Functional HDLs: A Historical Overview

Joseph Cordina
Dept. of Computer Science and A.I.
New Computing Building
University of Malta, Malta
joseph.cordina@um.edu.mt

Gordon J. Pace
Dept. of Computer Science and A.I.
New Computing Building
University of Malta, Malta
gordon.pace@um.edu.mt

ABSTRACT

When designing hardware systems, a variety of models and languages are available whose aim is to manage complexity by allowing specification of such systems at different abstraction levels. Languages such as Verilog and VHDL where designed with simulation in mind rather than synthesis and lack features such as parametrised complex circuit definitions, a must for the design of generic complex systems. A more modern approach is the use of functional languages for hardware description that take advantage of the inherent abstraction in this paradigm, resulting in a more concise and manageable description of the system. This paper gives an overview of different functional language implementations for hardware description, highlighting their historical significance in terms of their capabilities and design approach. We will compare and contrast different ways that certain features, such as circuit sharing, have been implemented in these.

Keywords

Hardware Description Languages, Language Embedding, Functional Languages

1. INTRODUCTION

When an electrical engineer needs to specify a complex system, he or she certainly does not specify the system by listing every single gate and connecting them. The usual techniques of modularity, re-use and abstraction are applied to improve both development time and also the quality of the final system. After specifying the system using some structural description, the engineer may want to perform actions on the circuit described:

- **Simulate the circuit.** At the very least the designer of the system would want to run a simulation of the circuit to observe its behaviour. This is also useful for testing the eventual circuit, where test cases are simulated on the circuit description and on the circuit proper.
- **Verify the circuit's behaviour.** Ideally one should be able to run some model checking techniques to verify that the described circuit obeys from specification. Having this facility increases the confidence of the circuit and also helps in discovering certain obscure bugs early on prior to the expensive hardware realization process.
- **Gather knowledge about the circuit.** These are metrics on the circuit such as number of components or expected signal propagation delay. More complex things which are desirable are behavioural description of the circuit, and an analysis of non-functional aspects.
- **Netlist generation.** Eventually the circuit will need to be placed in hardware, and thus it is crucial that the original circuit description is translated to the gate level together with an unambiguous description of how the gates connect to each other (also known as the *netlist*). The more abstract the specification language and the more automatic the low level description generation, the easier it is for the systems designer. It is also assumed that the original semantic behaviour of the system is maintained when the netlist is generated.
- **Circuit Transformation.** It is also commonly desirable for the designer to be able to make changes to the original circuit specification or to the netlist. This could include changes in functionality or tweaking the circuit to decrease the number of required components. Changes could be the result of bug-fixing or to tailor the system for re-use. Automated circuit transformation would be highly desirable.

The first step to allow the above is to generate some language that is able to describe the final target system. Very popular languages are VHDL and Verilog [1]. These languages offer a variety of features such as allowing the user to specify the circuit using a structural description of the circuit, alternatively using a behavioural description of a circuit, efficiently simulating the target system, synthesizing the high-level description to hardware and being general enough to describe any hardware system. One thing that is immediately obvious when looking at VHDL and Verilog compilers it that a lot of work has been done on optimising

them for simulation. In turn this led to the situation that synthesis becomes a very difficult process and it is not uncommon that one specifies a system that behaves differently after synthesis [4]. One can still use the behavioural description for testing, yet it obviously is not an ideal scenario. In addition, it soon became apparent that certain circuits are very difficult to specify in either VHDL or Verilog. Earlier versions of VHDL did not allow one to define complex circuits which vary according to the definition's parameter (for example a circuit which is made up of a number of components, the number of which is defined as a input parameter inside the circuit's definition). More modern versions allow a very limited subset of these[1]. In brief, while Verilog and VHDL are very successful tools, they do not lend themselves easily to higher-order descriptions when giving the structural description, thus limiting the amount of abstraction one can have. These higher-order descriptions are usually known as *connection patterns* and are functions that take other circuits as input parameters and whose outputs are generic patterns of the input circuits, for example rows, trees or arrays of circuits.

What we discuss in this document is an approach that has proved to be very successful in describing circuits at multiple levels of abstraction, while being able to maintain easy and automatic synthesis and simulation. We concentrate mainly on the structural description of circuits, even though we mention briefly some recent research work in terms of automatic synthesis of behavioural descriptions. Also we limit ourselves to describing synchronous systems, due to the fact that combinational circuits with feedback can only converge to a specific fixed output when taking into account low level propagation delay, something which one normally tries to abstract away from.

2. FUNCTIONAL HDLS

We will now discuss how the functional language paradigm has been been used to allow the specification of hardware systems. This has been done using one of two approaches: the development of new functional languages or the use of existing languages and embedding these with hardware description capabilities.

2.1 Early Work

Following Backus' Turing Award Paper in 1978, it was immediately obvious that descriptions of programs in functional languages tend to be concise and since abstraction is implicit in the model, it is an ideal platform for describing complex systems. In 1984, Sheeran [17] developed a variant of FP, μFP, and used it to describe the tally circuit. What is apparent is that the definition was much more understandable and concise than the same description in VHDL or Verilog (languages that had still to be created). One of the reasons is that the circuit was defined recursively, thus lending itself perfectly to the functional style of programming. In addition, such a concise description makes it easier to debug and to modify.

In μFP, circuits are defined in terms of built in connection patterns, i.e. functions that accept circuit descriptions as input parameters and connect these descriptions together depending on some other input parameters. The successor of μFP was Ruby [18]. In Ruby, while maintaining the concept of combinators, it looks at circuits as relations on streams. In other words, inputs and outputs are seen as a stream of data and circuits as the functions transforming them. This gives the advantage that components such as a delay can be easily specified.

[1]This is achieved through the `generic` keyword but is still not *generic* enough!

2.2 Functional Embedded Languages

Early attempts to create functional hardware description languages (HDLs) concentrated on the creation of new languages that make use of the functional paradigm. Yet this entails the construction of compilers and interpreters for each new language and also does present the problem of deciding upon the syntax and semantics of each new in-built feature.

An alternative approach, is to embed a hardware description language in a generic host language. This allows one to make use of all the features of the host language, thus taking advantage of all the packages available for the host language, including compilers and debuggers. Another large advantage is that the written circuit descriptions are themselves objects within the host language and can thus be manipulated, inspected and transformed. To embed the new language, one usually creates new libraries to allow the description of hardware in the host language.

One the earliest attempts to create an embedded HDL was HDRE [12]. This was implemented in Daisy, a non-strict functional language. In HDRE, wires are treated as a stream of values using lists. One can then easily simulate the circuit by defining circuits as transformation functions on these lists. One can also synthesize the circuit's definitions by having alternate values within the list and defining the circuits themselves as functions on lists of generic types. Depending on the type of values within the lists passed to the functions, one can evaluate (i.e. simulate) the circuit or one can generate a circuit description(i.e. netlist). This approach is called *shallow embedding*. A typical implementation within a Haskell-like language would be as follows:

```
type Signal = [Bit]

inv::Signal -> Signal
inv = map bit_invert

bit_invert::Bit -> Bit
bit_invert True = False
bit_invert False = True
```

Having a lazy programming language, one can also talk about infinite lists which allows any complex circuit to be specified, such as delay circuits whose delay is an input parameter. The biggest disadvantage with shallow embedding is that since circuits are programs within the language, they cannot be inspected and thus something simple like generating the number of gates in the target netlist is impossible.

2.3 Data Types and Deep Embedding

Instead of using lists of values, one can alternatively define circuit descriptions as values in a recursive data type. Then one can write functions that take these values and manipulate them. Thus the description given above would look as follows:

```
data Signal = Inv Signal | Low | High | ...

inv :: Signal -> Signal
inv = Inv
```

We can also create functions that are able to evaluate a given circuit description, thus effectively simulate it. Additionally one can write a function to generate the symbolic interpretation of the circuit, thus resulting in the netlist (symbolic evaluation).

This approach has been taken by the majority of functional HDLs today. One the early implementations to make use of this approach was Hydra [14], a functional HDL implemented in Haskell and now used as a teaching language. In Hydra one still has to annotate the circuit descriptions when needing to generate the netlists. A follow up to this language is Lava [4], a language with a large suite of features including automatic synthesis of circuits in VHDL, specification of generic connection patterns, automatic verification of properties through the use of observers and an adequate model checking tool. Another embedded HDL within Haskell is Hawk [7]. Hawk's main target is for modeling and simulating microprocessor architectures. Architectures can be described at the behavioural level. As Claessen notes in [4], it is very difficult to generate netlists from such a high level of abstraction. In Hawk, high-level components are used as elementary objects within the language and thus its very difficult to simplify these automatically to their gate-level counterparts. Another deeply-embedded HDL, this time within ACL2 is DE2 [9]. This language is mainly targeted towards rigorous hierarchical description and hierarchical verification of finite-state machines. Another variant is the modeling of streams within Daisy [10], a descendant of LISP, which can be used to model communication between self-timed communicating processes.

2.4 Haskell and Embedding

As one can see, Haskell is being used extensively for the use of embedding hardware description languages. One of the main reason for this is purely historical, in that the people working on embedded HDLs have been working closely with the Haskell development team. Yet arguably Haskell has got a very strong type system and is well renown for its elegance and clarity of syntax and semantics.

Other languages have also been used to embed within them HDLs, some with more success than others. There are several groups working on different languages yet one that seems very promising is the construction of an HDL within *reFLect*, a functional language based on ML [11]. Interestingly within such a language one can make use of shallow embedding and then use the reflection capabilities[2] to manipulate the circuit functions, thus arbitrarily changing from shallow to deep embedding as required.

Deciding to embed within a host language does have its disadvantages, primarily that one has to live with limitations of a generic language that was not designed primarily as an HDL. Within Haskell, one cannot define types that can also be given the allowed size of the type. These are known as *sized types*. Certain circuits makes certain assumptions on their input and output types and thus it would be desirable to be able to talk about type sizes within Haskell. Additionally as noted in [4], it would be helpful if Haskell was able to distinguish easily[3] between parameters to circuit descriptions, at the very least between the inputs and outputs signals.

3. OUTPUT SHARING AND FEEDBACK

It is obvious that one inherits a lot of advantages when making use of a functional language for hardware description. One major feature is *referential transparency* whereby any expression will always yield the same result for the same arguments, a property which is assumed in hardware and subsequently components are seen as functional elements. In functional languages, referential transparency is a result of lambda beta reduction, whereby evaluation becomes a simple exercise of argument replacement. Unfortunately this hides away context and does not allow us to refer to internal components form other parts of the program.

Consider Figure 1 that depicts two typical circuits made up of several components. When describing the first in a Haskell-like Language, one would use:

```
circuit_i::Signal -> Signal -> Signal
circuit_i in1 in2 =
  let interm = f in1 in2
  in g interm interm
```

While this definition might look strange, we know it will behave correctly precisely because of referential transparency whereby the evaluation of *g* is applied to two separate evaluations of *interm*. Yet this evaluation strategy leads us to the realization that circuit (ii) cannot possibly be described since f would have to be evaluated twice. Implementing the second circuit as follows, clearly highlights the resulting similarity between the first and second circuits.

```
circuit_ii::Signal->Signal->(Signal,Signal)
circuit_ii in1 in2 =
   let interm = f in1 in2
   in (g1 interm, g2 interm)
```

While this has no effect on its behavioural semantics (thus its simulation), it does have a huge effect when this circuit is realised in hardware. Since the above naive description of this circuit will result in two separate and identical implementations of *f*, this will result in a larger number of components than is strictly required.

[2] *reFLect* is able to reason and manipulate about the programs written within it, using quote and unquote operators

[3] Note that this is possible in Haskell as shown in Wired, yet the techniques used are not straightforward

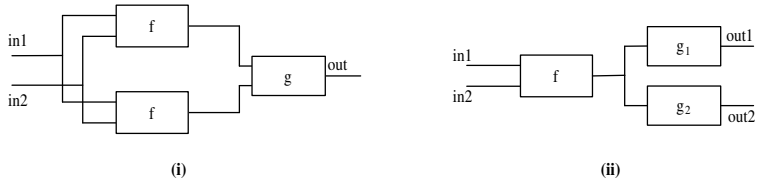

Figure 1: Typical circuits. (i) makes use of two identical circuits while (ii) shares the output of one single circuit.

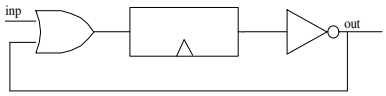

Figure 2: Circuit containing feedback. This circuit will alternate from true to false with every clock cycle, initiated by a true input.

The situation worsens when one has feedback loops, a common occurrence in real circuits. Consider Figure 2 where we have a typical circuit that inverts its output per cycle[4]. It is made up of an OR gate, a latch (a component that delays its input by one clock cycle) and an inverter. At first glance, one would implement such a circuit using the following code:

```
circuit::Signal
circuit inp =
   let out = inv (latch (or (inp,out) ))
   in out
```

Unfortunately when trying to generate the netlist of this circuit, a functional language compiler would try to expand the *out* argument in the second line, whose evaluation depends on the value of *out* again within the OR gate. This cyclic dependency has no terminating condition, and thus would be expanded until there is no more working stack space.

Such a terminating condition can only be found if the HDL in its execution trace can recognise components that it has already visited. Such a clause would also facilitate wire sharing as demonstrated in the beginning of this section. Note that through the use of shallow embedding and functional laziness, one can solve very nicely these problems, yet as noted before, shallow embedding does not allow us to analyse the circuit description within the program.

3.1 Wire Forking

One solution we can envisage is the use of a circuit that explicitly represents the forking of a particular output wire (see Figure 1(ii)). The semantics of this *fork* circuit is that an input wire is connected to this circuit that outputs two or more wires containing a copy of the input wire. This circuit can then be translated in the netlist generation to the diagram shown in the figure. This approach has several drawbacks, most importantly that the use will have to explicitly make use of this *fork* circuit to assemble the circuits. In addition, we cannot envisage how this can solve the problem of feedback loops.

[4]While such a circuit usually requires a synchronisation input, this has been omitted to simplify the circuit.

3.2 Explicit Naming

A better solution is to give a component an explicit name. This name will then be used when generating the description of the circuit. By storing the names of symbolically evaluated components and not evaluating already seen components, one can avoid having infinite recursion loops. A naive implementation would simply keep a list of names of evaluated components, and then traverse this list for every component that has to be evaluated. When implemented within a lazy functional language, making use of certain techniques that delay evaluation to the last possible moment can speed up evaluation considerably [8].

This approach was implemented in Hydra and proposed by O'Donnell in [13]. The code for the inverter circuit would now look as follows, where the inverter has been been given an explicit name.

```
circuit::Signal
circuit inp =
   let out = inv Name1 (latch (or (inp,out) ))
   in out
```

This code would not generate an infinite description since *Name1* will only be evaluated once. Another advantage of this approach is that the explicit name can be carried on all the way to the netlist generation, thus having a reference to the top-level design, aiding in debugging. Yet a major drawback is that it relies on the user to keep track of component names. While this might be a trivial task, it does tend to increase the possibility of error. One alternate approach is to make use of the *fork* approach mentioned above and one just explicitly names the fork components, thus reducing the management overhead required.

3.3 Monadic State

In traditional imperative languages, one makes use of variables to store data that will be needed by subsequent computations. In functional languages, one makes use of state monads to store some data through some computation [20]. In the first version of Lava [16], monads were used to hide away the details of components that have already been evaluated and need to be re-used. Using this method, a circuit identifier can be created automatically. While this approach does away with the user having to specify the names for components, it does require the user to use special operators and ways of programming that tend to quickly make the

code unreadable. In addition, to express feedback loops one needs to make use of some very nasty looking definitions.

3.4 Non-updateable references

Another approach proposed by Claessen and used in the latest version of Lava [6] is the use of non-updateable references resulting in *observable sharing*. Here the problem of sharing of wires is solved through the use of references, very similar to pointers in C. By allowing references to circuits and then evaluating reference equality, sharing can be easily implemented transparently from the user. In addition, both circuits shown in Figure 1 can be easily specified. Thus from the user's point of view, the only significant change is the types of the arguments to circuits.

The introduction of references in a functional language means that referential transparency is not upheld and one could also end up with side-effects. In Lava, the impact is limited by enforcing the references to be read-only. Underneath the hood, observable sharing is achieved without resorting to changes in the compiler by taking advantage of compilers that automatically evaluate an expression only once when this is repeated[5].

4. RECENT DEVELOPMENTS

One major drawback of functional embedded languages is that due to their abstraction mechanism, they are unable to describe *non-functional aspects*. In other words, while they are able to describe the components of the circuits (that contribute to the function of the circuits), an engineer might want to describe the components' eventual placement, how they are connected in terms of distance of wiring, etc. As importantly, before burning to hardware, the wires and the area configuration needs to be analysed since it has a direct impact on the cost of production. The reason why the description of non-functional aspects is difficult in functional HDLs is that it would require descriptions that cannot be evaluated, thus breaking the abstraction levels. Wired [2] is an extension to Lava where the placement of wires in the final circuit can be specified within the language. A series of operators are provided, with which one can connect different components together and specify their relative placement. One can also analyse the eventual space requirements and also power consumption. This approach has also been applied by Taha [19] where in their implementation, two specifications for the generation of the circuit are given. One specification talks about the circuit itself and the other specifies domain-specific optimizations targeted at the circuit generated. This approach avoids the transformation of circuits after they have been generated. Note that when using VHDL or Verilog, one specifies these aspects by using a different specification language than is used to describe the circuit. Yet this just adds another layer which the use has to manually correlate. As of yet, the problem of the complete specification of these aspects in a functional way has not been solved.

In our discussion, we have been mainly concerned with functional HDLs that can describe circuits by using a structural language, normally embedded within a functional language. In 2002, Claessen and Pace [5], showed a technique that allows the specification of a circuit using a behavioural description language. By embedding the behavioural language syntax using data types (very much like HDLs are embedded), one can specify the semantics of the language syntax in terms of other circuits (thus defining its semantics). Furthermore, by making use of the netlist generation mechanism within the HDL, one can easily also automatically synthesize the behavioural description. By making use of previously mentioned techniques, one can also verify properties about the behavioural program. In 2005, Claessen and Pace [15], also showed how one can verify properties about the compilation process itself. At time of writing, the authors of this paper are currently working on implementing an industry standard language, Esterel[3], into Lava allowing complex behavioural programs to be specified while also guaranteeing the semantics during compilation.

[5]This implies that the solution is not entirely portable. A naive compiler might not be able to implement observable sharing correctly.

5. CONCLUSION

Hardware systems are ever increasing in complexity, and are placing large demands on the hardware designer, both in terms of development time and complexity management. To solve these problems, the standard approach is to use different levels of abstraction and then map each layer to the one underneath. This was the approach taken with VHDL and Verilog, even though the mapping between some levels was not automatic and rarely dependable. Another approach is the use of hardware description languages that make use of functional programming paradigm, viewing circuits as maps between the inputs to the outputs. The main advantage of this paradigm is that abstraction is an implicit concept.

In this paper we have seen an overview of several functional HDLs in terms of their historical development. According to our opinion, the most advanced functional HDL to date is Lava with all its supporting libraries, automatic synthesis of circuits and automatic verification capabilities. We have also investigated a small selection of hardware characteristics that tend to be incompatible with the functional way of programming, namely circuits containing feedback and non-functional specification of circuits. We saw how these two characteristics have been recently tackled. We envisage several other approaches will arise in the near future since a lot of ongoing work is being done to solve these problems.

6. REFERENCES

[1] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 1996.

[2] C. Axelsson and Sheeran. Wired: Wire-aware circuit design. In *Charme 2005, LNCS 3725*. Springer, 2005.

[3] G. Berry. The Foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[4] K. Claessen. Embedded languages for describing and verifying hardware, April 2001. Dept. of Computer Science and Engineering, Chalmers University of Technology. Ph.D. thesis.

[5] K. Claessen and G. J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, 2002.

[6] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *Proc. of Asian Computer Science Conference (ASIAN)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[7] B. Cook, J. Launchbury, and J. Matthews. Specifying superscalar microprocessors in HAWK. In *Formal Techniques for Hardware and Hardware-Like Systems*. Marstrand, Sweden, 1998.

[8] HaWiki. Typing the knot, cyclic data structures. Available at http://www.haskell.org/hawiki/TyingTheKnot.

[9] W. A. Hunt, Jr. and E. Reeber. Formalization of the DE2 Language. In *The Proceedings of the 13th Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, No. 3725, pages 20–34. Springer-Verlag, 2005.

[10] S. D. Johnson and E. Jeschke. Modeling with streams in daisy/the schemengine project. In M. Sheeran and T. Melham, editors, *Designing Correct Circuits (DCC'02)*. ETAPS 2002, 2002. Proceedings of the Workshop on Designing Correct Circuits, held on 6–7 April 2002 in Grenoble, France.

[11] T. Melham and J. O'Leary. A functional HDL in ReFLect. In *Designing Correct Circuits*, Mar. 2006.

[12] J. O'Donnell. Hardware description with recursive equations. In *IFIP 8th Internation Symposium on computer Hardware Description Languages and their Applications*, pages 363–382. North-Holland, 1987.

[13] J. O'Donnell. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming Glasgow*, pages 178–194. Springer-Verlag Workshops in Computing, 1993.

[14] J. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in hydra. In *Functional Programming Languages in Education*, Volume 1125 of Lectures Notes in Computer Science. Springer Verlag, 1996.

[15] G. J. Pace and K. Claessen. Verifying hardware compilers. In *Computer Science Annual Workshop 2005 (CSAW'05)*. University of Malta, Sept. 2005.

[16] M. S. Per Bjesse, Koen Claessen and S. Singh. Lava - hardware design in haskell. In *International Conference on Functional Programming*. ACM SigPlan, September 1998.

[17] M. Sheeran. μfp, An Algebraic VLSI Design Language. In *LISP and Functional Programming*, pages 104–112. ACM, 1984.

[18] M. Sheeran. Describing Hardware Algorithms in Ruby. In *Functional Programming, Glasgow 1989*. Springer Workshops in Computing, 1990.

[19] W. Taha. Two-level languages and circuit design and synthesis. In *Designing Correct Circuits*, Mar. 2006.

[20] S. Thompson. *Haskell, The Craft of Functional Programming*. Pearson Assison-Wesley, 2nd edition, 1999.