View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OAR@UM

Access to Circuit Generators in Embedded HDLs

Gordon J. Pace
gordon.pace@um.edu.mt

Christian Tabone
christian.tabone@um.edu.mt

University of Malta

Abstract

General purpose functional languages have been widely used as host languages for the embedding of domain specific languages, especially hardware description languages. The embedding approach provides various abstraction techniques, enabling the description of generators for whole families of circuits, in particular parameterised regular circuits. The two-stage language setting that is achieved by means of embedding, provides a means to reason about the generated circuits as data objects within the host language. Nonetheless, these circuit objects lack information about their generators, or about the manner in which these where generated, which can be used for placement and analysis. In this paper, we use *reFLect* as a functional language with reflection features, to enable us not only to access the circuits, but also the circuit generators. Through the use of code quotation and pattern matching, we propose a framework through which we can access the structure of the circuit in terms of nested blocks that map the generation flow that was followed by the generator.

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.

Functional programming languages have proved to be excellent vehicles for embedding languages in a two-stage language approach. One domain in which this approach has been extensively applied is hardware design. These embedded hardware description languages enable access to the hardware descriptions, although not to the host language code that creates the domain-specific objects. Having access to the generators themselves may be useful since certain structuring information inherited from the control structure of the code generating the domain-specific program may be useful in the analysis of the resulting program. 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 and manipulate other programs or even themselves at runtime through the use reflection. Meta-programming techniques provides an opening not only to access the circuits being generated, but even the generators that created these circuits. Therefore, the reasoning about the structure of the circuit generators is made possible, enabling to inspect and analyse the composition of circuits in terms of nested blocks, thus achieving a higher level of abstraction. Furthermore, the information that is provided through the nesting of these circuit blocks, should provide placement hints for the circuits, that can be combined with other user defined placement information.

In this paper, we explore the use of *reFLect*, a meta-programming language, to embed an HDL in such a manner that we can not only access and manipulate the circuit descriptions, but also the circuit generators themselves. We present our approach to mark the boundary of circuit blocks, and illustrate its potential by means of a couple of prefix circuit examples. 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] is a strongly-typed functional language with meta-programming capabilities. *reFLect* was developed by Intel as the successor of *FL*, as part of the Forte tool [SJO$^+$05]; a hardware verification system used by Intel. *reFLect* and Forte were purposely developed to aid the development of applications in hardware design and verification, and are mostly used for model checking, decision making algorithms and theorem provers for hardware analysis.

reFLect provides quotation and antiquotation constructs, allowing the composition and decomposition of unevaluated expressions, defined in terms of the *reFLect* language itself. These meta-programming constructs provide a form of reflection within a typed functional paradigm setting, enabling direct access to the structure of programs as data objects. This is made possible by giving access to the internal representation of the abstact syntax tree of the quoted expressions. Traditional pattern matching can even be used on this representation, allowing the structure of unevaluated expressions to be inspected and interpreted according to the developer's requirements. Furthermore, by combining the pattern matching mechanism with the quotation features, the developer is able to modify or transform the quoted expression at runtime before evaluation. A more in-depth overview of *reFLect* can be found in [GMO06].

2.1 A brief introduction to meta-programming in *reFLect*

Expressions in *reFLect* can be quoted by enclosing them between `{|` and `|}`. Such expressions are typed as a *term*, denoting a representation for the abstract syntax tree for the enclosed expression. For instance, consider the simple expres-

sion `T AND F`. Normal functional features would evaluate this expression resulting to be semantically equal to `F`. However, the application of quotation marks around this expression, `{| T AND F |}`, delays the evaluation. Note that, the expression `{| T AND F |}` is therefore semantically, and not just syntactically different from `{| F |}`.

The antiquotation construct ` raises its operand one level outside the quotation marks. An antiquotation always appears within quotations, and has two main applications — composition and decomposition of the type term. To compose terms, an antiquotation acts as a splicing construct to join one abstract syntax tree to another. For example, the function below constructs a new term, representing the logical conjunction of two sub-expressions, `a` and `b`. Note that these sub-expressions are also quoted expressions.

```
let compose (a, b) = {| `a AND `b |};
```

A typical functional application of the above definition is given below, followed by the resulting output.

```
:  compose ( {| T OR F |}, {| NOT F |} );
{| (T OR F) AND (NOT F) |}
```

Antiquotations may be used to decompose a term type, by applying pattern matching on the structure of the quoted expression. For example, the function below decomposes the given term into the two operands applied to the `AND` operator, binding the left sub-expression as a term to the variable `x` and the right sub-expression as a term to the variable `y`.

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

Consider the previously composed term, and how this can be decomposed back to the original sub-expressions by means of pattern matching, where `x` is bound to `{| T OR F |}` and `y` to `{| NOT F |}`.

```
:  decompose {| (T OR F) AND (NOT F) |};
( {| T OR F |}, {| NOT F |} )
```

The antiquote is needed to extract the sub-expression as a term type. If the function had to be defined without antiquotes using the pattern `{| x AND y |}`, the variables `x` and `y` would be non-binding, thus this would match the expression `{| x AND 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 the contents of a given term, returning the result as a quotation. The **value** function is similar, since it also evaluates the given term, but the result is type casted into the specified type. A **lift** function is 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 HDLs. Typically, when embedding a language, a deep-embedding is required, since one would want not only to generate programs, but allowing the possibility to give them different interpretations as may be required, and have access to the underlying syntax of the domain-specific language.

In a meta-programming language, one may quote all of the language constructs, resulting in having 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. Term manipulation is easily achieved through the use of quotations and antiquotations. The ability to directly control the abstract syntax tree of quoted expression, can be applied to expressions representing elements within a circuit model.

Furthermore, using this style of embedding, one can mark blocks of code, effectively giving structure to the generator of the domain-specific program, which can be accessed and therefore reasoned about in terms of these blocks. This enables the reasoning about the embedded language itself at a higher level of abstraction.

3 Embedding a HDL in *reFLect*

Usually, in a language without reflection, to achieve a deep embedding of a language, one has to handle descriptions as complex data objects. Through the use of reflection, a shallow embedding approach suffices, since quotation can be used to maintain the structure. Terms thus become the primary type of embedded programs which, in our case, contain circuit descriptions with the potential to evaluate to any structure of signals. We use phantom types are used to keep track of the type of the quoted expression, thus enabling a strongly typed embedded language, able to handle type checking over quoted expressions, and distinguish between the different types of signal structures that a term can be evaluated to.

```
lettype *a signal = Signal term;
```

The primitive gates ensure that the signals are of the correct structure and type, whilst decomposing the structure within the type term into the appropriate input signals. These signals or sub-expressions are hence used to compose the required expression.

```
let inv (Signal {| ^a |}) = Signal {| NOT ^a |};
let and2 (Signal {| (^a, ^b) |}) = Signal {| ^a AND ^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 `Signal {|T|}` and `Signal {|F|}` respectively. Additional constants and display functions are also defined to hide

the meta-programming constructs from the end user.

3.1 Representing Signals

A crucial design decision that is needed when developing a HDL is the way circuits inputs and outputs are to be structured [CP07]. In Lava [BCSS98], for example, signals used by the circuit descriptions are grouped together as a structure of signals as opposed to a signal of structures as represented in Hawk [LLC99].

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 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|})
```

Following this approach, circuit descriptions are require additional code to handle the wrapping and unwrapping of signal. For instance a two-bit multiplexer circuit would be defined as follows:

```
let mux s_ab =
      val (s, ab) = unzipp s_ab in
      val (a, b)  = unzipp ab in
      or2 (zipp (and2 (zipp (inv s, a)), and2 (zipp (s, b))));
```

Applying input values to our multiplexer definition, the function would return the structure of an unevaluated program, which represents the circuit that has been defined. Note that named variable inputs can also be applied.

```
:  mux (zipp (low, zipp (low,high)));
Signal {| (low AND low) OR ((NOT low) AND high) |}
```

3.2 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 (implicit in the way the generators are invoked) is lost, and all that remains is a netlist of interconnected gates. To enable marking such sub-components inside a circuit, we enable marking blocks, which may be used at any stage in the description.

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 half-adder definition as a block, and then use two instances of this block to define a full-adder, 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.

The function `makeBlock` composes the structure of a lambda expression of the given circuit definition that is to be marked as a block. The function first generates quoted variables to match the inputs of the circuit. Next, we apply these variables to the function, which would return the program as a data object using the generated variables. Hence, we compose a lambda expression by means of the generated variables and the program structure. Finally, the output of the `makeBlock` function, is yet another function, which has the same type of the input circuit definition, where the input is placed to the rest of the structure as the input of the composed lambda expression, effectively composing a functional application within quotations.

```
let makeBlock circuit =
       let vars = genInputVariables circuit in
       let fnct = circuit vars in
       \(Signal inp) . {| (\`(getTerm vars) .  `(getTerm fnct)) `inp |}
```

The `makeBlock` function composes a term in which the functional application is delayed by means of a quoted lambda expression. Since this function returns another function of the same type, this can therefore be used seamlessly within other circuit definitions. For instance, consider a second multiplexer function, that is defined as a block of the previously defined multiplexer function.

```
let multiplexer = makeBlock mux;
```

By apply a set of inputs to the function multiplexer, the resulting structure is a lambda expression representing the multiplexer circuit as a block or a component. The inputs are separated from the rest of the structure, by means of the delayed functional application. Note that by marking a circuit definition as a block, the inputs are not folded within the internal structure of the circuit, leaving a clear boundary which can be extracted by means of pattern matching.

```
:  multiplexer (zipp (low, zipp (low, inv high)));
Signal {| (\ (v1,(v2,v3)) .
          ((v1 AND v3) OR ((NOT v1) AND v2)) (low,(low, NOT high)) |}
```

Adopting this approach does not affect the simulation of the program in any

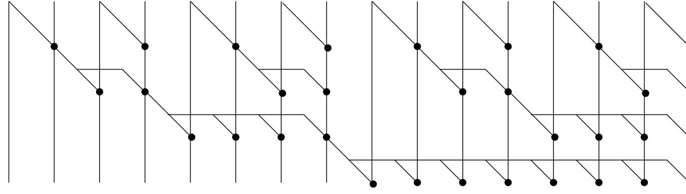

Figure 1: A Sklansky parallel prefix circuit

way, since the only modifications that are made to the object program, are of a syntactic and not semantic nature. The benefit for this program transformation is that patterns matching lambda expressions within terms can be interpreted as blocks or components.

3.3 An Illustrative Example: Parallel Prefix Circuits

In this section we present a some examples for the definitions of parallel prefix circuits, namely the Sklansky network and the Slices network [She07]. Our motivation is to show how to adopt the marking of nested blocks, thus enabling to output a final description with added block details that map the generation flow that was followed by the circuit generator.

The Sklansky network performs the parallel prefiex operation by dividing the input bus into two recursively. The binary operator is applied to the last bit of the first half over all of the second half of the bus. In implementing the Sklansky prefix network circuit (see figure 1), we focus on how the marking of blocks is handled within such descriptions. The recursive definition for the Sklansky network is given as the auxiliary function `skl'` marking blocks as the recursive description unfolds. Any additional parameters, such as the bus width and the operator, are eliminated leaving only the circuit signal as the input. The `makeBlock` function is then used to mark all function calls to the circuit description.

```
letrec skl n op inp =
         let skl' 1 op inps = inps
         /\  skl' n op inps =
               val (lst,rst)  = unzipp (splitSignalBus n inps) in
               let ls2        = skl (busLength lst) op lst in
               let rs2        = skl (busLength rst) op rst in
               let carry      = lastSignal ls2 in
               let apply r    = op (zipp (carry, r)) in
               zipp (ls2  @  map apply rs2) in
         makeBlock (skl' n op) inp;
```

By means of input variables we can create term structures to any specified bus width, hence the create structure is translated to a more readable format. Listing 1 gives the output for the Sklansky circuit for an 8-bit input bus. The statement `BLOCK(vars)... ENDBLOCK` signifies a circuit block with the variables `vars` as inputs, and the final result as outputs. While if this is preceded

Listing 1: The generated output for the Sklansky description of 8 inputs

```
BLOCK (bus_0)
  let (bus_1,bus_2) = ([bus_0 (1)... bus_0 (4)], [bus_0 (5)... bus_0 (8)]) in
  let bus_3 =
        INPUT (bus_1) IN
        BLOCK (bus_4)
          let (bus_5,bus_6) = ( [bus_4 (1), bus_4 (2)],
                                [bus_4 (3), bus_4 (4)] ) in
          let bus_7 =
                INPUT (bus_5) IN
                BLOCK (bus_8)
                  let (bus_9,bus_10) = ([bus_8 (1)], [bus_8 (2)]) in
                  [bus_9 (1), AND2 (bus_9 (1), bus_10 (1))]
                ENDBLOCK in
          let bus_13 =
                INPUT (bus_6) IN
                BLOCK (bus_14)  ...  ENDBLOCK in
          [ bus_7 (1), bus_7 (2),
            AND2 (bus_7 (2), bus_13 (1)), AND2 (bus_7 (2), bus_13 (2)) ]
        ENDBLOCK in
  let bus_19 =
        INPUT (bus_2) IN
        BLOCK (bus_20) ... ENDBLOCK in
  [ bus_3 (1) ... bus_3 (4),
    AND2 (bus_3 (4), bus_19 (1)) ...  AND2 (bus_3 (4), bus_19 (4)) ]
ENDBLOCK
```

by `INPUT(signal)` signifies that `signal` is supplied as the input to the block. Notice how the block markings follow the pattern in which the circuit has been generated.

To illustrate further the use of our embedded language, consider the following description of the Slices parallel prefix circuit description [She07]. Following the recursive decomposition of the circuit (see figure 2), the functions `applyOnEvens` and `applyOnOdds` applies the operator `op`, at even and odd intervals of the bus respectively:

```
let applyOnEvens n op inp =
        let t = unzipp n inp in
        zipp (evens op t);

let applyOnOdds n op inp =
       val (a:as) = unzipp n inp in
       zipp (a:(evens op as));
```

The function `applyOnOddL` makes use of the `unriffL` function to divide the bus into two separate buses, grouping the odd signal occurrences into a single bus, and the even signal occurrences into another bus. The parametrised function is applied to the even signals, and the function `riffL` is used to reverse the functionality of `unriffL`.

```
let applyOnOddL f n inp =
        let as = unzipp n inp in
        val (un_odds, un_evens) = unriffL as in
        let f_un_evens = unzipp (n/2) ( f (zipp un_evens) ) in
        zipp (riffL (un_odds, f_un_evens));
```

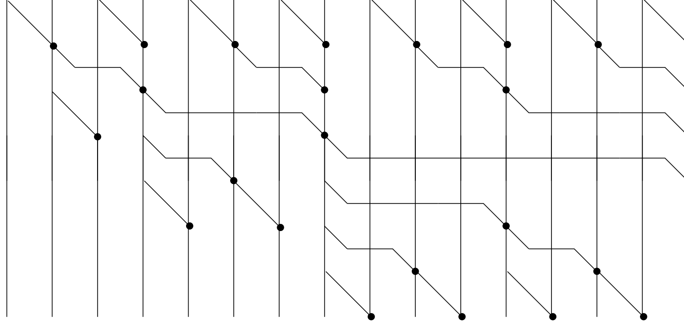

Figure 2: A Slices parallel prefix circuit

The definition for the Slices network is given below, where the functions presented previously are used to compose the prefix network for the general case. At this stage, the circuit is encapsulated in a block, similar to the approach used in the Sklansky definition.

```
letrec pslices n op as =
       let pslices' 1 op as = as
       /\  pslices' 2 op as = val (a:b:cs) = unzipp 2 as in
                              zipp [a, op (zipp (a,b))]
       /\  pslices' n op as =
             let slices  =  applyOnEvens n op ->-
                            applyOnOddL (pslices (n/2) op) n ->-
                            applyOnOdds n op in
             slices as in
       makeBlock (pslices' n op) as;
```

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 semantic 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 semantical 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, such as 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 were developed to define blocks and the placement of these blocks, while our implementation uses quotation constructs to compose lambda expressions, thus delaying the functional application to represent a block in a functional setting. We are currently adding Pebble-style placement constructs to our language.

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 foresee 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 approach is in the signal representation. 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 emphasise the use of marked blocks which we plan to extend for placement and circuit analysis. We still have a number of features unimplemented — such as the lack of 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 the functional meta-programming language *reFLect*. 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 having access to the circuit generators, it is possible to map the structure of the generators to the structure of the resulting circuits in terms of blocks. We plan to add placement constructs similar to those found in Pebble [LM98], to provide a means to describe how circuit blocks are to be placed in relation to each other. We plan to extend

this by adding circuit combinators, similar to the ones used in Ruby [JS94], and thus use the control given to us into looking at the circuit generators to aid the generation of placement hints.

The additional block information to the generated circuit, provides a higher abstract level than the actual circuit, on which compositional model checking techniques and verification can be applied. Furthermore, by analysing the structure of the generator itself, it should be possible to verify properties of a whole family of circuits.

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 optimisation. 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 *reFL^ect*.

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.

[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.

[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.

[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 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.

[She07] Mary Sheeran. Parallel prefix network generation: an application of functional programming. In *Hardware Design and Functional Languages (HFL'07), Braga, Portugal*, 2007.

[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.

View publication stats