Embedding a Hardware Description Language in a Functional Meta-Programming Language

Gordon J. Pace and Christian Tabone

Department of Computer Science, University of Malta
{gordon.pace|christian.tabone}@um.edu.mt

Abstract. General purpose functional languages such as Haskell, have been widely used as host languages for the embedding of domain specific languages. In particular, various hardware description languages have been successfully embedded in Haskell and other functional languages. More recently, meta-programming languages have also started being used for the embedding of such languages, where the meta-language features allow us to access the structure of data objects in a shallow-style embedding, while retaining the characteristics of a deep-embedding. In this paper, we discuss the application of meta-functional languages for the embedding of a hardware description language, based on *reFLect*, a functional meta-language which provides an alternative approach for embedding a hardware description language by means of built-in reflection features. Through the use of code quotation and pattern matching, we use *reFLect* to build a framework through which we can access the structure of our circuits by means of reflection.

1 Introduction

Designing and developing a new language for a specific domain, presents various challenges. Not only does one need to identify the basic underlying domain-specific constructs, but if the language will be used for writing substantial programs, one has to enhance the language with other programming features, such as module definition and structures to handle loops, conditionals and composition. Furthermore, one has to define a syntax, and write a suite of tools for the language — parsers, compilers, interpreters, etc — before it can be used. One alternative technique that has been explored in the literature is that of embedding the domain-specific language inside a general purpose language, borrowing its syntax, tools and most of the programming operators. The embedded language is usually developed simply as a library in the host language, thus effectively inheriting all its features and infrastructure.

Modern functional languages have been shown to excel as host languages for the embedding of specific domain languages. Features such as strong typing, lazy evaluation, pattern matching and higher-order functions, all make them ideal for the development of small languages within them [Hud96]. One domain for which functional languages have been extensively used to embed languages in is that of hardware description and design [BCSS98,LLC99,BWAH97,O'D06,ACS05]. An

overview of the use of functional languages in hardware design can be found in [She05].

The need for hardware description languages (HDL) emerged as the size and complexity of circuits increased beyond the point where the manual design of circuit systems became unfeasible, creating a need for an infrastructure to describe circuits textually and enable reuse and instantiation. This led to the development of languages such as VHDL [LMS86]. Since various large circuits have regular structures, various extensions and tools for these HDLs appeared, providing features such as iterative descriptions, and static (compile-time) recursion in the description of circuits. These extensions provided a simple meta-language, sitting above the basic HDL, enabling the algorithmic generation of regular circuits through a two-stage language. One main advantage in embedding a HDL in a general purpose programming language, is that these meta-language features essentially comes for free — it is simply the host language. Furthermore, having a full meta-language enables the analysis and transformation of circuits, enables general manipulation of circuits as with any other data objects. Therefore we can not only generate circuits, but also analyse (such as static information gathering, simulation, testing, verification), transform (such as retiming) and interpret (such as netlist generation).

Functional programming languages have proved to be excellent vehicles for embedding languages in a two-stage language approach, enabling allowing access to the HDL description, but do not offer access to the host language code creating the domain-specific objects. This may be useful since certain structuring information inherent in the control structure of the code generating the domain-specific program may be useful in its analysis. Recently, the use of meta-programming techniques for the embedding of HDLs has started to be explored [MO06,Tah06,O'D04]. A meta-programming language enables the development of programs that are able to compose or manipulate other programs or even themselves at runtime, through reflection.

In this paper, we explore the use of *reFLect*, a meta-programming language, to embed a hardware description language in such a manner that we can not only access and manipulate the circuit descriptions, but also the circuit generators themselves. We plan to use these features to access and control the structure of the circuit generated. In particular, in the future, we plan to use this to optimise circuits produced by hardware compilers, maintaining a compositional view of the compiler, but at the same time having access to information as to which parts of the circuits resulted from which features of the compiled language.

2 Functional Meta-Programming in *reFLect*

reFLect [MO06] was developed by Intel, based on the functional language *FL*, but extended with reflection features. *reFLect* is the main programming language used with the Forte tool [SJO$^+$05]; a hardware verification system used by Intel. Forte together with *reFLect* was purposely developed for the development of applications in hardware design and verification, and is mostly used for access

to model checking, decision making algorithms and theorem provers for hardware analysis.

The *reFLect* language is a strongly-typed functional language with added meta-programming features, such as quotaion and antiquotation constructs used to compose or decompose expressions written in the *reFLect* language itself. This provides a form of reflection within a typed functional paradigm setting. The *reFLect* meta-programming constructs provides the developer with an access to the structure of programs written in the whole of the *reFLect* language itself as data objects. Quoted program expressions are considered to be of a special type **term**, representing the abstract syntax tree of the program expression. Traditional pattern matching can be applied on the type **term**, allowing unevaluated expressions to be inspected and interpreted according to the developer's requirements. By combining the pattern matching mechanism with the quotation features, the developer is also able to modify or transform the quoted expression at runtime before evaluation. An in-depth overview of *reFLect* can be found in [GMO06].

2.1 Reflection Operators in *reFLect*

Expressions in *reFLect* are quoted by enclosing them between `{|` and `|}`. The whole expression is typed as a **term**, denoting the abstract syntax tree for the enclosed expression. For instance, consider the expression `1 + 2`. Normal functional features would evaluate this expression resulting to be semantically equal to `3`, and there is no way one can distinguish between the two. However, the application of quotation marks around the expression, `{| 1 + 2 |}`, halts the evaluation by capturing its syntax tree. Note that, the expression `{| 1 + 2 |}` is therefore semantically, and not just syntactically, different from `{| 3 |}`.

The antiquotation construct is expressed by the prefix operator `\`. The antiquotation mechanism essentially raises its operand one level outside the quotation marks. Antiquoted terms always appear within quotations, and have two main applications. Firstly, it is usually to embed a quoted term within another term. To avoid nested quotations, one uses the antiquotation operator to splice one abstract syntax tree into another, thus allowing the construction of terms. For example, given a term, the function below constructs a new term, representing the addition of the the original term with `1`.

```
let incTerm a = {| 1 + \a |};
```

A typical functional call would be as follows, where the input should also be of type term.

```
incTerm {| 2 + 3 |};
```

In *reFLect*, this term would reduce to the expression to `{| 1 + (2 + 3) |}`.

Another application of antiquotation is term decomposition, and used to enable pattern matching on the abstract syntax tree. For example, the function below decomposes the given term into the two operands applied to the addition

operator, binding the left term to the variable `x` and the right term to the variable `y`.

```
let decompose {| `x + `y |} = (x, y);
```

Consider, for example, pattern matching with `{| 5 * 4 + 2 * 3 |}` — `x` would be bound to `{| 5 * 4 |}` and `y` to `{| 2 * 3 |}`. Note how the antiquote is needed to extract the sub-expression as a term. If the function had to be defined without antiquotes using the pattern `{| x + y |}`, the variables `x` and `y` would be non-binding, thus this would match the expression `{| x + y |}` literally.

The *reFLect* language offers a number of built-in evaluation functions, to allow total control over the evaluation of the terms being constructed. The most elementary is the **eval** function, which is used to evaluate a given term, and returns the result as a quotation. The **value** function is similar, since it also evaluates the given term, but the result is returned as the specified type. A **lift** function is also available, and it can be applied to any *reFLect* expression. This works by first evaluating the given expression and then by applying quotation marks around the resulting expression, conclusively lifting the evaluated expression to a higher level of quotations.

2.2 Embedding Languages in *reFLect*

The *reFLect* language, together with the meta-functional features that it offers, provides interesting grounds for the implementation of hardware description languages. Typically, when embedding a language, a deep-embedding is required, since one would want not only to generate programs, but given them different interpretations as may be required, and have access to the underlying syntax of the domain-specific language.

Since, in a meta-programming language, one may quote language constructs, and antiquote terms, one has access to the actual programs as data objects. In *reFLect*, the possibility to pattern match over programs also gives the possibility to look at the structure of an expression. Consequently, in a language like *reFLect* one can build a deep embedding mechanism, simply by using quotations and antiquotations to represent the embedded language using the term datatype, over a simple shallow embedding of the embedded language. Term manipulation is easily achieved through the use of quotations and antiquotations. The ability to directly control the terms of quoted expression, can be applied to expressions representing elements within a circuit model.

Furthermore, using this style of embedding and nested quotations, one can actually reason about marked (quoted) blocks of code hence giving access to the structure of generator of the domain-specific program, effectively enabling reasoning about the embedded language itself at a higher level of abstraction.

3 Embedding a HDL in *reFLect*

3.1 Shallow Descriptions

The simplest way to develop an embedded hardware description language is to define a number of functions that represent the circuits' behaviour. If one uses the boolean values *true* and *false* to represent the circuit constant streams *high* and *low*, the description of the primitive and-gate will simply be an application of the built-in conjunction, thus modelling the logical behaviour of the hardware. The evaluation of such functions, when applied to a set of inputs, would result in the simulation of the circuit. Such a shallow embedding can be implemented in a straightforward manner in *reFLect*. For the sake of simplicity, we consider two basic gates, and-gates, and inverters.

```
let and2 (x, y) = x AND y;
let inv x = NOT x;
```

In a shallow embedding approach the circuits are represented as programs within the host language, thus circuits can be described using the more simple functions that have already been defined. Note that the use of the in-built and-gate is no different from the use of the user-defined components:

```
let or2 (x, y) = inv (and2 (inv x, inv y));
let xor2 (x, y) = or2 (and2 (x, inv y), and2 (inv x, y));
let mux (s, (x, y)) = or2 (and2 (s, y), and2 (inv s, x));
```

The shallow embedding approach offers a straightforward technique for the implementation of a hardware description language. There is no need for the programmer to learn new syntax or programming paradigm since these are inherited directly from the host language, and the default interpretation of the embedded programs, in this case that of simulation, is achieved directly through the interpreter of the host language itself. Nevertheless, as already discussed, through such a shallow embedding one loses all information about the structure of the circuit, and unless the basic gates are overloaded with other interpretations, one loses the option to apply non-standard interpretations of a circuit.

3.2 Using Reflection for a Deep Embedding

Usually, in a language without reflection, to achieve a deep embedding of an embedded language, one creates a datatype to which descriptions are reduced. Using the reflection features, one can take a shallow embedding, as described in the previous section, and quote the circuit descriptions, thus maintaining the structure of the circuit using the structure use of the shallow embedding in *reFLect*. Thanks to pattern-matching on terms in *reFLect*, one can inspect and traverse such circuit descriptions within the language.

Signals can thus consist of either (shallow) values, corresponding to booleans, or (deep) structures, corresponding to terms. In the following datatype definition, *Value* corresponds to the raw boolean value, while *Structure* represents the whole

structure of the circuit operations given as a term. Note that the latter can be evaluated to result in the actual simulation style interpretation of a boolean value.

```
lettype signal = Value bool | Structure term;
```

The primitive gates now have two possible behaviours — the shallow simulation semantics, and the deep quoted version of the shallow interpretation. Using pattern matching one can distinguish between boolean values and structures:

```
forward_declare {inv :: signal -> signal};
let inv (Value a) = Value ( NOT a )
/\  inv (Structure a) = Structure {| inv `a|};

forward_declare {and2 :: (signal, signal) -> signal};
let and2 (Value a, Value b) = Value (a AND b)
/\  and2 (Structure a, Structure b) = Structure {| and2 (`a, `b) |};
```

Other primitive gates are defined using functions similar to the above, which can be presented to the end user to be used for other circuit descriptions. The constant expressions *high* and *low* are defined for *Value T* and *Value F* respectively. Additional constants are also defined to hide quotation constructs from the end user.

```
let high = Value T;
let low  = Value F;

let shigh = Structure {| high |};
let slow  = Structure {| low |};
```

The structure embedded in the above manner enables circuits to be described in a functional style. Furthermore, the use of user-defined blocks is identical to the use of the basic primitive gates. Consider the following definition of a multiplexer:

```
let mux (s,(a,b)) = or2 (and2 (s, b), and2 (inv s, a));
```

Such a description can be interpreted in different ways. Passing a boolean value, one obtains the result of simulating the circuit, while passing a structure, one obtains the internal structure of the multiplexer circuit.

```
: mux (high, (low, high));
: high;

: mux (shigh, (slow, shigh));
: Structure {| or2 (and2 (high, high), and2 (inv high, low)) |}
```

An alternative approach, which we are also considering is the overloading of `high` and `low`, then adding simulation, and structure creation functions, which would enforce one, or the other interpretation of `high` and `low`.

3.3 Representing Signals

A crucial design decision that is needed when developing a HDL is the way circuits inputs and outputs are considered to be structured [CP07]. In the previous section, we have presented the signals used by the circuit descriptions as structure of signals, similar to how signals are represented in Lava [BCSS98]. In other words, an and-gate takes a pair of two wires as input, each carrying a boolean signal. Another form of representation, the one adopted in Hawk [LLC99], is to consider only circuits with one input and output wire, but carrying a signal of structures upon it. Contrast the Lava and Hawk types of a two-input and-gate below:

```
// Signals in Lava
and2 :: (Signal bool, Signal bool) -> Signal bool

// Signals in Hawk
and2 :: Signal (bool, bool) -> Signal bool
```

Currently, we are using the signal of structures representation, primarily since it simplifies language design (although not necessarily language usage). An advantage of this representation is that all circuits defined in a language using this representation will always have the same type — taking a single input and producing a single output. This makes the design much cleaner, and the interpretations work seamlessly even when describing complex circuits built from smaller circuit descriptions. On the other hand, the user has to to handle the wrapping and unwrapping of the signal type whenever the inner vector values are required. For this we provide functions to convert the signal structure back and forth to the structure values.

```
// From signal values to signal structure
zipp :: (Signal bool, Signal bool) -> Signal (bool, bool)

// From signal structure to signal values
unzipp :: Signal (bool, bool) -> (Signal bool, Signal bool)
```

3.4 Marking Blocks in Circuits

In *reFLect*, as in most other HDLs, one views and defines circuits as functions. As a circuit description is unfolded, all the internal structure is lost, and all that remains is a netlist of interconnected gates. To enable marking such subcomponents inside a circuit, we introduce the concept of a block, which a hardware designer may use at will. Such blocks are used in netlist generation, and are planned to be used also in other non-functional features of circuits we plan to implement, including modular verification, placement and local circuit optimisation. For example, one may mark a *halfAdder* as a block, and then use two instances to define a *fullAdder*, which may itself be marked as a block (thus containing two sub-blocks inside).

When the abstract circuit description corresponds to a good layout, or describes together related components, preserving such information can be useful. Adding block information to the whole structure of the circuit, adds a higher level of abstraction over the circuit description, enabling not only the possibility to reason about the structure in terms of primitive gates, but also in terms of blocks. For instance, information gathering functions could be defined to count full-adders or half-adders, or any other block. The placement of circuits will also benefit, since this can be organised into blocks, hence decreasing the level of complexity.

Block information is handled by the meta-programming features of *reFLect* by using nested quotations to represent levels of a blocked circuit structure. A function *block* is defined to create a lambda expression equivalent to the given function, which is then lifted with a higher level of quotation marks, marking the lambda expression as a block.

```
let multiplexer = block mux;
```

The circuit *multiplexer* can now be used in the same manner as *mux*, but with a extra level of nesting being automatically added to enable us to identify blocks as we traverse a circuit. One can also name a block through an extra string parameter which is used in netlist generation.

3.5 Circuit Interpretations

Although the underlying interpretation of a circuit, as we develop it in our HDL is that of simulation one can provide various other interpretations of a circuit description.

Simulation: The simulation interpretation works similar to how a shallow-style embedding operates. Since reflection is used to maintain control over the structure, the simulation is achieved by the *reFLect* interpreter, thus an interpretation function is not required. Therefore, a quick simulation can be achieved by evaluating a circuit description using raw values as inputs. However, if the structure is retained by the use of structured inputs an evaluation function is required to simulate the structure. This simulation function does not interpret the structure but rather it handles the signal structure before applying the built-in evaluation function of *reFLect*.

Information gathering: Information (such as a gate or block count) can easily be gathered about a circuit using pattern matching functions, which follow through quoted circuits to identify sub-circuits and evaluate the information accordingly.

Netlist generation: To enable outputting a circuit description in a format which can be used by external tools, it is of utmost importance to be able to generate a netlist of a circuit description in *reFLect*. The default description is a flat netlist, which does not take into account blocks. One way blocks may be used is to modularise descriptions, giving a separate description of sub-circuits marked as blocks, and referring to that description when used.

In the future we also plan to use blocks to mark primitive components for reducing the description into a netlist for a particular gate technology.

Postscript generation: Descriptions of circuits can also be translated into basic Postscript figures, marking labelled blocks for reference. We are currently exploring the use of placement operators in the language, which would also affect the presentation in Postscript. Furthermore, sharing of circuits creates additional complications which still have to be resolved.

3.6 A Illustrative Example: Serial Carry Adders

In this section, we will present, the development of an n-bit serial carry adder. We start off by defining a half-adder. Note that, since the signal carrying pairs is passed on to the underlying gates, the the unzipping of the inputs into two separate signals is not required in this case.

```
let halfAdder a_b =
      let sum   = xor2 a_b in
      let carry = and2 a_b in
      zipp (sum, carry);
```

Next we declare the function *halfAdder* as a block, using the function *namedBlock*:

```
let halfAdder = namedBlock "halfAdder" halfAdder;
```

Based on the description of the half-adder, we can now define a full-adder circuit structurally, also declaring it as a block. Note that in this case, the circuit designer has to handle the wrapping and unwrapping of the signal structure explicitly.

```
let fullAdder inps =
      val (carryIn, a_b) = unzipp2 inps in
      val (sum1, carry1) = unzipp2 (halfAdder a_b) in
      val (sum, carry2)  = unzipp2 (halfAdder (zipp (carryIn, sum1))) in
      let carryOut       = xor2 (zipp (carry1, carry2)) inv
      zipp (sum, carryOut);

let fullAdder = block "fullAdder" fullAdder;
```

Finally, we can define an n-bit adder is defined as a recursive function:

```
letrec nBitAdderAux (carryIn, ([], []))     = ([], carryIn)
/\     nBitAdderAux (carryIn, (a:as, b:bs)) =
         let inps             = zipp (carryIn, zipp (a, b)) in
         val (sum, carry)     = unzipp2 (fullAdder inps) in
         val (sums, carryOut) = nBitAdderAux (carry, (as,bs)) in
         (sum:sums, carryOut);
```

```
let    nBitAdder cin_as_bs =
       let (cin, as_bs) = unzipp2 cin_as_bs in
       let (as, bs)     = unzipp2 as_bs in
       zipp2 (nBitAdderAux (cin, (unzipps as, unzipps bs)))
```

The use of an auxiliary function avoids the wrapping and unwrapping of structures of signals. However, since blocks can only be used on a structure of signals, should we want to encapsulate each call to the n-bit adder, we would need to wrap before closing the function in a block:

```
letrec nBitAdderAux (carryIn, ([], []))     = ([], carryIn)
/\     nBitAdderAux (carryIn, (a:as, b:bs)) =
         let inp              = zipp (carryIn, zipp (a, b)) in
         val (sum, carry)     = unzipp2 (fullAdder inp) in
         val inps             = zipp2 (carry, zipp2 (zipps as, zipps bs)) in
         val (sums, carryOut) = nBitAdderBlock inps in
         (sum:sums, carryOut);

let    nBitAdder cin_as_bs =
       let (cin, as_bs) = unzipp2 cin_as_bs in
       let (as, bs)     = unzipp2 as_bs in
       zipp2 (nBitAdderAux (cin, (unzipps as, unzipps bs)))

let    nBitAdderBlock = block nBitAdder
```

4 Related work

HDL implementations like Lava [BCSS98], Hydra [O'D06] and Hawk [LLC99], differ from the work presented in this paper, since these have been developed using the deep embedding technique within the functional language Haskell, while our approach is that of using reflection within *reFLect* as a replacement for deep embedding. Deep embedding allows the developer to provide multiple semantically interpretations of the defined circuits, which is clearly seen in Lava, Hydra and Hawk. These HDLs provide several alternative interpretations of a circuit. For example, an inverter gate can have alternative interpretations defined for simulation, netlist creation and timing analysis. Unlike this approach, our implementation uses quotations to capture the circuit structure as an unevaluated expression. Note that, given a different setting, this expression would have been used to simulate the circuit. However, by delaying the evaluation and by having access to the abstract syntax tree of the expression, we are able to traverse this structure and output additional semantically interpretations. The advantage is that the different semantic interpretations operate on the same instance of the quoted expression. However, this needs to be done in two separate stages, first to compose the structure, and then to interpret the structure.

The meta-programming features found in *reFLect*, provides not only the possibility to manipulate terms representing primitive gates, but also to manipulate

terms representing whole circuit definitions. Embedding a HDL using such features can result in an advantage over other HDL embeddings, since the access and manipulation of whole circuit definitions (the circuit generators), should aid in the reasoning of non-functional aspects of circuits. The hardware description languages mentioned earlier have shown that the deep embedding approach offer more advantages over the shallow embedding approach, yet, these don't have full control over certain circuit features, especially over the non-functional aspects. For instance, an important non-functional aspect of circuits is the placement of the primitive elements. Pebble [LM98], a small language similar to structural VHDL, defines circuit components in terms of blocks. The end-user can describe how the blocks are positioned, meaning that a block can be defined to be placed above or beside another allowing blocks to be placed either vertically or horizontally to each other. In our implementation we adopted this idea of blocks, by means of the meta-programming features provided by *reFLect*. However, the challenges are different from those of Pebble, since Pebble is not an embedded language within a function language. In Pebble, language constructs where developed to define blocks and the placement of these blocks, while our implementation uses nested quotation constructs to represent a block in a functional setting, by abstracting away the details of whole circuit definitions.

Wired [ACS05] is another embedded HDL, built upon the concept of connection patterns, in a certain way extending Lava to enable reasoning about connection of circuit blocks. The concepts behind Wired are mostly inspired by Ruby [JS94], more precisely on the adoption of combinators for the placement of circuits. We plan to follow certain features of Wired, for instance to use combinators at the abstract level of blocks.

Our work is based on similar work done in embedding a Lava-like HDL in *reFLect* [MO06]. As in their case, we base our access to the structure of the circuit descriptions on reflection features of the host language. One difference in our approaches, is that we use structures to represent signals, as opposed to raw boolean values used in *reFLect*. One of the reasons for this variation is that we try to conceal the use of quotation marks in the circuit descriptions, hence making the reflection features used only in the underlying framework — not forcing the end user to use these constructs. In our approach we emphasis the concept of a marked block in a circuit, which we plan to use for placement and circuit analysis. We still have a number of features unimplemented — such as the lack of component sharing and implicit wrapping and unwrapping of structures of signals — which we plan to develop in the near future.

5 Conclusions and Future Work

In this paper, we have presented a rudimentary HDL embedded in a functional meta-programming language. Our main motivation behind the use of reflection is to enable the creation of tagged blocks by looking at the structure and control-flow of the circuit generator. By adding circuit combinators, similar to the ones used in Ruby [JS94], we plan to use the control given to us into looking at the

circuit generators to aid the generation of placement hints as used, for instance, in Pebble [LM98].

Another area we intend to explore is that of optimisation of circuits produced by hardware compilers. The use of embedded HDLs for describing hardware compilers has been explored [CP02]. Despite the concise, compositional descriptions enabled through the use of embedded languages, the main drawback is that the circuits lack any form of optimisation or information. Furthermore, introducing this into the compiler description breaks the compositional description, resulting with a potential source of errors in the compilation process. If one still has access to the recursive structure of the control flow followed by the compiler to produce the final circuit, one can perform post-compilation optimisation, without having to modify the actual compiler code. We plan to investigate this further through the use of the features provided by *reFLect*.

References

[ACS05] Emil Axelsson, Koen Linström 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.

[BCSS98] Per Bjesse, Koen Linström Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 1998.

[BWAH97] Bishop C. Brock and Jr. Warren A. Hunt. The DUAL-EVAL hardware description language and its use in the formal specification and verification of the FM9001 microprocessor. *Form. Methods Syst. Des.*, 11(1):71–104, 1997.

[CP02] Koen Claessen and Gordon J. Pace. An embedded language framework for hardware compilation. In *Designing Correct Circuits '02, Grenoble, France*, April 2002.

[CP07] Koen Linström Claessen and Gordon J. Pace. Embedded hardware description languages: Exploring the design space. In *Hardware Design and Functional Languages (HFL'07), Braga, Portugal*, March 2007.

[GMO06] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[JS94] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. *Sci. Comput. Program.*, 22(1-2):107–135, 1994.

[LLC99] John Launchbury, Jeffrey R. Lewis, and Byron Cook. On embedding a microarchitectural design language within Haskell. *SIGPLAN Not.*, 34(9):60–69, 1999.

[LM98] Wayne Luk and Steve McKeever. Pebble: A language for parametrised and reconfigurable hardware design. In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm*, pages 9–18, London, UK, 1998. Springer-Verlag.

[LMS86] Roger Lipsett, Erich Marchner, and Moe Shahdad. VHDL — the language. *IEEE Design and Test*, 3(2):28–41, April 1986.

[MO06] Tom Melham and John O'Leary. A functional HDL in *reFLect*. In Mary Sheeran and Tom Melham, editors, *Sixth International Workshop on Designing Correct Circuits: Vienna, 25–26 March 2006: Participants' Proceedings*. ETAPS 2006, March 2006. A Satellite Event of the ETAPS 2006 group of conferences.

[O'D04] John T. O'Donnell. *Embedding a Hardware Description Language in Template Haskell*, chapter Embedding a Hardware Description Language in Template Haskell, pages 143–164. Springer Verlag, 2004.

[O'D06] John O'Donnell. Overview of Hydra: a concurrent language for synchronous digital circuit design. *International Journal of Information*, pages 249–264, 2006.

[She05] Mary Sheeran. Hardware design and functional programming: a perfect match. *Journal of Universal Computer Science*, 11(7):1135–1158, 2005.

[SJO+05] Carl-Johan H. Seger, Robert B. Jones, John O'Leary, Tom Melham, Mark D. Aagaard, Clark Barrett, and Don Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, September 2005.

[Tah06] Walid Taha. Two-level languages and circuit design and synthesis. In *Designing Correct Circuits*, 2006.