Embedded Hardware Description Languages: Exploring the Design Space
(Presentation Abstract)

Koen Lindström Claessen[1] and Gordon Pace[2]

[1] Chalmers University, Gothenburg, Sweden
[2] Dept. of Computer Science and AI, University of Malta, Msida, Malta.
{koen@chalmers.se; gordon.pace@um.edu.mt}

Abstract. The embedding of hardware description languages in functional programming languages has been actively explored for a number of decades, and a surprising number of such embedded languages have been developed and used. The first thing a developer of such a language realises, is the rich set of options and possible approaches. In this presentation, we discuss these issues, and examine their advantages and disadvantages in different settings, for languages developed with different objectives in mind.

1 Introduction

Using the *embedded language approach*, one attacks a problem domain by designing a *domain-specific description language* (in which to describe elements of the problem domain), and by implementing this language as a library in a *host language*. The host language is usually a powerful general programming language. Thus, the embedded language inherits many of the features, tools, and users of the host language, allowing the designer of the embedded language to concentrate on the actual problem domain rather than on programming language design.

Functional languages have shown themselves to be excellent host languages for all kinds of domains, and for embedded hardware description languages in particular. Some modern examples are DUAL-EVAL [12], Hawk [5], Hydra [8], Lava [4], "Lava in ReFLect" [6], and Wired [1]. In this paper, we identify some elements of the design space that one faces when designing and implementing an embedded hardware description language, and discusses advantages and disadvantages.

We focus on embedded structural synchronous hardware description languages, embedded in a pure functional language with a rich type system. The intention is for descriptions to be simulatable (i.e. we know the behavioral meaning of our circuits) as well as traversable (i.e. we can generate netlists, call model checkers, or transform circuits). However, we believe that many of the issues are applicable, either directly or indirectly, to other settings as well.

In the following sections, we will illustrate the concepts using Haskell [10] code. However, note that the concepts are in no way limited to embedded hardware description languages in Haskell.

2 Modelling circuits

The first issue that comes up is how to model circuits in the language.

A natural choice to modelling a circuit in a functional language, is to model them directly as functions from inputs to outputs. In this setting, combinational circuits can simply be modelled using boolean operators. However, describing synchronous circuits forces us to use a richer type for signals. Various options present themselves here, which we discuss in more detail in section 3. We illustrate this approach to circuit modelling using two alternative signal implementations — the first representing a signal as a function from time to its value, and the second using lazy lists.

```
-- Functions from time to values     -- Lazy lists
inv s t = not (s t)                  inv s = map not s
and2 (s1,s2) t = s1 t && s2 t        and2 (s1,s2) = zipWith (&&) s1 s2

delay s 0     = False                delay s = low:s
delay s (t+1) = s t
```

Using this approach, we get simulation of circuits for free, and circuit composition is trivial. However, it is more difficult to implement traversability of circuits; some kind of symbolic simulation or reflection needs to be implemented. For example, simply attempting to count the number of gates in a circuit description is impossible in this setting.

An alternative to this approach is to model circuits as a function, but to keep the structure of the signals in the circuit as a datatype. Primitive circuit combinators now simply become constructors. This choice is for example taken in Lava. With this approach, we still get trivial circuit composition, but simulation requires an additional interpretation of the circuit constructors. More importantly, since circuits are now simply data objects upon which we can check equality, we can traverse, access and reason about their structure directly.

```
data Bit = And Bit Bit | Inv Bit | Delay Bit | Low | High

-- A concrete datatype
inv s = Inv s
and2 (s1,s2) = And s1 s2

delay s = Delay s
```

Another choice is to model circuits as special objects (a choice taken in Wired and DUAL-EVAL). In this way, the structure of the circuit can be kept track of

explicitly, and there is greater freedom in what kind of information can be stored within components. However, since function composition cannot be used directly anymore, simulation and composition of circuits becomes more cumbersome.

3 Modelling signals

When modelling circuits as functions, the abstraction level we pick is the level of signals. A signal models the value of a wire that varies over time. What kind of signals do we allow? Should these be reflected in the types? In Hawk, any value is allowed to flow through a signal. In Lava, only booleans (modelling bits) and integers (modelling abstract numbers) are allowed. In Hydra, signals can not only model the values, but also some aspects of non-functional behavior.

Another choice to make is how to model vectors of signals. In Hawk, these are modelled by a "signals of lists of booleans", in Lava, these are modelled by "lists of signals of booleans". This seemingly trivial choice actually has a profound impact on how the rest of the language is designed. One such example is the types of standard components and connection patterns.

```
-- Hawk-style
and2 :: Signal (Bit,Bit) -> Signal Bit

-- Lava-style
and2 :: (Signal Bit, Signal Bit) -> Signal Bit
```

The advantage of the Hawk-style is that everything is a signal, which makes for a clean and flexible design and implementation of an API. The disadvantage of the Hawk-style is that the user is forced to convert back and forth between "structures of signals" and "signals of structures", using *zip*- and *unzip*-like functions:

```
unzipp :: Signal (a,b) -> (Signal a, Signal b)
zipp   :: (Signal a, Signal b) -> Signal (a,b)
```

4 Components

What components are we allowed to use to construct the circuit? Possible components are standard logical gates, busses, transistors, abstract components (perhaps implemented in another language like VHDL). A natural choice seems to be to take the components of our back-end (different kinds of FPGA, CMOS gates, etc.). However, if we want circuits to be model-checked, or if we want to be able to generate parameterized Verilog or VHDL code, we have different restrictions on what kind of components we should allow. What effects does this choice have on the design of the language? What type safety issues are there?

5 Sharing and Loops

No talk on using pure functional languages to describe hardware is complete without a discussion on the effect of the language design on reasoning about the circuit. A pure functional language enjoys the property of referential transparency; we can substitute equals for equals, without changing the meaning of our program. However, we might not end up with the same structural circuit if we do this. Consider the circuit `let c = inv low in and2 (c,c)`, which should (by referential transparency) be equivalent to `and2 (inv low, inv low)`. Does it have one, or two negation gates? In the case of (sequential) loops, the situation is seemingly even worse: `let c = or2 (delay c, s) in c`. One can open up the definition of `c` arbitrarily many times, resulting in that many instances of the disjunction gate and the delay. How sharing of circuit wires (and, relatedly, loops) can be expressed is influenced directly by the choice of how to model circuits [2, 9].

5.1 Named Circuits

One approach to identify sharing (and loops) is to name wires, and using these names to check for sub-circuit equality. This approach has been used in Hydra [7]. With this approach, the major disadvantage is that the users are responsible for responsible naming of circuits — the same name may not be used more than once. Furthermore, unless users are forced to name every single wire, they are also responsible for introducing names in every instance of shared circuits and loops. The advantage of introducing explicit names, is that these can also be used for naming the wires when generating netlist descriptions, or output to model-checkers, since the users would be able to relate the resulting wires with the wires in their design.

```
and2 :: Name -> (Signal Bit, Signal Bit) -> Signal Bit
```

5.2 Monadic Descriptions

A slightly different approach, to relieve the user from the responsibility of naming, one can create a counter which is needed by all circuit generators. The counter is used internally to generate unique names, and is incremented as necessary and returned as an output.

```
and2 :: (Signal Bit, Signal Bit) -> Counter -> (Counter, Signal Bit)
```

The user is now solely responsible for threading the counter though the circuit being generated:

```
halfAdder (a, b) c = (c'', (sum, carry))
  where
    (c', sum)    = xor2 (a, b) c
    (c'', carry) = and2 (a,b) c'
```

Apart from the clutter in the circuit descriptions, this still can give rise to errors through mis-threading the counter. To avoid this, and to make the counter implicit, one can encapsulate it in a state monad:

```
and2 :: (Signal Bit, Signal Bit) -> ST Counter (Signal Bit)

halfAdder (a, b) =
  do
    sum   <- xor2 (a, b)
    carry <- and2 (a, b)
```

This approach has been used in the original version of Lava [3], although dropped in subsequent versions.

6 Deep vs Shallow Embedding

The discussion regarding the modelling of circuits in section 2 started by discussing the merits of describing a circuit simply as a function from inputs to outputs, as opposed to storing its structure as a data object which one can access and traverse. The former is called a shallow embedding, in which the meaning, but not the structure is stored, while the latter is called a deep embedding. If one requires access to the structure of a system, one has to opt for a deep embedding. If one is further interested in the structure of the function calls constructing the circuit (for example to be able to reason about recursive descriptions), one would have to deeply embed further language constructs.

One approach proposed to avoid this issue, is to use a reflective language to enable access to the actual code itself. This approach has been used in ReFLect [6] and Meta-ML [11]. In such languages, one would then be able to use a shallow embedding, and still have access to the circuit components.

7 Conclusions

Although it seems a trivial exercise at first to embed a structural hardware description language in a functional language, there are a number of choices to make, each combination of which leads to a possibly different end result. It is a good idea to make these choices concrete and to categorize them.

References

1. Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, October 2005.

2. Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Proc. of Asian Computer Science Conference (ASIAN)*, Lecture Notes in Computer Science. Springer Verlag, 1999.
3. Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME*. Springer, 2001.
4. Koen Claessen, Mary Sheeran, and Satnam Singh. Using Lava to design and verify recursive and periodic sorters. *International Journal on Software Tools for Technology Transfer*, 4(3):349–358, 2003.
5. Nancy A. Day, Jeffrey R. Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in Haskell. In *CHARME'99 Poster Session*, 1999.
6. Tom Melham and John O'Leary. A functional HDL for ReFLect. In *Designing Correct Circuits*, 2006.
7. John 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.
8. John O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In *Functional Programming Languagues in Education*, volume 1125 of *Lecture Notes In Computer Science*, pages 221–234. Springer Verlag, 1996.
9. John T. O'Donnell. Interconnect and geometric layout in Hydra. In *Designing Correct Circuits*, 2006.
10. Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.
11. Walid Taha. Two-level languages and circuit design and synthesis. In *Designing Correct Circuits*, 2006.
12. Jr. Warren A. Hunt and Erik Reeber. A hierarchical modeling system. In *Designing Correct Circuits*, 2004.