and the type checker. These parts together form the front-end of the prototype compiler pipeline, as seen in Figure 5.

The output of the GHC front-end consists of the translation of the original Haskell description to Core [13], which is a small typed functional language. This Core language is relatively easy to process compared to the larger Haskell language. A description in Core can still contain elements which have no direct translation to hardware, such as polymorphic types and function-valued arguments. Such a description needs to be transformed to a normal form, which corresponds directly to hardware. The second stage of the compiler, the normalization phase, exhaustively applies a set of meaning-preserving transformations on the Core description until this description is in a normal form. This set of transformations includes transformations typically found in reduction systems and lambda calculus, such as β-reduction and η-expansion. It also includes transformations that are responsible for the specialization of higher-order functions to ‘regular’ first-order functions, and specializing polymorphic types to concrete types.

The final step in the compiler pipeline is the translation to a VHDL netlist, which is a straightforward process due to the resemblance of a normalized description and a set of concurrent signal assignments. The end-product of the C\text\{a\}SH compiler is called a VHDL netlist as the result resembles an actual netlist description, and the fact that it is VHDL is only an implementation detail; e.g., the output could have been Verilog or even EDIF. For verification purposes of the generated VHDL, the compiler also creates a test bench and corresponding input, allowing a developer to compare the external behavior of the VHDL netlist against the original C\text\{a\}SH design.
A. FIR Filter

An example of a common hardware design where the relation between functional languages and mathematical functions, combined with the use of higher-order functions leads to a very natural description is a FIR filter:

\[
y_t = \sum_{i=0}^{n-1} x_{t-i} \cdot h_i
\]  

(13)

A FIR filter multiplies fixed constants \(h\) with the current and a few previous input samples \(x\). Each of these multiplications are summed, to produce the result at time \(t\). The equation of a FIR filter is equivalent to the equation of the dot-product of two vectors, which is shown below:

\[
\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i \cdot b_i
\]

(14)

The equation for the dot-product is easily and directly implemented using higher-order functions:

\[
\text{as} \cdot \text{bs} = \text{fold} (+) (\text{zipWith} (*) \text{as} \text{bs})
\]

(15)

The zipWith function is very similar to the map function seen earlier: It takes a function, two vectors, and then applies the function to each of the elements in the two vectors pairwise (e.g., zipWith \((*)\) \([1,2] [3,4]\) becomes \([1*3,2*4]\)).

The fold function takes a binary function, a single vector, and applies the function to the first two elements of the vector. It then applies the function to the result of the first application and the next element in the vector. This continues until the end of the vector is reached. The result of the fold function is the result of the last application. It is obvious that the zipWith \((*)\) function is pairwise multiplication and that the fold \((+)\) function is summation. The complete definition of the FIR filter in CAaSH is:

\[
\text{fir} \left(\text{State} \left(\mathbf{x}, \mathbf{h}\right)\right) x =
\text{(State} (\text{shiftInto} x \mathbf{x}, \mathbf{h}), (x \triangleright \mathbf{x}) \cdot \mathbf{h})
\]

(16)

where the vector \(\mathbf{x}\) contains the previous input samples, the vector \(\mathbf{h}\) contains the FIR coefficients, and \(x\) is the current input sample. The concatenate operator \((\triangleright)\) creates a new vector by placing the current sample \((x)\) in front of the previous samples vector \((\mathbf{x})\). The code for the shiftInto function, that adds the new input sample \((x)\) to the list of previous input samples \((\mathbf{x})\) and removes the oldest sample, is shown below:

\[
\text{shiftInto} x \mathbf{x} = x \triangleright \text{init} \; \mathbf{x}
\]

(17)

where the init function returns all but the last element of a vector. The resulting netlist of a 4-taps FIR filter, created by specializing the vectors of the FIR code to a length of 4, is depicted in Figure 6.

B. Higher-order CPU

This section discusses a somewhat more elaborate example in which user-defined higher-order function, partial application, lambda expressions, and pattern matching are exploited. The example concerns a CPU which consists of four function units, \(f_u_0, \ldots, f_u_3\), (see Figure 7) that each perform some binary operation.

Every function unit has seven data inputs (of type Signed 16), and two address inputs (of type Index 6). The latter two addresses indicate which of the seven data inputs are to be used as operands for the binary operation the function unit performs.

These seven data inputs consist of one external input \(x\), two fixed initialization values \((0\) and \(1\)), and the previous outputs of the four function units. The output of the CPU as a whole is the previous output of \(f_u_3\).

Function units \(f_u_0, f_u_2,\) and \(f_u_3\) can perform a fixed binary operation, whereas \(f_u_0\) has an additional input for an opcode to choose a binary operation out of a few possibilities. Each function unit outputs its result into a register, i.e., the state of the CPU. This state can e.g. be defined as follows:

\[
\text{type } \text{CpuState} = \text{State} \left[\text{Signed} \; 16 \mid 4\right]
\]

i.e., the state consists of a vector of four elements of type Signed 16. The type of the CPU as a whole can now be defined as (Opcode will be defined later):

\[
\text{cpu} :: \text{CpuState}
\]

\[(\text{Signed} \; 16, \text{Opcode}, ([\text{Index} \; 6, \text{Index} \; 6] \mid 4))]

\[(\text{CpuState}, \text{Signed} \; 16)]
Note that this type fits the requirements of the \textit{run} function. Every function unit can be defined by the following higher-order function, $fu$, which takes three arguments: the operation $op$ that the function unit should perform, the seven \textit{inputs}, and the address pair $(a_0, a_1)$. It selects two inputs, based on the addresses, and applies the given operation to them, returning the result:

$$fu \ op \ inputs \ (a_0, a_1) = \ op \ (inputs \ ! a_0) \ (inputs \ ! a_1)$$

(18)

Using partial application we now define:

$$fu_1 = fu \ add$$

$$fu_2 = fu \ sub$$

$$fu_3 = fu \ mul$$

(19)

Note that the types of these functions can be derived from the type of the \textit{cpu} function, thus determining what component instantiations are needed. For example, the function \textit{add} should take two \textit{Signed} 16 values and also deliver a \textit{Signed} 16 value.

In order to define $fu_0$, the \textit{Opcode} type and the \textit{multiop} function that chooses a specific operation given the opcode, are defined first. It is assumed that the binary functions $shift$ (where $shift \ a \ b$ shifts $a$ by the number of bits indicated by $b$) and $xor$ (for the bitwise $xor$) exist.

$$\text{data Opcode} = \text{Shift} | \text{Xor} | \text{Equal}$$

$$\text{multiop Shift} = shift$$

$$\text{multiop Xor} = xor$$

$$\text{multiop Equal} = \lambda a \ b \rightarrow \text{if } a == b \ \text{then} \ 1 \ \text{else} \ 0$$

(20)

Note that the result of \textit{multiop} is a binary function from two \textit{Signed} 16 values into one \textit{Signed} 16 value; this is supported by \texttt{C\La\La}SH. The complete definition of $fu_0$, which takes an opcode as additional argument, is:

$$fu_0 \ c \ = \ fu \ (\text{multiop} \ c)$$

(21)

The complete definition of the \textit{cpu} function is (note that \textit{addr} contains four address pairs):

$$\text{cpu} \ \text{(State} \ s) \ (x, \ opc, \ addrs) \ = \ \text{(State} \ s', \ out)$$

where

\begin{align*}
inputs &= x \triangleright (0 \triangleright (1 \triangleright s)) \\
\text{where} \\
\ s' &= [fu_0 \ opc \ inputs \ (addr \ ! 0) \\
& \ , fu_1 \ inputs \ (addr \ ! 1) \\
& \ , fu_2 \ inputs \ (addr \ ! 2) \\
& \ , fu_3 \ inputs \ (addr \ ! 3) ] \\
\ out &= \text{last} \ s
\end{align*}

(22)

Due to space restrictions, Figure 7 does not show the internals of each function unit. We remark that \texttt{C\La\La}SH generates e.g. \textit{multiop} as a subcomponent of $fu_0$.

While the CPU has a simple (and maybe not very useful) design, it illustrates some possibilities that \texttt{C\La\La}SH offers and suggests how to write actual designs.

V. RELATED WORK

This section describes the features of existing (functional) hardware description languages and highlights the advantages that \texttt{C\La\La}SH has over existing work.

HML [5] is a hardware modeling language based on the strict functional language ML, and has support for polymorphic types and higher-order functions. There is no direct simulation support for HML, so a description in HML has to be translated to VHDL and the translated description can then be simulated in a VHDL simulator. Certain aspects of HML, such as higher-order functions are however not supported by the VHDL translator [14]. The \texttt{C\La\La}SH compiler on the other hand can correctly translate all of its language constructs.

Like the research presented in this paper, many functional hardware description languages have a foundation in the functional programming language Haskell. Hawk [6] is a hardware modeling language embedded in Haskell and has sequential environments that make it easier to specify stateful computation (by using the \texttt{ST} monad). Hawk specifications can be simulated; to the best knowledge of the authors there is, however, no support for automated circuit synthesis.

The ForSyDe [10] system uses Haskell to specify abstract system models. A designer can model systems using heterogeneous models of computation, which include continuous time, synchronous and untimed models of computation. Using so-called domain interfaces a designer can simulate electronic systems which have both analog and digital parts. ForSyDe has several backends including simulation and automated synthesis, though automated synthesis is restricted to the synchronous model of computation. Although ForSyDe offers higher-order functions and polymorphism, ForSyDe’s choice elements are limited to if \texttt{– then} \texttt{– else} and case expressions. ForSyDe’s explicit conversions, where functions have to be wrapped in processes and processes have to be wrapped in systems, combined with the explicit instantiations of components, also makes ForSyDe far more verbose than \texttt{C\La\La}SH.

Lava [7], [9] is a HDL embedded in Haskell which focuses on the structural representation of hardware. Like \texttt{C\La\La}SH, Lava has support for polymorphic types and higher-order functions. Besides support for simulation and circuit synthesis, Lava descriptions can be interfaced with formal method tools for formal verification. As discussed in the introduction, taking the embedded language approach does not allow for Haskell’s choice elements to be captured within the circuit descriptions. In this respect \texttt{C\La\La}SH differs from Lava, in that all of Haskell’s choice elements, such as case-expressions and pattern matching, are synthesized to choice elements in the eventual circuit. Consequently, descriptions containing rich control structures can be specified in a more user-friendly way in \texttt{C\La\La}SH than possible within Lava, and hence are less error-prone.

Bluespec [15] is a high-level synthesis language that features guarded atomic transactions and allows for the automated derivation of control structures based on these atomic transactions. Bluespec, like \texttt{C\La\La}SH, supports polymorphic typing and function-valued arguments. Bluespec’s syntax and language
features had their basis in Haskell. However, in order to appeal to the users of the traditional HDLs, Bluespec has adapted imperative features and a syntax that resembles Verilog. As a result, Bluespec is (unnecessarily) verbose when compared to C\alpha SH.

The merits of polymorphic typing and function-valued arguments are now also recognized in the traditional HDLs, exemplified by the new VHDL-2008 standard [1]. VHDL-2008 support for generics has been extended to types and subprograms, allowing a designer to describe components with polymorphic ports and function-valued arguments. Note that the types and subprograms still require an explicit generic map, while the C\alpha SH compiler automatically infers types, and automatically propagates function-valued arguments. There are also no (generally available) VHDL synthesis tools that currently support the VHDL-2008 standard.

VI. Conclusion

This research demonstrates once more that functional languages are well suited for hardware descriptions: function applications provide an elegant notation for component instantiation. While circuit descriptions made in C\alpha SH are very concise when compared to other (traditional) HDLs, their intended functionality remains clear. C\alpha SH goes beyond the existing (functional) HDLs by including advanced choice elements, such as pattern matching and guards, which are well suited to describe the conditional assignments in control-oriented circuits. Besides being able to translate these basic constructs to synthesizable VHDL, the prototype compiler can also translate descriptions that contain both polymorphic types and user-defined higher-order functions.

The C\alpha SH compiler has also been used to translate non-trivial functional descriptions such as a streaming reduction circuit [16] for floating point numbers. Table I displays the design characteristics of both the C\alpha SH 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. Future research includes a more thorough investigation into the performance differences of the two designs.

VII. Future Work

The choice of describing state explicitly as an extra argument and result can be seen as a mixed blessing. Even though descriptions that use state are usually very clear, distributing and collecting substate can become tedious and even error-prone. Automating the required distribution and collection, or finding an abstraction mechanism that suppresses state would make C\alpha SH easier to use. Currently, one of the examined approaches to suppress state in the specification is by using Haskell’s arrow-abstraction.

The transformations in the normalization phase of the prototype compiler are developed in an ad-hoc manner, which makes the existence of many desirable properties unclear. Such properties include whether the complete set of transformation will always bring the description into a form that can be translated to hardware or whether the normalization process always terminates. Although extensive use of the compiler suggests that these properties hold, they have not been formally proven. A systematic approach to defining the set of transformations allows one to proof that the earlier mentioned properties do indeed hold.

References