THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING IN

A Functional Approach to Hardware Software Co-Design

MARKUS ARONSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Göteborg, Sweden 2018

A Functional Approach to Hardware Software Co-Design.
MARKUS ARONSSON

Thesis for the degree of Licentiate of Engineering 2018:184L
ISSN 1652-8565
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

Chalmers Reproservice
Göteborg, Sweden 2018

A Functional Approach to Hardware Software Co-Design.
MARKUS ARONSSON
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Developing software for embedded systems presents quite the challenge—not only do these systems demand good knowledge of the hardware they run on, but their limited resources also make it difficult to achieve efficiency. For embedded systems with different kinds of processing elements, the challenge is even greater; the presence of heterogeneous elements both raises all of the issues associated with homogeneous systems, and may also cause non-uniform system development and capability.

In this thesis we explore a functional approach to heterogeneous system development, with a staged hardware software co-design language embedded in Haskell, to address many of the modularity problems typically found in such systems. This staged approach enables designers to build their applications from reusable components and skeletons, while retaining control over much of the generated source code. Design exploration also benefits from the functional approach, since Haskell's type classes can be used to ensure that certain operations will be available. As a result, a developer can not only write for hardware and software in the co-design language, but she can also write generic programs that are suitable for both.

Internally, the co-design language is based on a monadic representation of imperative programs that abstracts away from its underlying statement, expression, and predicate types by establishing an interface to their respective interpreters. Programs are thus loosely coupled to their underlying types, giving a clear separation of concerns. The compilation process is expressed as a series of translations between progressively smaller typed languages, which safeguards against many common errors.

In addition to the hardware software co-design language, this thesis also introduces a language for expressing digital signal processing algorithms, using a model of synchronous data-flow that is embedded in Haskell. The language supports definitions in a functional style, reducing the gap between an algorithm's mathematical specification and its implementation. A vector language is also presented, which builds on a functional representation that guarantees fusion for arrays. Both of these languages are intended to be extensions of the co-design language, but neither one is dependent on it and can thus be used to extend other languages as well.

Keywords: functional programming, domain specific languages, signal processing, code generation

Acknowledgements

To my dearest, my fiancée Emma Bogren: because I owe it all to you.

My constant cheerleaders, that is, my parents Dag and Lena: I am forever grateful for your moral and emotional support, you were always keen to know what I was doing and how I was proceeding. Although I'm fairly certain you never fully grasped what my work was all about, you never wavered in your encouragement and enthusiastic inquires. I am also grateful to my sibling Caroline who have supported me along the way, and her wonderful dog Alfons whom never failed to brighten my day.

A very special gratitude goes out to my advisor Mary Sheeran, for your continuous help and support in my studies, for your never ending patience, guidance and immense knowledge. With a special mention to my former co-worker Emil Axelsson, without your precious support I would not have been able to conduct my research. I will always miss our interesting and long-lasting chats.

I am also grateful to Anders Persson, Josef Svenningsson and Koen Claesson for their unfailing support and assistance. Your hard work, ideas and insights in the Feldspar project have proved a great source of inspiration in my research. Finally I express my gratitude to all my other colleagues at Chalmers, who make this a fantastic place to work.

Thank you all for your encouragement!

Thesis

This thesis consists of an extended summary and the following appended papers:

Paper A Markus Aronsson et al. "Stream Processing for Embedded Domain Specific Languages". *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages*. ACM. 2014, p. 8

Paper B Markus Aronsson and Mary Sheeran. "Hardware Software Co-Design in Haskell". *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. ACM. 2017, pp. 162–173

Paper A introduces my signal processing language and an early version of our compilation techniques. I am the lead author but Emil Axelsson wrote the majority of sections 3.1 and 3.2. The paper was awarded the Peter Landin award for best paper at the *International Symposium on Implementation and Applications of Functional Languages, IFL 2014*

Paper B introduces my hardware software co-design language and a mature version of our compiler and representation of imperative programs. The techniques presented are however not restricted to hardware software co-design, and can be of use for developers of embedded languages in general. I am the lead author.

Contents

Part I
Extended Summary

1 Introduction

An embedded system is, in brief, any computer system that is part of a larger system but relies on its own microprocessor. It is embedded to solve a particular task, and often does so under memory and real-time constraints, using the cheapest hardware that can meet the performance requirements. Developing for embedded systems therefore requires good knowledge about the architecture on which a program is supposed to run. Not only should computations be efficient, but they should also take full advantage of the hardware; every line of code counts.

A modern field programmable gate array is an example of an embedded system that integrates co-processor and forms a prototypical heterogeneous computing system. The benefit of a heterogenous system like the gate array is not that several processors are combined, it is rather the incorporation of different kinds of co-processors that each one provides specialized processing capabilities to handle a particular task. A substantial amount of research has today been carried out to find good ways of programming for embedded heterogenous systems, to make them accessible for programmers without experience in embedded hardware or software system design. Hardware description languages are however still the most used tools, along with dialects of C for specific co-processors. While such low-level languages work well for extracting maximum performance from a processor, their portability is severely limited, design exploration is tedious at best, and moving entire programs between C and hardware descriptions is a major undertaking.

A group of languages that show great promise in describing both software and hardware designs are the functional languages. Higher-order functional languages in particular offer an especially useful abstraction mechanism [10, 15, 31] through higher-order functions and lazy evaluation. These features allow for program designs to be treated as first-class objects, and for larger applications to be constructed by composing such designs in a modular fashion; thanks to lazy evaluation, only the relevant parts of a program will show up in the generated source code. Despite the benefits of functional languages, they are rarely considered for embedded system development. One reason for this is the difficulty to give performance guarantees and resource bounds.

This thesis takes the first few steps towards a functional programming language for embedded heterogeneous systems, such as a modern field programmable gate array. Instead of taking on the full challenge of heterogeneous programming head on, a more modest approach is explored: developing a hardware software co-design language, embedding it in Haskell, and seeing how far it goes. The language is staged and utilizes the rich type system of its host language to facilitate design exploration. Furthermore, two languages, one for vector processing and one for signal processing, are introduced to accompany it. These provide useful abstractions for their respective programming domains.

As an example of the co-design language, consider a dot product (also known as a scalar product). The dot product is an algebraic operation that takes two vectors of equal length and returns the sum of all the products of corresponding entries in the two vectors:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{1.1}$$

Using an imperative language like C, the dot product's result could be computed with a single for-loop that iterates over the elements of the two arrays and calculates the sum of their products, one step at a time. Such a sequential solution can be implemented in the co-design language as well:

```
1 dotSeq :: Arr Int32 → Arr Int32 → Program (Exp Int32)
2 dotSeq x y = do
3   sum ← initRef 0
4   for 0 (min (length x) (length y)) $ λix → do
5     a ← getArr x ix
6     b ← getArr y ix
7     modifyRef sum $ λs → s + a * b
8   getRef sum
```

While the above function is faithful to its corresponding implementation in C, its low-level design has forced a focus on implementation details rather than the mathematical specification of the dot product - indices and lengths are both handled manually for instance. A more idiomatic solution is to make use of the new vector language, in which the solution can be expressed as:

```
1 dotVec :: Vec Int32 → Vec Int32 → Program (Exp Int32)
2 dotVec x y = sum (zipWith (*) x y)
```

The summation and element-wise multiplication are now fully handled by the smaller `sum` and `zipWith` functions, respectively.

Vectors are a kind of functional array, represented by a length and an indexing function. Vectors are translated into arrays before compilation and, since arrays are supported by both C and VHDL, `dotVec` can be realized in software and hardware. Most interesting programs for heterogeneous systems do however use a mixture of the software and hardware programs. As an example, assume that `dotVec` should be mapped onto hardware and that a software program then should call it and print its result to standard output. Before `dotVec` can be offloaded, it needs to be given a signature over its input and output channels:

```
1 comp :: Component (Arr Int32 → Arr Int32 → Sig Int32)
2 comp = inputVec 4 $ λx → inputVec 4 $ λy → returnVal $ dotVec x y
```

which consists of two input arrays of length four and a single output signal. Note the arrays instead of vectors in the signature. While `dotVec` expects vectors as input, vectors cannot be transmitted over a network and arrays are used instead, the conversion between the two are handled automatically by `inputVec`.

A hardware component description such as `comp` can automatically be connected to an AXI4-lite interconnect—a standard bus interface— by the compiler, and generate a hardware design that is ready for synthesis. It is then possible to reach `comp` from software through, for example, memory-mapped I/O. The general idea is that a memory-mapped hardware component will share its address space with the software program and can be called as a regular function:

```
1 prog :: Software ()
2 prog = do
3   dot ← mmap "0x43C00000" comp
4   xs  ← initArray [1,2,3,4]
5   ys  ← initArray [5,6,7,8]
6   r   ← newRef
7   call dot (xs .: ys .: r .: nil)
8   res ← getRef r
9   printf "sum: %d" res
```

where `mmap` initiates the memory-mapping of `comp` with its address, which is obtained during synthesis, and `call` then calls the mapped component and stores its result in `r`.

2 Background

Today we see embedded systems consisting of everything from general purpose processors (GPPs) to application specific integrated circuits (ASICs). GPPs and ASICs represent two extremes of available architectures, and somewhere in the middle, field programmable gate arrays (FPGAs) can be found. FPGAs provide the best of both worlds: they are close to hardware and can be reprogrammed [12]. Modern FPGAs also contain various components and co-processors which may be specialized to handle a certain task well, and they have a good performance per Watt ratio. These properties have made them increasingly popular to use in high-performance, computationally intensive systems [39].

While a modern FPGA shows great promise as a prototypical system for heterogeneous computing, its adoption has been slowed down by the fact that it is difficult to program. The logic blocks of an FPGA are usually programmed in a hardware description language, while its co-processors are programmed in some low-level dialect of C or even assembler. Low-level languages are typically used for embedded systems since they give a designer fine control over the system's capabilities. Such fine control does however come at a cost—a programmer cannot abstract away from the specific system architecture, but must keep the implementation on a low level during the entire design process. Other issues, related to functionality and modularity, come as consequences of the lack of abstraction.

In his paper "Why functional programming matters" [35], Hughes argues that many of the problems with low-level languages can be addressed by making use of functional programming. In particular, the glue code that functional programming languages offer (through higher-order functions and lazy evaluation) enables building useful combinators. The benefits of a functional programming language are not limited to describing software, as Sheeran shows in her paper "Hardware Design and Functional Programming: a Perfect Match" [50]. Sheeran exemplifies how a functional language can make it easy to explore and analyze hardware designs in a way that would have been difficult, if not impossible, in traditional hardware description languages.

2.1 Functional Programming

Functional programming is based around the application of a function to its arguments. In this programming style, a program is written as a function that accepts input and delivers its result. That function itself is defined in terms of smaller functions, which in turn are defined using smaller functions still and, in the end, a function consists of nothing but language primitives.

An important distinction between the functions in a functional programming language and functions in, say an imperative language like C, is that these always return the same value for the same arguments. It is said that functional programs have no side effects, and this makes it possible for functions to be safely evaluated in parallel, as long as their data dependencies are satisfied.

A function that accepts other functions as arguments is often referred to as a higher-order function, or a combinator, and provides a useful piece of glue code that lets a programmer build complex functions from smaller ones. In Haskell, a number of such higher-order functions are provided by its standard libraries. One of these is `map`, which can be defined as follows:

```
1 map :: (a → b) → [a] → [b]
2 map f []     = []
3 map f (x:xs) = f x : map f xs
```

The first line specifies the type of `map`, because, in Haskell, every function is assigned a static type in an effort to attain safer programs—if a function is applied to a value that does not match its type, the compiler will reject the function and instead point out the type mismatch. In the case of `map`, its type is a function that takes two arguments: a function `f :: a → b` and a list `xs :: [a]`, it returns a list of elements of type `b`. The second line of `map` specifies that it, when given an empty list as denoted by `[]`, returns another empty list. The third line states that for non-empty lists, `f` should be applied to the list's head and a recursive call should be made with the tail of the list as an argument.

The usefulness of higher-order functions like `map` comes from their ability to encode common patterns: `map` works for all functions and lists that fit its type signature. Functions like `map` are often referred to as combinators—a style of organizing libraries around a few primitive values and functions for combining them. These combinators allow for complex structures to be built from a set of smaller functions. For example, the dot-product from section 1 is composed by two combinators: `zipWith`, a generic way of joining two vectors, and `sum`:

```
1 zipWith :: (a → b → c) → Vec a → Vec b → Vec c
2 zipWith f a b = fmap (uncurry f) $ zip a b
3
4 sum :: Vec Int → Int
5 sum = fold (+) 0
```

`fmap` here is similar to the above `map` but works for vectors instead of lists, `zip` joins two vectors into a single vector of pairs, and `fold` reduces a vector into a scalar value using addition and starting at zero.

The other piece of glue code that functional programming languages provides is often

referred to as function composition, and enables programs to be glued together. Say that `f` and `g` are two programs, then `g` composed with `f` is written `g . f` and is a program that, when applied to its input `x`, computes `g (f x)`. In Haskell, we can define function composition as:

```
1 (.) :: (b → c) → (a → b) → a → c
2 (.) g f x = g (f x)
```

where parentheses around the dot imply that function composition is an infix function.

While the size of the intermediate result of `f` could potentially spoil any usefulness of the composition, functional programming solves this by only evaluating `f` as much as is needed by `g`. This property is referred to as lazy evaluation and lets us fuse functions without creating any unnecessary, intermediate values. Its benefits extend to embedded types as well, and guarantees fusion of vectors.

This section has given a brief overview of functional programming in Haskell and its beneficial properties for embedded languages. So far, the distinction between regular and embedded Haskell has yet to be made. The following section introduces the concept of domain specific languages, and explains what it entails to be an embedded language in Haskell.

2.2 Domain Specific Languages

A domain specific language (DSL) is a special-purpose language, tailored to a certain problem and capturing the concepts and operations in its domain. For instance, a hardware designer might write in VHDL, while a web-designer who wants to create an interactive web-page would use JavaScript. DSLs come in two fundamentally different forms: external and internal, where VHDL and JavaScript are both examples of the former.

Internal DSLs are embedded in a host language, and are often referred to as embedded domain specific languages (EDSLs). Haskell, with its static type system, flexible overloading and lazy semantics, has come to host a range of EDSLs [26]. For instance, popular libraries for parsing, pretty printing, hardware design, and testing have all been embedded in Haskell [37, 34, 15].

EDSLs in Haskell are further divided into one of two kinds: shallow or deep. Conceptually, a shallow embedding captures the semantics of the data in a domain, whereas a deep embedding captures the semantics of the operations in a domain. Both kinds of embeddings have their own benefits and drawbacks. To illustrate the differences between shallow and deep embeddings, a small example domain can be implemented:

```
1 type Exp = Int
2
3 const :: Int → Exp
4 const a = a
5
6 times :: Exp → Exp → Exp
7 times a b = a * b
```

where `Exp` is a short-hand for expressions and is defined as a type synonym for integers, thus an example of a shallow EDSL. Two functions are also provided: `const` to lift integer literals, and `times` to multiply expressions.

Two benefits of a shallowly embedded language like `Exp` are that it is easy to add new functions and that evaluation is straightforward—the a value of type `Exp` is the result of some expression. On the other hand, it is difficult to compile shallow types as there is no representation of the expression that built its value. It is easier to compile an embedded language if its functions instead return an intermediate representation of their result, which sits between Haskell and the compiled code [26]. This technique is known as deep embedding, and `Exp` can be reimplemented using it:

```
1 data Exp = Const Int | Times Exp Exp
2
3 const :: Int → Exp
4 const a = Const a
5
6 times :: Exp → Exp → Exp
7 times a b = Times a b
```

where `Exp` is now a datatype that lists all supported expressions, which `const` and `times` use to construct their results.

As values in a deeply embedded language like `Exp` are representations of the expressions that built them, rather than their result, it is possible to interpret them and, for example, define a function that evaluates them into integers:

```
1 eval :: Exp → Int
2 eval (Const a)   = a
3 eval (Times a b) = (eval a) * (eval b)
```

The ability to interpret values come at the cost of making it harder to add new functions over `Exp` without first extending its datatype.

While the implementation of shallow and deep embedding are usually at odds, there has been work done in order to combine their benefits [55]. The co-design language makes use of such a combination of deep and shallow embeddings: its core datatype is implemented using a deep embedding and user facing libraries use shallow embeddings built on top of the core. This mixture of embeddings ensures that the core is easy to interpret while simultaneously allowing user-facing libraries to provide a nice and extensible syntax.

This section and the previous one have given a brief overview of functional programming and domain specific languages, showcasing Haskell and the benefits its functional style provide for embedded languages. The next section introduces the co-design language through a few examples and highlights these benefits.

2.3 Embedded Programming in Haskell

Programming in a functional language like Haskell is quite different from the imperative style of programming used in a language like C. As an example of these differences, consider a finite impulse response (FIR) filter, one of the two primary types of digital

filters used in digital signal processing applications [47]. The mathematical definition of a FIR filter of rank N is as follows:

$$y_n = b_0 x_n + b_1 x_{n-1} + \cdots + b_N x_{n-N} = \sum_{i=0}^{N} b_i x_{n-i} \tag{2.1}$$

where x and y are the input and output signals, respectively, and b_i is the value of the impulse response at time instant i. The inputs x_{n-i} are sometimes referred to as "taps", since they *tap into* the input signal at various time instants.

The FIR filter can be implemented in C as:

```
1  void fir(int N, int L, double *b, double *x, double *y) {
2   int j, k;
3   double tap[256];
4   for(j=0; j<N; j++) tap[j] = 0.0;
5   for(j=0; j<L; j++) {
6    for(k=N; k>1; k--) tap[k-1] = tap[k-2];
7    tap[0] = x[j];
8    y[j] = 0.0;
9    for(k=0; k<N; k++) y[j] += b[k] * tap[k];
10  }
11 }
```

where N is the filter rank, L is the size of the input, and b, x, and y are pointers to the filter's coefficients, input, and output, respectively.

At first glance, the C code seems to be a good representation of the FIR filter, but there are a few problems with its implementation. For example, the for-loop on line 9 calculates a dot-product of the arrays b and *tap* inline. It is possible to extract the product:

```
1 double dot(int N, double *xs, double *ys) {
2   double sum = 0;
3   for (int i=0; i<N; i++) sum += xs[i] * ys[i];
4   return sum;
5 }
```

but it is still specialized to values of type *double*, it assumes that b and *tap* both have at least N elements, and it is not compositional in the sense that it cannot be merged with the producers of `xs` or `ys` without looking at their implementation first.

The same dot product can be implemented in the co-design language, using a similar, but not idiomatic, imperative style:

```
1 dotSeq :: Arr Float → Arr Float → Program (Exp Float)
2 dotSeq x y = do
3   sum ← initRef 0
4   for 0 (min (length x) (length y) $ λix → do
5     a ← getArr x ix
6     b ← getArr y ix
7     modifyRef sum $ λs → s + a * b
8   getRef sum
```

Note that `dotSeq` returns a program, which in turn returns a floating point expression. Programs are a type of *monad*, that is, they are a kind of composable computation description; the functions that manage references and arrays are all monadic, and they are sequenced to look like an imperative program by using Haskell's `do` notation.

`dotSeq` is not without its own faults, as it is limited to floating point expressions and indices are given manually. The first of these issues can be resolved by Haskell's type class for basic numerical operations, called `Num`. With it, `dotSeq` can be made polymorphic in the kind of values it accepts but still restricted to numerical values that support the required operations:

```
1 dotSeq :: Num a ⇒ Arr a → Arr a → Program (Exp a)
```

In order to address the manual indexing of `dotSeq`, the idiomatic approach would be to reimplement the dot product using the vector language instead:

```
1 dotVec :: Num a ⇒ Vec a → Vec a → Exp a
2 dotVec xs ys = sum (zipWith (*) xs ys)
```

A dot product based on vectors is not only closer to its mathematical specification, but also sturdier in the sense that it is harder for users to make an error: indices and lengths are now hidden by vector functions. Furthermore, Haskell's lazy evaluation ensures that `dotVec` can be merged freely with the producers of `xs` and `ys`.

Compiling `dotVec` to C with two small example inputs and printing its result to standard output produces the following code (with imports omitted for brevity):

```
1  int main() {
2    uint16_t _a0[] = {1, 2, 3, 4}, *a0 = _a0;
3    uint16_t _a1[] = {4, 3, 2, 1}, *a1 = _a1;
4    uint16_t state2 = 0;
5    uint32_t v3;
6    for (v3 = 0; v3 < 4; v3++)
7      state2 = a0[v3] * a1[v3] + state2;
8    fprintf(stdout, "result: %d\n", state2);
9    return 0;
10 }
```

The vector language excels at describing array transformations, but it has some difficulty in describing the kinds of recurrence equation that makes up a FIR filter. Nevertheless, a few such recurrence equations are provided by the vector library, one of which is the `recurrenceI` function that can be used to implement the filter:

```
1 firVec :: Num a ⇒ Vec a → Vec a → Program (Arr a)
2 firVec cs v = recurrenceI (replicate (length cs) 0) v $ λi → dotVec cs i
```

The recurrence function takes an initial buffer to store old inputs in, a vector to iterate over, and a step function that produces one output at a time given these two inputs. Compiling `firVec` to C yields the following code, where imports and code unrelated to the filter have been omitted:

```
int main() {
  r5 = 0;
  for (v6 = 0; v6 ≤ 3; v6++) {
    a3[r5] = a1[v6];
    r5 = (r5 + 1) % 4;
    state7 = 0;
    for (v8 = 0; v8 < 4; v8++)
      state7 = a0[v8] * a3[(4 + r5 - v8 - 1) % 4] + state7;
    a2[v6] = state7;
  }
}
```

where `a0` contains the coefficients, `a1` the input array, and `a3` is the input buffer. Note that, since all inputs are present at once, it is possible to rewrite the filter and do without the queue for vectors for all but the first initial segments of the input:

```
firQ :: Num a ⇒ Vec a → Vec a → Vec a
firQ coeff = map (dotVec coeff . reverse) . tail . inits
```

Both `firQ` and `firVec` showed that an implementation of the FIR filter based on vectors is certainly possible, but they required that either the filter was rewritten to fit the vector library, or that a helper function for recurrence equations was available. Another approach is to instead use the signal processing language. Signals are possibly infinite sequences of values that carry a notion of time, that is, it is possible to prepend a value to a signal, effectively delaying its previous output by one time unit. As an example, the FIR filter can be implemented with signals as follows:

```
firSig :: Num a ⇒ [Exp a] → Sig a → Sig a
firSig coeffs = sums . muls coeffs . dels 0
```

where `sums`, `muls`, and `dels` implement the three main components of the filter, that is, a summation, a multiplication with coefficients, and a number of successive delays access earlier inputs.

```
sums :: Num a ⇒ [Sig a] → Sig a
sums as = foldr1 (+) as

muls :: Num a ⇒ [Exp a] → [Sig a] → [Sig a]
muls as bs = zipWith (*) (map constant as) bs

dels :: Exp a → Sig a → [Sig a]
dels e as = iterate (delay e) as
```

`constant` and `delay` are signal functions that introduce a constant signal and a unit delay, respectively.

From a hardware perspective, `firSig` is arguably closer to the FIR filter's mathematical specification than `firVec`: the input signal is iteratively delayed to form the filter's taps, each tap is then multiplied with a coefficient, after which the taps are summed to form the filter's output. Compiling `firSig` to VHDL produces the following hardware description:

```
ENTITY comp0 IS
  PORT (in0 : IN unsigned (7 DOWNTO 0);
        out1 : OUT unsigned (7 DOWNTO 0);
        clk : IN std_logic;
        rst : IN std_logic) ;
END ENTITY comp0 ;
ARCHITECTURE behav OF comp0 IS
  SIGNAL state2 : unsigned (7 DOWNTO 0) ;
  SIGNAL state2_d : unsigned (7 DOWNTO 0) := "00000000" ;
BEGIN
  l8 :
    PROCESS (in0) IS
      VARIABLE v3, v4, v5, v6, v7 : unsigned (7 DOWNTO 0) ;
    BEGIN
      v3 := "00000001" ;
      v4 := "00000010" ;
      v5 := resize (v3 * in0, 8) ;
      v6 := resize (v4 * state2_d, 8) ;
      v7 := resize (v5 + v6, 8) ;
      state2 ⇐ in0 ;
      out1 ⇐ v7 ;
    END PROCESS l8 ;
  l9 :
    PROCESS (clk) IS
    BEGIN
      IF rising_edge (clk) THEN
        state2_d ⇐ state2 ;
      END IF ;
    END PROCESS l9 ;
END ARCHITECTURE behav ;
```

2.4 Summary

Section 1 gave an introduction to heterogeneous embedded systems as an interesting development towards energy efficient computing. These systems are not without challenges, as the presence of multiple processing elements raises all of the issues involved with homogeneous systems in addition to the issues of heterogeneity in the system. A modern FPGA was presented as a prototypical heterogeneous system of note.

Functional languages were proposed in section 2 as a way to address issues typically found in embedded system design, in particular the modularity issues that come from the low-level languages they are usually programmed with. Section 2.1 gave a brief overview of functional programming and section 2.2 went on to demonstrate techniques for embedded languages in Haskell. Finally, section 2.3 showcased the co-design language, the current attempt at bringing the benefits of functional programming languages to the domain of embedded heterogeneous systems.

The remainder of this thesis covers the co-design, vector, and signal processing languages in detail and highlights the techniques they are built on. In particular, the following contributions are made:

- We present a language for hardware software co-design that is embedded in Haskell and designed with modern FPGA programming in mind, able to generate both C and VHDL for the FPGA's various processing elements, including the necessary glue code for connections between elements. Also, the language is extensible to support any differences in instruction sets between elements.

- We present two extensions to the hardware software co-design language, one for vector computations and another for signal processing. The vector language supplements an array type with vector types and combinators that support fusion, and the signal processing language adds support for synchronous data-flow to an expression type. Both extensions are intended to be used with the co-design language, but abstract over their underlying types and can be used with other embedded languages as well.

- We present the techniques used to implement a language like the co-design language, that is based on a monadic representation of imperative programs. The model is loosely coupled to its underlying statement, expression, and predicate types, enabling each type to be defined separately. Compilation of programs is defined as a typed translation between progressively smaller languages, which not only safeguards against common errors in untyped translations, but also provides control over the generated code.

3 Hardware Software Co-Design

Starting with a single Haskell program, the hardware software co-design library is designed with three main tasks in mind: generate C for the software parts, VHDL for the hardware parts, and a combination of C and VHDL for the transmission of data between processing elements. While C and VHDL are different from one another in that one describes sequential software and the other parallel hardware, both languages can be described with an imperative style of programming. As a consequence, the co-design language is based on a monadic representation of imperative programs.

The general idea behind a monadic representation of imperative programs is that one can view an imperative program as a sequence of instructions to be executed on some machine, which looks similar to functions written in a stateful monad. In fact, statements written in a stateful monad can be directly translated into statements in an imperative language. As an example of these similarities, consider the following program for reversing an array in place:

```
1 rev :: SArr Int32 → Software ()
2 rev arr = for 0 (len `div` 2) $ λix → do
3    aix ← getArr arr ix
4    ajx ← getArr arr (len - ix - 1)
5    setArr arr ix ajx
6    setArr arr (len - ix - 1) aix
7   where
8     len = length arr
```

Note that `rev` is a software program, as told by its type, but its implementation is not specific to software: for-loops and arrays are part of both C and VHDL. The function could just as well have been implemented in hardware. In fact, a hardware version of `rev` can be defined by swapping its software types with their corresponding hardware types:

```
1 rev :: HArr Int32 → Hardware ()
```

The fact that `rev` can be implemented in both software and hardware by simply changing its type does imply that labeling it as either is unnecessarily restrictive. Programs that are constrained by the functionality they require, rather than a specific language, can be described with type classes provided by the co-design language:

```
1 rev :: (Monad m, Arrays m, Control m, Type m Int32) ⇒ Arr m Int32 → m ()
```

The now generic `rev` substitutes `SArr` and `HArr` for an `Arr m`, the array type associated with monad `m`, and introduces three type classes: `Arrays`, which defines the `Arr` type and its related functions; `Control`, for control flow operations like a for-loop; and `Type`, which ensures a type is representable. Parts of these classes are defined as follows:

```
1  class Monad m ⇒ Arrays m where
2    type Arr m
3    newArr :: Type m a ⇒ Exp m Length → m (Arr m a)
4    getArr :: Type m a ⇒ Arr m a → Exp m Index → m a
5    setArr :: Type m a ⇒ Arr m a → Exp m Index → a → m ()
6
7  class Monad m ⇒ Control m where
8    for :: (TypeM m a, Integral a) ⇒ Exp m a → Exp m a → (Exp m a → m ())
9        → m ()
```

where each class lists the functions it provides and in the case of arrays, the type to use with them; `Exp` represents the expression type associated with `m`. The following short-hand for a collection of purely computational operations is defined in order to keep types short:

```
1 type Comp m = (Monad m, References m, Arrays m, Control m)
```

In addition to the collection of classes in `MonadComp`, there are also classes of operations that only one of the two languages support. The type classes there form a hierarchy with monads at the base. Classes intended for either the software or hardware branches also require the type is an extension of their respective monads. For example, the function for a hardware process is defined by the following type class:

```
1 class HardwareMonad m ⇒ Process m where
2   process :: m () → m ()
```

The co-design language is intended to provide a convenient model of imperative programs and, as was shown in section 2.3, also serves as a base up which extensions like the vector and signal processing languages can be built. Furthermore, the ability to write generic programs facilitates design exploration, and while they are not always ideal as hardware descriptions—purely computational instructions do not support, for instance, pipelining—once a layout has been decided, it is easy to fix a program's type to hardware and optimize its implementation.

3.1 Imperative Model of Programs

The co-design language is inspired by the work of Svenningsson and Svensson [54] and by the Operational Monad [1]: the program type is deeply embedded in order to capture a computation as an algebraic data type and parameterized on the instructions used in said computations. In addition to instructions, the new program type is also parameterized on a list of other types associated with a program:

```
1 data Program instr fs a
```

where the type-level list `fs` could include, for example, the types of subprograms and expressions and a type predicate.

As an instruction's effect will only depend on its interaction with other instructions, they can safely be separated from their sequencing as programs. The task of implementing a program type is therefore equivalent to writing an interpreter for its instructions. One such interpreter, that maps programs to their intended meaning as a monad, is provided by the co-design library:

```
1 interpret :: (Interp i m fs, HFunctor i, Monad m) ⇒ Program i fs a → m a
```

`interpret` lifts a monadic interpretation of instructions, which may be of varying types, to a monadic interpretation of the whole program. By using different types for the monad `m`, it is possible to implement different "back ends" for programs. For example, interpretation in Haskell's `IO` monad creates a way to *run programs*, while interpretation in a code generation monad can be used to make a *compiler*.

For some instruction type `instr`, its interpretation into a monad `m` is given by:

```
1 class Interp instr m fs where
2   interp :: instr '(m, fs) a → m a
```

where `'(m, fs)` is type level parameter list of two elements, `m` and `fs`. To extend this interpretation to entire programs, the interpreter must be able to traverse the instruction and recursively apply itself to all of its subprograms. `HFunctor` captures this constraint:

```
1 class HFunctor h where
2   hfmap :: (∀b . f b → g b) → h '(f, fs) a → h '(g, fs) a
```

Instructions are parameterized on the type of their subprograms to facilitate a compositional definition of them that works for both simple instructions and control instructions, using a technique like Data Types à La Carte [57]. As both `Interp` and `HFunctor` can be instantiated for a combination of instructions that in turn support their respective class, new instructions can be defined and given an interpretation without worrying about whatever instruction set of which they are part. Instructions that can be extended in this way are quite useful to the co-design language, as each computational element usually come with its own set of operations.

An instruction that models if-statements can be implemented as follows:

```
1 data If fs a where
2   If :: exp Bool → prog () → prog () → ControlCMD (P3 prog exp pred) ()
```

where `prog` refers to sub programs, `exp` to expressions, and `pred` is a type predicate (although `pred` is not used by `If`, it is often included to keep its type consistent with other instructions that do use it). `P3` is synonym for a type-level list of three arguments. The `HFunctor` instance for `If` is straightforward:

```
instance HFunctor If where
  hfmap f (If c thn els) = If c (f thn) (f els)
```

A program can now add `If` to its set of instructions:

```
type Prog = Program (Old :+: If) (P2 Exp Pred)
```

The program applies itself, `Exp`, and `Pred` as arguments for its instruction set. If the old instruction set supported interpretation in, for instance, a C code generation monad, then `If` must support the same interpretation before `Prog` can be compiled again. Assuming `Exp` can compiled to C, the compilation of `If` could be defined as:

```
instance Interp If CGen (P2 Exp Pred) where
  interp (If b tru fls) = do
    cc ← compExp b
    addStm [cstm| if ($cc) {$items:tru} else {$items:fls} |]
```

where `CGen` is the C code generation monad, and `addStm` adds the quoted C statement to the code generator—quotation is provided by the Haskell package *language-c-quote*.

`interp` is by no means the only interpretation available for programs. For example, the above `If` type has two sub-structures that can be mapped, `prog` and `exp`, and is therefore a higher-order *bi-functor*:

```
class HFunctor h ⇒ HBifunctor h where
  hbimap :: (Functor f, Functor g)
    ⇒ (∀b . f b → g b)
    → (∀b . i b → j b)
    → h '(f, '(i, fs)) a
    → h '(g, '(j, fs)) a
```

This special kind of functor is beneficial in that it enables an interpretation scheme that decouple the interpretation of instructions from that of expressions:

```
interpretBi :: (InterpBi instr m fs, HBifunctor instr, Functor m, Monad m)
  ⇒ (∀b . exp b → m b) → Program instr '(exp, fs) a → m a
```

3.2 Pure Expressions

An embedding based on monads provides a representation of the statements for imperative languages, but most meaningful languages also include a notion of pure expressions. These expressions contain a combination of one or more values, constants, variables, and operators that take two or more expressions. As a small example of such expressions in the co-design language, consider a function for computing the distance between two points in a plane:

```
dist :: (SExp Float, SExp Float) → (SExp Float, SExp Float) → SExp Float
dist (x1, y1) (x2, y2) = sqrt (dx**2 + dy**2)
  where
    dx = x1 - x2
    dy = y1 - y2
```

`dist` certainly has the look and feel of an ordinary Haskell expression, but wraps its result `a` in the expression type `exp`. In the co-design language, expressions have a deeply embedded core syntax, and a type of `exp a` is therefore a computation that produces a value of type `a` rather than just a value. The pairs are purely syntactical sugar, as they are not part of the expression syntax and will be removed during interpretation. Also, while `dist` is implemented using `SExp` for software expressions, it could have been defined as a generic expression as well:

```
dist :: (Floating (exp a), Num (exp a), Syntax exp a) ⇒
        (exp a, exp a) → (exp a, exp a) → exp a
```

In addition to the standard numerical and floating point operations, the co-design language also provides a number of higher-order abstractions, like let-bindings:

```
class Let exp where
  share :: (Type exp a, Type exp b) ⇒ exp a → (exp a → exp b) → exp b
```

Such abstractions are one of the hallmarks of functional programming and let users avoid unnecessary detail and mundane operations. Abstractions can however complicate the compilation process, as they distance the language from its generated source code. The co-design language addresses this problem by using two expression types: one with only primitive operations that is easy to compile, and one that includes higher-order features. The idea is then to provide an elaboration from the feature rich expressions to program snippets over primitive expressions:

```
elaborateSExp :: SExp a → Program SIns (P2 CoreExp SPred) (CoreExp a)
```

Wrapping the primitive expressions in programs is necessary to elaborate, for instance, let-bindings, as those are replaced with reference statements.

`elaborateSExp` provides a means to elaborate a single software expression. In order to elaborate an entire program, each instruction must be traversed to find and elaborate any expressions it contains. However, as the result of `elaborateSExp` is monadic, this traversal can be described as interpretation of programs with rich expressions to programs with primitive expressions. This idea is captured by the following function:

```
elaborate :: HBifunctor ins1
    ⇒ (∀b . exp1 b → Program ins2 '(exp2, fs) (exp2 b))
    → Program ins1 '(exp1, fs) a → Program ins2 '(exp2, fs) a
```

Compilation, for instance, can now be defined in two steps: first an elaboration to remove higher-order expressions, then an interpretation of the resulting program. This staged approach has a few benefits: it is typed, which rules out many potential errors, and it is easier to write than a complete translation into source code.

3.3 Mixing Hardware and Software

The hardware software co-design language aims to generate all the code necessary to program a modern FPGA, which includes the communication between processing elements. This communication is typically done over an AXI4 interconnect. Full AXI4 offers a range of interconnects that include variable data and address bus widths, high bandwidth burst and cached transfers, and various other transaction features that are useful for streaming.

A lighter version of the AXI4 interconnect is offered through AXI4-lite, which is a subset of the full specification that forgoes the streaming features for a simpler communication model that writes and reads data one piece at a time. While a full AXI4 interconnect certainly is useful, implementing one in the co-design language is still future work. An implementation of the AXI4-lite interconnect is however provided:

```
1  axi4lite :: AXIPred a
2    ⇒ Component a
3    → Component (
4            Sig (Bits 32) -- Write address.
5          → Sig (Bits 3)  -- Write channel protection type.
6          → Sig Bit       -- Write address valid.
7          → Sig Bit       -- Write address ready.
8          → Sig (Bits 32) -- Write data.
9          → Sig (Bits 4)  -- Write strobes.
10         → Sig Bit       -- Write valid.
11         → Sig Bit       -- Write ready.
12         → Sig (Bits 2)  -- Write response.
13         → Sig Bit       -- Write response valid.
14         → Sig Bit       -- Response ready.
15         → Sig (Bits 32) -- Read address.
16         → Sig (Bits 3)  -- Protection type.
17         → Sig Bit       -- Read address valid.
18         → Sig Bit       -- Read address ready.
19         → Sig (Bits 32) -- Read data.
20         → Sig (Bits 2)  -- Read response.
21         → Sig Bit       -- Read valid.
22         → Sig Bit       -- Read ready.
23         → ())
```

`axi4lite` takes any hardware component of type `a`, assuming `a` can be transmitted over wires, and automatically connects it to an AXI4-lite interconnect. The signals generated by `axi4lite` all follow the AXI4 naming standard, and should as such be recognized by most hardware synthesizers as a valid interconnect. In case the detection does not work, or simply is not supported, it is also possible to add the component manually to a design, as shown in Appendix A.

Hardware components on the FPGA can interface with each other using port-maps as usual, but a component that is wrapped in an AXI4 interconnect can also be accessed from software with the help of memory-mapped I/O. The general idea is that memory-mapping into a component's physical address causes it to share its address space with the memory of whatever software program is running, that is, the component can be reached by simply reading and writing to pointers into its address. The necessary memory-mapping to

connect to a component from software is done by the `mmap`:

```
1 mmap :: Address → Component a → Software (Pointer (Soften a))
```

Note that `mmap` "softened" the component's type by translating its hardware types to their corresponding types in software. A softened pointer can be called from software with a matching set of arguments:

```
1 call :: Pointer a → Argument a → Software ()
```

As an example of offloading a program to hardware and interfacing with it from software, consider the sequential dot product from section 2.3 implemented as a hardware program with the following type:

```
1 dotSeq :: HArr Int32 → HArr Int32 → Hardware (HExp Int32)
```

Types are not first-class values in Haskell, and cannot be inspected by other functions. In order to connect `dotSeq` to an AXI4 interconnect, it must first be given a type signature declaration that is represented using a regular datatype within Haskell, which then can be inspected by other functions. An example of such an signature is given for `dotComp`:

```
1 dotComp :: Component (HArr Int32 → HArr Int32 → Signal Int32 → ())
2 dotComp = inputArr 3 $ λa → inputArr 3 $ λb → outputVal $ dotSeq a b
```

which declares a signature of two input arrays, both with a length of three, and a combinatorial output given by `dotSeq`.

To offload `dotSeq` to hardware, it is first wrapped in an AXI4-lite interconnect by calling `axi4lite`, it is then compiled to a VHDL design and synthesized and, lastly, the generated bit stream is loaded onto, for example, the FPGA's programmable logic. During synthesis, the physical address of the design is established. This address can then be accessed from software with the help of `mmap` and `call`:

```
1 program :: Software ()
2 program = do
3   dot ← mmap "0x4C300000" dotComp
4   arr ← initArr [1 .. 3]
5   brr ← initArr [4 .. 6]
6   res ← newRef
7   call dot (arr :>> brr :>> res :> nil)
8   val ← getRef res
9   printf "%d\n" val
```

Here, `(:>>)` adds an arrays to an argument list, `(:>)` adds a reference, and `nil` represents an empty list. The type of `call` ensures that `dot` is called with the correct arguments, after which the result can be read from `res`. The memory mapping only needs to be done once to bring a pointer to the hardware component into scope, whereas calling a component can be done any number of times.

The example `program` can be compiled to software as any other program, but the "function call" to `dot` will be replaced by a few pointer operations istead:

```
1  void program() {
2    f_map(0x43C00000, (void **) &pointer_dot, &offset_dot);
3    pointer_dot = pointer_dot + offset_dot;
4    int * input_dot1  = pointer_dot;
5    int * input_odt2  = pointer_dot + 4;
6    int * output_dot1 = pointer_dot + 9;
7    ...
8    for (i=0;i<4;i++) {
9      *(input_dot1 + i) = a1[i];
10     *(input_odt2 + i) = a2[i];
11   }
12   printf("%d\n", *(output_dot1));
13 }
```

where `f_map` is a C function that performs all the necessary memory-mapping setup for `pointer_dot`, and is generated by the co-design language's software compiler. The general idea of these pointers are that they provide the different address into the hardware components registers (for example, `*(input_dot1 + 2)` refers to the third element in the first input array). Underneath these pointers lies a buffer that the AXI4 bus will read and forward any values to `dot` in a synchronous manner.

Running the above code on one of the embedded ARM cores will only provide a slight speedup, if any, when compared to a dot-product fully implemented in software. This is however expected, as the computational weight of dot product is quite small and does not necessarily outweigh the communication cost of sending the arrays to hardware. For larger examples where the computation being offloaded far outweighs its communication cost, as those presented in Paper B, the mixed software and hardware approach can and does provide a bigger benefit.

3.4 Vectors

A sequential program in the co-design language makes use of its array type to express array and vector computations with mutable updates. Arrays provide full control over their allocation and assignment, but do so through a low-level and imperative interface. As shown in section 2.3, some functions are better expressed in a compositional manner than as a sequential program. For such cases, the vector language provides a useful set of combinators.

Vector computations typically start with a *manifest* vector, that is, a vector that refers directly to an array stored in memory. Vector operations are then applied, where each operation is overloaded to accept any *pully* vector as input and produces another pully vector. Then, once the various vectors have been constructed, they are assembled into a *pushy* vector and written to memory, resulting in a new manifest vector.

The names for manifest, pull and push vectors draw inspiration from the Pan language [26] and push arrays [23], where pully vectors are all pull-like types, that is, types that have a length and support indexing, and pushy vectors are all types that can be converted into push vectors. The distinction between, for instance, pull vectors and pully vectors is made since there are cases where its preferable to "skip" parts of the typical

cycle of vector computations. Also, note that pushy and pully vectors are both methods of computing arrays, rather than elements stored in memory like manifest vectors.

A pully vector consists of a length—the number of elements in the vector—and a function that, given an index in the vector, returns an element. Furthermore, pull vectors are designed in such a way that all operations fuse together without creating any intermediate structures in memory, a property which is often referred to as vector fusion. Pull vectors are represented as:

```
1 data Pull exp a where
2   Pull :: exp Length → (exp Index → a) → Pull exp a
```

Push vectors go in the opposite direction of pull vectors, and provide control over a vectors evaluation to the producers rather than the consumer, that is, pushy vectors have a representation that supports nested writes to memory and fusion of operations. A `Push` vector is represented as:

```
1 data Push m a where
2   Push :: Exp m Length → ((Exp m Index → a → m ()) → m ()) → Push m a
```

A push vector consists of a length, as in pull vectors, but its function describes how elements are evaluated rather then how they are fetched. As such, they are parameterized on the monad `m` rather than an expression language; `Exp` is an associated type that refers to the expression type associated with `m`. Push arrays implement efficient concatenation and interleaving, which would otherwise introduce unnecessary conditionals had they been implemented with pull vectors instead.

As an example of vectors, consider the sum of the squares of all numbers from zero to n:

```
1 squares :: (Num a, Type exp a) ⇒ exp a → exp a
2 squares n = sum $ map (λx → x * x) (1 ... n)
```

Note that no vector occurs in the function's type, but they are used internally to compute the result: the infix function `(...)` constructs a pully vector with values ranging from one to n, to which a mapping is applied that squares each element. The vector is then consumed by `sum`, which turns it into a single value.

Each vector type has a different set of operations associated with it, and each type only supports those operations that can be performed efficiently for that type (to find out more, read [23]). In many cases, the vector type is guided by the types of the operations involved, and follows the typical pattern of a manifest vector being turned into a pull vector, which turns into a push vector, which is then written to memory and thus turns back into a manifest vector. There are however cases where its preferable to skip parts of this cycle. For instance, the `squares` function starts with a pull vector rather than a manifest vector.

3.5 Signal processing

The signal language is based on the concept of signals: possibly infinite sequences of values in some pure expression language, given by the type `Sig`. Conceptually, signals

can be thought of as infinite lists. Unlike lists however, a signal is not a first-class value and cannot be nested—it is impossible to construct a signal that produces other signals. Programming with signals is done compositionally, that is, a signal program is a collection of mutually recursive signal functions, each built from repeating values or other signals:

```
repeat :: pred a ⇒ exp a → Sig exp pred a

map :: (pred a, pred b) ⇒ (exp a → exp b) →
  Sig exp pred a → Sig exp pred b

zipWith :: (pred a, pred b, pred c) ⇒ (exp a → exp b → exp c) →
  Sig exp pred a → Sig exp pred b → Sig exp pred c
```

The above signal functions model the similarly named functions in Haskell's base library: `repeat` creates a signal by repeating some value, `map` applies a function to each value of a signal, and `zipWith` joins two signals element-wise using the given function. The idea is to mimic the kind of compositional programming that users normally do in Haskell. Ground types in the expression language are lifted to operate element-wise over signals as well, typically with a type class like `Num`:

```
instance (Num (exp a), pred a) ⇒ Num (Sig exp pred a) where
  fromInteger = repeat . fromInteger
  (+)         = zipWith (+)
  (-)         = zipWith (-)
  ...
```

All signal functions shown so far have been combinatorial in nature, in the sense that their output only depends on the current inputs. Sequential functions on the other hand needs to access older values. For these, the signal language provides a unit delay:

```
delay :: pred a ⇒ exp a → Sig exp pred a → Sig exp pred a
```

`delay` prepends a value to a signal, delaying its original output by one time instant—the function introduces the notion of a *next time step*, making time enumerable. While `delay` may appear innocent, when combined with feedback it can describe any kind of sequential signal network. For example, a parity checker can be defined as:

```
parity :: Sig exp pred Bool → Sig exp pred Bool
parity inp = out where out = zipWith xor (delay false out) input
```

As a larger example, consider an infinite impulse response (IIR) filter, which comprises the second primary type of digital filters used in digital signal processing applications and contains feedback. The IIR filter is typically described and implemented in terms of a difference equation:

$$y_n = \frac{1}{a_0} \cdot \left(\sum_{i=0}^{P} b_i \cdot x_{n-i} - \sum_{j=1}^{Q} a_j \cdot y_{n-j} \right) \tag{3.1}$$

where P and Q are the feed-forward and feedback filter orders, and a_j and b_i are the filter

coefficients. Note that a_0 is used in the outer division and is not part of the feedback sum.

Examining the above equation, the IIR filter can be identified to loosely consist of two FIR filters, where the second filter has an extra delay and is recursively defined on its output:

```
1 iir :: (Fractional a, Num (exp a), pred a) ⇒ exp a → [exp a] → [exp a] →
2   Sig exp pred a → Sig exp pred a
3 iir a0 as bs x = y
4   where
5     y = (1 / repeat a0) * (upper x - lower y)
6     upper = fir bs
7     lower = fir as . delay 0
```

`fir` is the FIR filter from section 2.3.

A circular definition of `y` in the IIR filter is possible thanks to the `delay` operator, which ensures a productive network as each output only depends on previous input. In general, recursively defined signals introduce feedback, while recursion over Haskell values, like lists, can be used to build repeating graphs structures.

This behavior of `delay` implies that functions can distinguish values that have and haven't been delayed, which is something that is normally not possible to do in Haskell—being able to observe the sharing of `y` in `iir` will, by definition, break any referential transparency. In fact, the internals of `delay` makes use of a restricted form of observable sharing [22, 30]. This allows the signal compiler to turn signal functions into a directed graph, where sharing is visible as edges in the graph, connecting the nodes of its operations.

A graph representation of a signal network enables the signal compiler to check for cycles, order its nodes, and translate it into an imperative program. Directly mapping a signal network into a program in the co-design language is however a poor choice, as programs have no notion of the streaming that its signal function originally had.

Co-iterative streams [21] makes for a better compilation target, as they allow for infinite data types to be handled in a strict and efficient way. Co-iterative streams consists of an initial state and a transition function from its current state to an output and a new state. The benefit of this approach is that it :

```
1 data Stream instr exp pred a where
2   Stream :: Program instr (P2 exp pred) (Program instr (P2 exp pred) a)
3     → Stream instr exp pred a
```

where the outer program initializes the stream, which results in another program that produces the stream's output values and updates the state.

3.6 Related Work

Sheeran pioneered the use of functional languages in hardware design with μFP [52], a language that uses functional combinators to describe complex hardware from a set of smaller circuits and gates. The Lava family [15, 31, 41] of functional languages have since expanded upon the ideas of muFP and introduced modern functional features. For instance, Lava exploits monads and type classes to provide multiple interpretations of

circuit descriptions, such as simulation, formal verification and generation of netlists, and they use polymorphism and higher order functions to provide general descriptions of hardware designs. No Lava has yet ventured into the design of heterogeneous systems.

Outside of the Lava family, there is Bluespec [43], a hardware description language that is influenced by functional languages and includes, for instance, higher-order functions and polymorphism. In contrast to Lava, Bluespec can describe both software and hardware. Nevertheless, Bluespec descriptions are written at a clock-cycle granularity and therefore provide a lower level of abstraction than most functional languages. A third example is Chisel [11], a hardware description language embedded in Scala, which, like Bluespec, supports both cycle-accurate software simulation and hardware generation from its descriptions.

Compiling an ordinary C program to a hardware description has great appeal, but finding a translation between the two has however proven to be difficult. Tools like Catapult C [32] are able to generate register transfer level code from ordinary C descriptions, but its sequential programs are often a bad fit for the parallelism inherent to most hardware architectures. Additional C based attempts includes the creation of SystemC [29], a set of classes and macros which provide an even-driven simulation interface in C++. Although strictly a C++ library, SystemC is often viewed as a language of its own that simply reuses C++ syntax. Semantically, it has quite a few similarities to VHDL and Verilog.

Cryptol [18] is a DSL for the specification of cryptographic algorithms, and can generate both C and VHDL/Verilog from the same description. While not an embedded language, Cryptol has similar ambitions to the co-design language, in particular the ability to do rapid development and design exploration—although the latest versions of Cryptol no longer support hardware generation. However, since Cryptol is a stand alone language, any extension to it cannot benefit from the ecosystem of tools available in the host language. Another language outside the domain of HLS is Microsoft's Accelerator [58], for programming GPUs and various other platforms. Accelerators provides a high-level data-parallel programming model as a library that is available from a conventional imperative programming language like C. However, the project seems, unfortunately, to be no longer active.

There exists several methods that aid in the embedding of languages in Haskell. Among these different approaches is finally tagless [19], which associates each group of language constructs with a type class, and each interpretation with a semantic domain type. The result is an expressive and compositional way of embedding languages. This does however come at a cost of awkward types—the type of an embedded language is simply a qualified type variable—and it tends to expose implementation details to users. For Scala, the Delite [53] library provides a framework of reusable components for embedded languages, like optimizations, and code generators. Delite produces an intermediate representations of user programs that is similar to the model used by the co-design language, but targets a combination of CPU and GPU systems.

Lightweight Modular Staging (LMS) has also been explored as an option to ease the construction of a domain-specific High Level Synthesis (HLS) system [28]. The argument is that LMS eases the reuse of modules between different HLS flows, and makes it easier to link to existing tools, such as the C compilers that are able to produce register transfer level descriptions. Though the language-specific challenges for LMS are different from

those faced by the co-design language, the two approaches are comparable in terms of capability. The way in which code generation of programs is built upon the translation of monads to imperative programs is also reminiscent of Sunroof [17], a DSL for generating JavaScript.

The extensible types used in the co-design language are built upon Syntactic [4] and Data Types à La Carte [57]. Syntactic provides a generic model of syntax trees that is extensible and supports generic traversals. Its model is partly derived from Data Types à La Carte, which defines a composition operator for symbol domains and an interface for symbol subsumption. These techniques can collaborate, and together provide an extensible syntax tree for expressions in the co-design language, as well as the glue for composing instructions.

Among the DSLs embedded in Haskell, Feldspar [7, 8] has perhaps been the biggest source of inspiration for the co-design language. In fact, the co-design language could be considered the spiritual successor of its newer branch called Resource-Aware Feldspar [6]. Feldspar has many aspects in common with the co-design language, and is designed for programming digital signal processing algorithms, although it does so using vectors rather than signals or streams. Also, the Feldspar compiler is based on similar techniques as those used in the co-design language [5, 9]. Unlike the co-design language however, Feldspar only targets embedded software elements.

Obsidian [56] is an embedded language in Haskell for GPU programming that supports a style of vector programming that is in many ways similar to that of the co-design language. There are however a few differences, as Obsidian is build to specifically target GPUs. For instance, loops are automatically unrolled and it provides a control over the location of data in memory. Which is something that the co-design language currently does not support.

The signal processing language is based on the synchronous data-flow paradigm and is inspired by similar languages from this domain. Of the synchronous languages, Lucid Synchrone [49, 24] is perhaps the biggest source of inspiration and is designed to be used with reactive systems. Initially, the language was introduced as an extension of LUSTRE [33], and extended the language with new and powerful features. For instance, automatic clock and type inference were introduced and a limited form of higher-order functions was added. Lucid Synchrone is however a standalone language, and thus cannot be easily integrated with the co-design language. Zélus [16], a successor of Lucid Synchrone, has shown that synchronous languages can be extended to model hybrid systems as well, that is, system that consist of both continuous and discrete components.

Another, and rather different approach to modeling signal processing is functional reactive programming. Yampa [25] is one member of this paradigm, and is used for programming hybrid systems. At its core, functional reactive programming is about describing a system's behaviors and events, where behaviors are continuous and time-varying, reactive values, and events are time-ordered sequences of discrete-time event occurrences [44]. Apart from the different notions of time in synchronous data-flow and functional reactive programming, the idea behind behaviors are quite similar to the signals used in synchronous languages. An event is usually a separate entity, modeling the control flow of a system. Some languages merge the two concepts at a cost of some elegance by having discrete behaviors, but in return they can describe events in terms of behaviors.

3.7 Discussion

Heterogeneous computing represents an interesting development in the domain of embedded systems, and refers to systems making use of more than one kind of processor or accelerator. This comes at a cost of increased programming complexity, as the level of heterogeneity in a system can introduce non-uniformity in its system development and overall capabilities. Embedded systems are predominantly developed using low-level, imperative languages and, in the case of most heterogeneous systems, hardware description languages. While low-level languages are good for obtaining maximum performance of system, they provide little or no abstractions or modularity.

The co-design language presented in this thesis are the first steps towards a functional language which can describe the entire design process of a heterogeneous system. Both the co-design language and its compiler are works in progress, aiming to provide modular, generic and portable description of embedded systems, with a reasonable degree of control over the generated code—programs are designed to have predictable performance and memory usage. That is, the co-design language is based on a design where memory usage is explicitly managed by the user, leading to a lower and easier to predict memory usage than that of a typical lazy language.

Being embedded in Haskell, the co-design language exploits its host's parametric polymorphism and type classes to support generic program descriptions, facilitating the exploration of good boundary between hardware and software. That is, the co-design library provides a type for software programs, hardware programs, and a kind of generic programs that are constrained by the operations it requires rather than the entire software or hardware languages. Once a satisfactory hardware software partitioning has been found, these generic programs can be turned into language specific programs by simply changing their type, after which the program can be tailored to its target if necessary. The boundary between software and hardware is itself fully implemented in the co-design language, and provided as a function that, given an encoding of a program's type signature, generates a custom AXI4-lite interconnect.

The co-design language is also designed to be modular in not only the user's perspective, but also a language implementers perspective. That is, the language's use of a mixture of shallow and deep embeddings makes it easy to add new features and combinators, as well as instructions, expressions, and interpretations. The vector and signal processing extensions are an example of the former, while section 3.1 gave an example of the latter. Extensions of instructions and expressions are safe in the sense that they can be defined separately from the type they extend, and any interpretation the original types may have supported can be regained for the extended type by adding support for its new extension.

3.8 Future Work

While the co-design language is designed with heterogeneous systems in mind, so far it has only been tested on the Parallella system [46], which consists of an FPGA with two embedded ARM cores and a many-core accelerator. Of the four processing elements on the Parallella, only the FPGA and a single ARM core is so far used. There has been

earlier attempts at describing the many-core accelerator as well, but they have yet been included in the latest version of the co-design language. The accelerator is programmed in C, but the compiler must then also consider the location of sub programs on the different small processors. Furthermore, an implementation of the full AXI4 interconnect should be added to support burst writing of arrays, rather than the single value transmissions offered by the current AXI4-lite implementation.

The process of compiling a hardware program to VHDL, synthesizing a bitstream, and put that onto the FPGA's programmable logic is entirely manual, as shown in Appendix A. Most synthesizers do however support a scripting language that could be used to automate the process. Also, the physical address of a hardware component has to be manually incorporated into a software program and memory-mapped. These steps should ideally be hidden and automated, where the hardware compiler automatically feeds the software compiler with addresses it got from the synthesizer.

A considerable part of future work for the co-design language are its extensions. The vector and signal processing languages do aid in the design of some systems, but there is still work to be done for fitting programs to many-core devices. For example, only one of the Parallella's embedded ARM cores are currently used, and a language of combinators for programming many-core accelerators is still missing. Also, the co-design language could benefit from incorporating a verification back-end to validate its generated designs, and effort has already been put into using a SMT solver to verify assertions produced by the co-design language's compiler. Another interesting extension of the co-design language is the introduction of a new signal processing framework like Ziria [38], but embedded in Haskell.

3.9 Conclusion

The co-design project originally set out to see how far Haskell and embedded languages would go towards the design of a heterogeneous embedded system, where at least two sub languages were required to describe the system's various processing elements. So far, this seemingly unique approach to co-design has proved to be a useful one.

A hardware software co-design language has been defined, capable of generating C for software elements, VHDL for hardware elements, and a mixture of the two for orchestrating the communication between software and hardware elements. The co-design language is also able to describe generic programs, a feature that is quite useful as it facilitates design exploration.

The vector and signal processing extensions introduced a compositional style for computations in their respective domains. Both extensions enabled, for instance, a FIR filter to be implemented using only a single line of code while precisely capturing its mathematical specification. Their high-level combinators is welcome reprieve form the typically low-level programming of embedded systems, while features such as fusion ensures performance does not suffer. While both extensions were intended to be used with the co-design language, they abstract over their underlying expression types and can be used with other languages that provide pure expressions.

A common model of imperative programs for both the software and hardware languages

proved to be quite useful, as it enabled many of its interesting interpretation techniques to be reused for the two languages. Furthermore, as the model was designed to be loosely coupled with the program's associated types, the model is useful for language development outside of hardware software co-design. The current approach to compilation as a typed translation between progressively smaller languages has helped safeguard against most of the issues typically associated with untyped compilation schemes.

Appendices

A Synthesizing in Vivado

This section explains the processes of manually importing and synthesizing a design in the 2015.2 version of Vivado [27]. As this processes is the same for all designs, the explanation will focus on the required steps in a general sense rather than showcase a specific example.

Having generated code for a hardware program wrapped in the AXI4-lite interconnect, the first step is loading the code into a Vivado project. In the case of AXI4 projects, Vivado provides a wizard to create an AXI4 template under the "Create and Package IP" action under "Tools" in the project's menu—the base project is typically supplied by the system's manufacturer, otherwise a basic one can be created by Vivado.

The AXI4 wizard will ask for a name, organization, and a few other details that are largely irrelevant to the template it generates, only the name is important as it will identify the component once it has been packaged. Afterward, the wizard will ask for a few settings, such as the number and size of its slave registers, these are again irrelevant and will be replaced later on. Once the wizard is finished the template can be added to the project's IP catalog and opened in an editing session, which should contain two files as shown in Figure A.1.

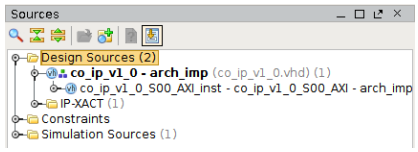

Figure A.1

The first file in Figure A.1 simply contains a shell entity that instantiates a few constants an calls the empty AXI4-lite slave, which is located in the second file. Its this empty slave that should be substituted by the generated code. Note that, after updating the second design file the references from the first design file needs to be updated as well, since the slave's name most likely changed. Once that has been taken care of, the project can be reviewed and packaged—the review will probably complain that an address port has changed in size, as it always use the full 32 bits rather than a subset like Vivado. Go ahead and update its size accordingly.

Once the AXI4 peripheral has been updated, saved, and packaged, the main project can incorporate it as an "IP Core" in a few steps. Firstly, under the "IP Settings" menu, there is an option to locate the peripheral's project as a repository and to add its contents to the available IP cores. The component can the be added from either the context menu or from simply right-clicking the block design window and picking the option "Add IP". Vivado is able to connect the peripheral automatically to the main project by running its "Run Connection Automation" tool—Vivado will typically prompt for this option once an IP has been added. Figure A.2 shows the block design view of an example for a Parallell project that has been stripped to the bare minimum of components (a Zynq with a few components for managing AXI4 peripherals), where the AXI4-lite slave is the smaller box in the lower right of the view. Note that it is possible to get the physical address of the slave from the "Address Editor" at this point.

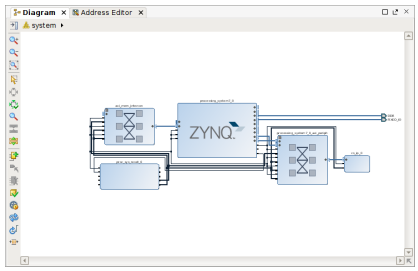

Figure A.2

Finally, the last step is to generate a bitstream that flashed onto the FPGA's programmable logic. The project's context menu provides an action for generating the bitstream, and with the example project from Figure A.2, produces an implementation like the one in Figure A.3.

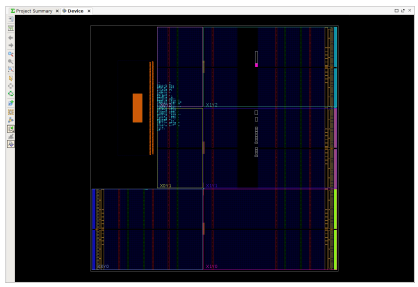

Figure A.3: *Implementation of the project.*

References

[1] Heinrich Apfelmus. The Operational Monad Tutorial. *The Monad. Reader* **15** (2010), 37–55 (cit. on p. 15).

[2] Markus Aronsson, Emil Axelsson, and Mary Sheeran. "Stream Processing for Embedded Domain Specific Languages". *Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages*. ACM. 2014, p. 8 (cit. on pp. v, 39).

[3] Markus Aronsson and Mary Sheeran. "Hardware Software Co-Design in Haskell". *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. ACM. 2017, pp. 162–173 (cit. on pp. v, 53).

[4] Emil Axelsson. "A Generic Abstract Syntax Model for Embedded Languages". *ACM SIGPLAN Notices*. Vol. 47. 9. ACM. 2012, pp. 323–334 (cit. on p. 25).

[5] Emil Axelsson. Compilation as a Typed EDSL-to-EDSL Transformation. *arXiv preprint arXiv:1603.08865* (2016) (cit. on p. 25).

[6] Emil Axelsson et al. *Resource-Aware Feldspar*. Aug. 30, 2016. URL: `hackage.haskell.org/package/raw-feldspar` (cit. on p. 25).

[7] Emil Axelsson, Anders Persson, Mary Sheeran, et al. "Feldspar: A Domain Specific Language for Digital Signal Processing Algorithms". *Formal Methods and Models for Codesign, 2010 8th IEEE/ACM International Conference on*. IEEE. 2010, pp. 169–178 (cit. on p. 25).

[8] Emil Axelsson, Anders Persson, Mary Sheeran, et al. "The Design and Implementation of Feldspar". *Symposium on Implementation and Application of Functional Languages*. Springer. 2010, pp. 121–136 (cit. on p. 25).

[9] Emil Axelsson and Andrea Vezzosi. "Lightweight Higher-Order Rewriting in Haskell". *International Symposium on Trends in Functional Programming*. Springer. 2015, pp. 1–21 (cit. on p. 25).

[10] Christiaan Baaij et al. "ClaSH: Structural Descriptions of Synchronous Hardware using Haskell". *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. IEEE. 2010, pp. 714–721 (cit. on p. 3).

[11] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 1216–1225 (cit. on p. 24).

[12] David F Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Communications of the ACM* **56**.4 (2013), 56–63 (cit. on p. 5).

[13] Michael Barr and Anthony Massa. *Programming Embedded Systems: with C and GNU Development Tools*. O'Reilly Media, Inc., 2006.

[14] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. "Clock-directed Modular Code Generation of Synchronous Data-Flow Languages". *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. June 2008.

[15] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. "Lava: Hardware Design in Haskell". *ACM SIGPLAN Notices*. Vol. 34. 1. ACM. 1998, pp. 174–184 (cit. on pp. 3, 7, 23).

[16] Timothy Bourke and Marc Pouzet. “Zélus: A Synchronous Language with ODEs”. *16th International Conference on Hybrid Systems: Computation and Control*. Mar. 2013, pp. 113–118. URL: http://www.di.ens.fr/~pouzet/bib/hscc13.pdf (cit. on p. 25).

[17] Jan Bracker and Andy Gill. “Sunroof: A Monadic DSL for Generating JavaScript”. *Proc. 16th Int. Symp. on Practical Aspects of Declarative Languages*. Springer International Publishing, 2014, pp. 65–80. ISBN: 978-3-319-04132-2 (cit. on p. 25).

[18] Sally Browning and Philip Weaver. “Designing Tunable, Verifiable Cryptographic Hardware using Cryptol”. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 89–143 (cit. on p. 24).

[19] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming* **19**.5 (2009), 509–543 (cit. on p. 24).

[20] Magnus Carlsson and Thomas Hallgren. “Fudgets: A graphical User Interface in a Lazy Functional Language”. *Proceedings of the conference on Functional programming languages and computer architecture*. ACM. 1993, pp. 321–330.

[21] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. *Electronic Notes in Theoretical Computer Science* **11**.0 (1998), 1–21. ISSN: 1571-0661. URL: http://www.sciencedirect.com/science/article/pii/S1571066104000507 (cit. on p. 23).

[22] Koen Claessen and David Sands. “Observable Sharing for Functional Circuit Description”. *Annual Asian Computing Science Conference*. Springer. 1999, pp. 62–73 (cit. on p. 23).

[23] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. “Expressive Array Constructs in an Embedded GPU Kernel Programming Language”. *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*. ACM. 2012, pp. 21–30 (cit. on pp. 20, 21).

[24] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. “Towards a Higher-Order Synchronous Data-Flow Language”. *Proceedings of the 4th ACM International Conference on Embedded Software*. EMSOFT ’04. Pisa, Italy: ACM, 2004, pp. 230–239. ISBN: 1-58113-860-1. URL: http://doi.acm.org/10.1145/1017753.1017792 (cit. on p. 25).

[25] Antony Courtney, Henrik Nilsson, and John Peterson. “The Yampa Arcade”. *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. Haskell ’03. Uppsala, Sweden: ACM, 2003, pp. 7–18. ISBN: 1-58113-758-3. URL: http://doi.acm.org/10.1145/871895.871897 (cit. on p. 25).

[26] Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling Embedded Languages. *Journal of functional programming* **13**.3 (2003), 455–481 (cit. on pp. 7, 8, 20).

[27] Tom Feist. Vivado Design Suite. *White Paper* **5** (2012) (cit. on p. 30).

[28] N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne. “Making Domain-Specific Hardware Synthesis Tools Cost-Efficient”. *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 120–127 (cit. on p. 24).

[29] Frank Ghenassia et al. *Transaction-level modeling with SystemC*. Springer, 2005 (cit. on p. 24).

[30] Andy Gill. "Type-Safe Observable Sharing in Haskell". *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. ACM. 2009, pp. 117–128 (cit. on p. 23).
[31] Andy Gill et al. "Introducing Kansas Lava". *Implementation and Application of Functional Languages*. Springer, 2010, pp. 18–35 (cit. on pp. 3, 23).
[32] Mentor Graphics. Catapult C synthesis. *Website: http://www. mentor. com* (2008) (cit. on p. 24).
[33] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. International Series in Engineering and Computer Science 215. Springer, 1993 (cit. on p. 25).
[34] John Hughes. "The design of a Pretty-Printing Library". *International School on Advanced Functional Programming*. Springer. 1995, pp. 53–96 (cit. on p. 7).
[35] John Hughes. Why Functional Programming Matters. *The computer journal* **32**.2 (1989), 98–107 (cit. on p. 5).
[36] David M Kunzman and Laxmikant V Kale. "Programming heterogeneous systems". *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE. 2011, pp. 2061–2064.
[37] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World (2002) (cit. on p. 7).
[38] Geoffrey Mainland, Dimitrios Vytiniotis, et al. Ziria: A DSL for Wireless Systems Programming. *ACM SIGPLAN Notices* **50**.4 (2015), 415–428 (cit. on p. 27).
[39] R McMillan. Microsoft supercharges Bing Search with Programmable Chips. *Wired Business* **16** (2014) (cit. on p. 5).
[40] Trevor Mudge. Power: A first-class architectural design constraint. *Computer* **34**.4 (2001), 52–58.
[41] Matthew Naylor. *York Lava*. Sept. 15, 2009. URL: `hackage.haskell.org/package/york-lava` (cit. on p. 23).
[42] Cisco Visual Networking Index. Forecast and methodology, 2016-2021, white paper. *San Jose, CA, USA* (2016).
[43] Rishiyur Nikhil. "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications". *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*. IEEE. 2004, pp. 69–70 (cit. on p. 24).
[44] Henrik Nilsson, Antony Courtney, and John Peterson. "Functional Reactive Programming, Continued". *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania: ACM, 2002, pp. 51–64. ISBN: 1-58113-605-6. URL: `http://doi.acm.org/10.1145/581690.581695` (cit. on p. 25).
[45] W Obile. *Ericsson Mobility Report*. 2016.
[46] Andreas Olofsson, Tomas Nordström, and Zain Ul-Abdin. Kickstarting High-Performance Energy-Efficient Manycore Architectures with Epiphany. *arXiv preprint arXiv:1412.5538* (2014) (cit. on p. 26).
[47] Alan V Oppenheim, Ronald W Schafer, John R Buck, et al. *Discrete-Time Signal Processing*. Vol. 2. Prentice-hall Englewood Cliffs, 1989 (cit. on p. 9).
[48] Anders Persson. Towards a functional programming language for baseband signal processing (2014).

[49] Marc Pouzet. Lucid Synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI* (2006). URL: `http://www.di.ens.fr/~pouzet/lucid-synchrone/manual_html/` (cit. on p. 25).

[50] Mary Sheeran. Hardware Design and Functional Programming: a Perfect Match. *J. UCS* **11**.7 (2005), 1135–1158 (cit. on p. 5).

[51] Mary Sheeran. "Hardware Design and Functional Programming: Still Interesting after All These Years". Presented the 20th International Conference on Functional Programming, 2015.

[52] Mary Sheeran. "muFP, a language for VLSI design". *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM. 1984, pp. 104–112 (cit. on p. 23).

[53] Arvind K Sujeeth et al. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Transactions on Embedded Computing Systems* **13**.4s (2014), 134 (cit. on p. 24).

[54] Josef David Svenningsson and Bo Joel Svensson. Simple and Compositional Reification of Monadic Embedded Languages. *Proc. 18th Int. Conf. on Functional Programming (ICFP)* (2013) (cit. on p. 15).

[55] Josef Svenningsson and Emil Axelsson. "Combining Deep and Shallow Embedding for EDSL". *International Symposium on Trends in Functional Programming*. Springer. 2012, pp. 21–36 (cit. on p. 8).

[56] Joel Svensson, Mary Sheeran, and Koen Claessen. "Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors". *Symposium on Implementation and Application of Functional Languages*. Springer. 2008, pp. 156–173 (cit. on p. 25).

[57] Wouter Swierstra. Data Types à La Carte. *J. Funct. Program.* **18**.4 (July 2008) (cit. on pp. 15, 25).

[58] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Simplified Programming of Graphics Processing Units for General-Purpose Uses via Data-Parallelism. *Rapport Technique, Microsoft Research* (2005) (cit. on p. 24).