ABSTRACT

This paper proposes a method of combining the concepts of recursive algorithms and functional programming in hardware design. Though recursion is readily available in functional programming languages, a translation step is required to convert the recursion in such functional programs to a synthesizable hardware specification. This proposal uses the idea of Finite State Machines used in [5], combined with the Haskell to VHDL compiler proposed in [1]. As a conclusion, a short proof of concept of the framework is given, along with an example algorithm (a naive fibonacci sequence implementation) in plain Haskell, using types that can easily be converted to a CλaSH compliant subset of Haskell.

Keywords
CλaSH, Recursion, VHDL, Haskell, Hardware

1. INTRODUCTION

Presently, in programming, extensive use is made of recursion: a set of instructions (in Haskell, a function) within a program which invokes itself. Recursion offers a clear and concise way to represent algorithms describing (the solution to) a multitude of problems, such as searching through a binary tree, or sorting an unordered list of numbers. In this paper, a simple algorithm calculating the nth number in a Fibonacci sequence will be used, an algorithm which calls itself twice in one iteration (when n is greater than 1). Though traditionally, recursive functions are a concept available only in programming languages for software, earlier papers have shown that implementing recursive algorithms directly on a hardware level (using a Hardware Description Language) offers a faster computation, reaching an answer in less clock cycles than software implementations of the same algorithms [3, 5]. Several ways have currently been proposed to implement recursive functions in hardware, but all of them compromise in one way or another (performance, ease of use, space usage) to achieve their goals [4]. Most of these approaches to the problem try to emulate the way recursion is implemented on a higher level, using stacks to store the state of a computation before the recursive step[2, 3, 5], later “reverting” to the state once all subsequent recursive calls have finished computation. Sklyarov has proposed a method along these lines using something called a Recursive Hierarchical Finite State Machine (RHFSM) to represent the algorithm, dividing it into distinct parts called “modules”, each represented by a Hierarchical Graph Scheme (see fig. 1). [5]

The goal of this paper is to demonstrate the viability of combining certain aspects of a functional programming language (Haskell) delivered by CλaSH [1] with another feature, recursion, using a small example written in the form of a specific method (the RHFSM method mentioned above). This method has been chosen for its completeness and relative ease of use. To achieve this goal, the paper will first expand on the RHFSM method, summarizing and explaining the relevant parts of this method, to give the reader a general understanding of how the proof of concept will work, and how it models recursion. A general idea of the process by which CλaSH converts Haskell code to (synthesizable) VHDL is then given. This is done in order to clarify at which level (i.e., software, hardware, netlist) the proof of concept presented in this paper is currently implemented, and at which level the conversion from software based recursion to synthesizable recursion (using the aforementioned RHFSMs) should be implemented.

A proof of concept is then given alongside a regular Haskell implementation of the same algorithm to demonstrate how these RHFSMs and their administrative overhead could be converted to a functional programming language and thus, indirectly, to hardware. Limitations do still exist in the proposed methods, however. These limitations are discussed in the conclusions of this paper, together with proposed future work in the field.

2. SKLYAROV’S METHOD

A crucial part of this paper is to elaborate the method proposed in [5], demonstrating its use and viability as a way to add recursion to the list of features in CλaSH. To do this, this section shall do the following:

- Identify key components in the method,
- Clarify each individual component,
- Provide insight as to the total picture by way of a small example.

In the RHFSM method, a small framework (let’s call it the RHFSM circuit) is built around the entire combinatorial circuit to help it store the information it needs to simulate recursion. As a result, a circuit using this method always contains these elements, even for its non-recursive parts.
Recall that the algorithm is divided in modules, each of which can be represented as a Hierarchical Graph Scheme (HGS, see fig. 1). These modules consist of several states through which they advance by one every clock cycle. At every transition, several computations are done, and possibly a new module is invoked (which could be a recursive call). To model this, we need the following data elements:

- a stack containing module labels of previous levels of recursion as well as the current one,
- a stack containing the current state of previous levels of recursion as well as the current one,
- a stack or register (possibly multiple ones) containing the output of the algorithm,
- a register containing a module label indicating the next module to load when a recursive call is made,
- a register containing the next state of a module, to be executed next clock cycle,
- a register containing a boolean value, the "return flag", indicating whether the circuit has just processed a return from a recursive call.

At the rising edge of the clock, the following operations at the current state label of the loaded module are executed:

1. the register containing the return flag is read. If this is set to true, all computations and output steps are skipped and only the next state is determined,
2. some computations might occur, followed by (possibly) pushing these to an output element (a stack, register or even to the final output of the circuit, if applicable),
3. a recursive call might be made, in which case the label of the new module is first loaded in the module register. If the stack size limit has already been reached, an error can be thrown ending the computation (setting an upper bound on the recursion depth). If this limit has not been reached, The new module label is pushed on the module stack and the begin state ("a0") is pushed on the state stack. The circuit is then ready to begin executing the next module,
4. the top element of the state stack is popped and the current state is pushed to the same stack (effectively updating the current state of this module),
5. the next state is determined and set in the register containing the next state,
6. when a return is in order, the return flag is set to true, the top of the module and state stacks are popped and the next state is set to the top of the state stack. This last operation combined with the return flag prevents a faulty recursive call from being made but properly executes operations determining the next state.

Two separate processes are defined in the VHDL code provided in [5], one for the management of calls and returns, and one containing all the modules (the actual combinatorial circuit). This second process contains case statements determining what state of what module should be executed.

### 3. THE CLASH COMPILER

Considering that this paper aims to provide a possible extension to the C#A#H compiler, it will touch on the nature of C#A#H, in particular the series of steps it takes to reach synthesizable VHDL code. The specifics of how Haskell is rewritten to VHDL, however, will not be covered here. For a more complete description of the process, please refer to [1].

C#A#H itself is actually a subset of Haskell, conforming to a particular syntax. It contains several of the features of Haskell such as polymorphism, pattern matching and higher order functions. The C#A#H compiler, however, can actually take this syntax and generate output in VHDL netlist format. Fig. 2 illustrates the way by which the compiler does this: the Glasgow Haskell Compiler (GHC) takes Haskell code and converts it to an internal, representation called Core, stripped from its syntactic sugar: a "desugared" version of the code. Core is relatively simple compared to Haskell, making it a better starting point for transforming the code to code which is synthesizable. Core is converted to a normal form via a set of transformations. This rewritten, "normalized" code is then transformed to a VHDL netlist ready for synthesis.

Ideally, recursion should be implemented "behind the scenes", as a part of the compilation process within the C#A#H compiler. The framework that surrounds the combinatorial circuit in the RHFSM method lends itself to this process since it neither places a specific limit on recursion depth, nor requires a manual transformation of any sort to the code: it is truly a generic framework. Its only drawback remains that a considerable amount of overhead is incurred, even over non-recursive parts of the algorithm, due to the nature of the framework.

In the current paper, however, we limit ourselves to a...
4. CONVERGING THE FRAMEWORK TO HASKELL

Now that both the method and the environment have been put into their proper context, the next step is a small proof of concept demonstrating the way these templates might look in Haskell code. For this concept, we exchange the somewhat obscure example presented in [5], traversing a binary tree and finally returning a list (stack) with all values of the tree in order from large to small, with a simpler one: retrieving the $n^{th}$ value of a fibonacci sequence.

4.1 The Fibonacci sequence

To clarify the way in which the framework works, a simple algorithm is demonstrated in this paper, namely calculating the $n^{th}$ value of a fibonacci sequence. A fibonacci number can be defined with the following recursive algorithm:

$$n = (n - 1) + (n - 2)$$

A (naive) Haskell implementation of this algorithm can be defined as follows:

```haskell
fib :: Int -> Int
fib n = |
| n == 0 → 0 |
| n == 1 → 1 |
| otherwise → fib (n-1) + fib (n-2)
```

This function in Haskell takes an integer and returns an integer. The vertical lines in the code above are a particular form of case expressions, where, in this case, the first two cases define base cases: grounds for termination of the algorithm. This algorithm is not optimized, but is the most human readable format of the algorithm, making it an appropriate format for this paper.

4.2 Defining the Haskell RHFSM framework

4.2.1 Data definitions

Several features have been changed in this proof of concept compared to the VHDL version found in [5]. Most notably, the type of state is not a newly defined data type, but simply an integer value. To illustrate both approaches, however, the modules have been defined explicitly. A disadvantage of such an explicit data type would be scaleability: every module would need to be defined (registered) to be recognized in the framework. It does, however, provide the programmer with a data type that can be defined as the programmer wishes, taking advantage of Haskell’s features and granting greater control over the flow of the program (i.e., the user can define its own equality and sort operators for his data types for use in the controlling function).

First, we define the state:

```haskell
data State = State {  
nModReg :: Module,  
nStateReg :: Int,  
moduleStack :: [Module],  
retValues :: returnType,  
stateStack :: [Int],  
returnFlag :: Bool  |
| Err
```

This state contains all of the elements summed up in section 2, (note that stacks are defined as lists, a data type specific to Haskell) with an added variable: retValues, of type returnType. This variable can be defined as anything needed internally as well as externally to represent data, such as a tuple containing lists (stacks) and integer values (registers). In the case of the current example, it is defined in the following way:

```haskell
type returnType = ([Int], [Int])
```

This is a tuple containing two lists: let’s call the left one callValues, containing the numbers $n$ which identify the fibonacci number at the current (and previous) level(s) of recursion. The right one, in the current program, is called retValues and contains the current output of the algorithm (an iteration of the fibonacci function pushes the result of its computation to this stack when done).

In our current example, only one module is needed for the execution of the algorithm, though more can be defined as needed. Comments are represented with “–”, and in this case illustrate the syntax of adding more modules to the data type:

```haskell
data Module = Mod00 -- add as needed: | Mod1 | Mod2  
deriving (Eq)
```

data types constructors start with a capital letter. In this case, the line deriving (Eq) is added to make use of the equality operators (used to compare the label currently loaded on top of the module stack to possible options).

4.2.2 Functions

Only three functions are needed to define the RHFSM part of the circuit: a call function, a return function and the stack manager itself. All the modules, finally, are also defined as a function each. The increment function can be defined as follows:

```haskell
hiCall :: State -> State
hiCall s@((State nModR nStateR nStack dStack sStack rFlag))  
| (State nModR 0 (nModR:nStack) dStack (0:sStack) False)
```

This function updates the state by loading the module in the new module register (nModR:nStack) and setting the return flag to False, as it has just made a call, which excludes the possibility of a return being the last operation next iteration. The next “clock cycle” in the model will execute the initial state of the new module.

The return function is defined similarly:
Since popping an element from an empty list produces an error, two cases have been distinguished: one where no modules are left when this function is called, and one where there are. The first case should not happen in normal execution, hence an error state is returned. In normal execution, the updates to the state in this function are a pop of the module stack and state stack \( (\text{tail } \text{sStack} \text{ and } \text{head } \text{sStack}) \), a revert to the current state of the new module \( (\text{head } \text{sStack}) \) and an update of the return flag variable, setting it to True, indicating it has just invoked a return.

Finally, the stackManager is the control function defining which function should be executed after each clock cycle. In the current example, only one module is used, but more modules can be registered by adding the appropriate case above theotherwise and below themStack cases. The stackManager function is defined as follows:

```haskell
stackManager :: State -> ReturnType
stackManager s@((State nModR nStateR mStack dStack
 sStack rFlag)
 | mStack == [] = stackManager (modz0 s)
 | otherwise = Err

| tail \text{sStack} /= [] = stackManager (head \text{tail } \text{sStack})
| \text{head } \text{sStack} (dStack (\text{tail } \text{sStack}) True)
| \text{otherwise} = \text{Err}
```

The function uses the top of the module stack to check which module is active and then executes this module by invoking the corresponding function (note that \( \text{modz0 } s \) in this case is a function rather than a module label due to the lack of a capital letter). A new state is then returned to the stackManager, after which the process repeats itself. If the module stack is empty at any point, the algorithm terminates and \( \text{Err} \) is returned. If at any point an erroneous module label is on top of the module stack, the program terminates with an error value.

Of the three functions defined above, only the stackManager will need a fundamental change apart from changing the types it uses to make it \textit{C}\texttt{ll}\texttt{aSH} compliant: This function should serve as a top entity, executing every clock cycle. It will need inputs and some manner of control structure to help it process the algorithm: it will not be able to calculate any Fibonacci number in one clock cycle, instead accepting input once per \( x \) clock cycles and delivering outputs every \( x \) clock cycles, where \( x \) is the amount of cycles needed to execute the algorithm. Once this control structure and the type conversions are in place, the recursive part of this function should be removed (thus, \textit{"head mStack = z0 = stackManager (modz0 s)"} should be \textit{"head mStack = z0 = modz0 s"} because the stackManager will execute every clock cycle).

### 4.3 Defining the algorithm

In the current example, we have only one module, which can be defined as in fig. 3.

This function models the algorithm in module format as described in fig. 4. In state 0, it is determined whether \( n \) is not 0 or 1. If this is the case, label 2 is executed, which calls the algorithm again with \textit{callValue} \((n - 1)\). If a return function has been invoked in the last iteration, the return flag is \textit{True} and no call is made. The next state is then set to 3, where a similar call is made with \textit{callValue} \( n - 2 \).

Finally, if the current \textit{callValue} is not 0 or 1, the top 2 elements of the \textit{retValues} stack are popped and added to each other. (which are the results of the computations of the \textit{fibonacci} with \((n - 1)\) and \((n - 2)\)) This resulting value is then pushed on the \textit{retValues} stack. In the case where \( n \) is 0 or 1, this value is simply put on the \textit{retValues} stack. Finally a return is called. Recall that the data type \textit{State} had two constructors, \textit{State <elements> and \textit{Err}}. In the case of an invalid state pointer, the function will return an error, terminating the algorithm.

### 5. CONCLUSIONS

The method proposed in [5] is not perfect, but it allows for general recursion with relatively few limits. Two specific issues have been identified as a conclusion to the proof of concept given in the previous section:

- a specific problem with this method discovered as of writing is the fact that even the non-recursive parts of an algorithm have to fall under the administrative portion of the framework, causing a lot of code overhead. This is no problem with automatically generated code as such code is of disputable readability anyway, but as a framework on a higher level, such as Haskell, this may not be considered acceptable. For this reason, it is recommended that this framework be added as a feature generated automatically by the \textit{C}\texttt{ll}\texttt{aSH} compiler, eliminating the need for manual coding in this rather difficult to read and error prone format,

- one of the limitations of implementing recursion in this way is caused by the usage of stacks: the depth of the recursion is limited to the size of the stacks in the framework surrounding the algorithm.

Apart from these two disadvantages, this seems to be an attractive, relatively easy way of adding an important fea-
module State -> State
| nStateR == 0 = State nModR nStateR mStack dStack (0:(tail sStack)) rFlag
| nStateR == 1 = if (rFlag == False) then hiCall (State Modz0 nStateR mStack ((callValue - 1):callValues, retValues) (1:(tail sStack)) rFlag) else State nModR 2 mStack dStack (1:(tail sStack)) False
| nStateR == 2 = if (rFlag == False) then hiCall (State Modz0 nStateR mStack ((callValue - 2):callValues, retValues) (2:(tail sStack)) rFlag) else State nModR 3 mStack dStack (2:(tail sStack)) False
| nStateR == 3 = hiReturn (State nModR nStateR mStack (tail callValues, newDStack) (3:(tail sStack)) rFlag)
| otherwise = Err
where
(callValues, retValues) = dStack
nStateR_0 = case callValue of
| 0 -> 3
| 1 -> 3
| _ -> 1
newDStack = case callValue of
| 0 -> 0:retValues
| 1 -> 1:retValues
| _ -> (head retValues + head(tail retValues)):(tail(tail retValues))

Figure 3. The fibonacci algorithm using the RHFSM framework

Future work may be required to determine in-depth the disadvantages of implementing the RHFSM method in Haskell/Core/CλaSH. Possible fields of study will be briefly discussed below.

5.1 Future work

Future work includes areas of study that fell outside of the current scope of research, as well as possible areas of interest after more advantages and/or disadvantages of the RHFSM approach have been discovered. Below is a short list of possible subjects:

- currently, two methods have been proposed for the storage of module and state labels. Research in this direction investigating the impact on size, performance and feature lists is advised before any further implementation is considered,

- currently, the format of this proof of concept is plain Haskell. For further study, it would be very enlightening to change this framework (and possibly the example algorithm) to CλaSH compliant syntax, enabling the actual synthesis of some example recursive algorithms,

- once initial studies have been made on the actual implementation of this framework in CλaSH, further study into the integration of this framework into the compilation process will almost certainly yield valuable information. One possible implementation could involve a switch in the compiler, after which it would automatically divide functions into modules and states and build a RHFSM framework around it, enabling recursion without the use of an explicit framework visible to the programmer,

- finally, the original method used by [5] uses signals rather than code segments in the actual modules. Research might be valuable ascertaining possible delegations to other circuit parts by setting certain data paths to high, as in the original RHFSM method by Sklyarov et al.

6. REFERENCES

APPENDIX

A. HASKELL SOURCE CODE OF THE RHFSM FRAMEWORK

```haskell
-- the state definition. Contains currently loaded module, a pointer to the next state,
-- a stack of modules in lower scopes and the state they are currently in, a type
-- containing the intermediate/final output of the code, as well as a return flag signaling
-- whether a function is currently due to invoke its hierarchical call or returning from it.
data State = State {
  nModReg :: Module,
  nStateReg :: Int,
  moduleStack :: [Module],
  retValues :: ReturnType,
  stateStack :: [Int],
  returnFlag :: Bool
} deriving (Eq)

-- functions

main = do (n, _) <- getArgs
  print $ "Result: ", stackManager (Modz0 0 [Modz0] ([],[]) [0] False)

-- The hiCall function sets the active module to the newModule, loaded beforehand, then
-- sets the next state within this module to '0', the starting state.
hiCall :: State -> State
hiCall s@(State nModR nStateR mStack dStack sStack rFlag)
  = (State nModR 0 (nModR:mStack) dStack (0:sStack) False)

-- The hiReturn function 'pops' the top elements of both the module and state stack,
-- reverting the current state of the program to just before the hiCall. The returnFlag
-- is set to true in this case, ensuring the program does not enter an infinite loop by
-- immediately calling hiCall again.
hiReturn :: State -> State
hiReturn s@(State nModR nStateR mStack dStack sStack rFlag)
  | tail mStack /= [] = (State nModR (head(tail sStack)) (tail mStack) dStack (tail sStack) True)
  | otherwise = (State nModR 0 (tail mStack) dStack (tail sStack) True)

-- The stackManager function is the top entity in the algorithm, determining which module is
-- to be executed, if any. In this Haskell form, this is the only function that is truly recursive,
-- calling itself to ensure a loop is created until the algorithm terminates. This is to be
-- rewritten if these functions are to be rewritten for Clash, as the top entity should just execute
-- with the given inputs, every clock cycle.
stackManager :: State -> ReturnType
stackManager s@(State nModR nStateR mStack dStack sStack rFlag)
  | head mStack == Modz0 = stackManager (modz0 s)
  | otherwise = ([[-1]], [-1])

-- Algorithm functions

-- a fibonacci implementation
modz0 :: State -> State
modz0 s@(State nModR nStateR mStack dStack sStack rFlag)
  | nStateR == 0 = State nModR nStateR 0 mStack dStack (0:(tail sStack)) rFlag
  | nStateR == 1 = if (rFlag == False) then hiCall (State Modz0 nStateR mStack (callValue - 1):callValues, retValues)
                             (1:(tail sStack)) rFlag
                  else State nModR 2 mStack dStack (1:(tail sStack)) False
  | nStateR == 2 = if (rFlag == False) then hiCall (State nModR nStateR mStack (callValue - 2):callValues, retValues)
                             (2:(tail sStack)) rFlag
                  else State nModR 3 mStack dStack (2:(tail sStack)) False
  | nStateR == 3 = hiReturn (State nModR nStateR mStack (tail callValues, newDStack) (3:(tail sStack)) rFlag)
  | otherwise = Err

where
  (callValues, retValues) = dStack
  callValue = head callValues
  nStateR_0 = case callValue of
    0 -> 3
    1 -> 3
    _ -> 1
  newDStack = case callValue of
    0 -> 0:retValues
    1 -> 1:retValues
    _ -> (head retValues + head(tail retValues)):(tail(tail retValues))
```