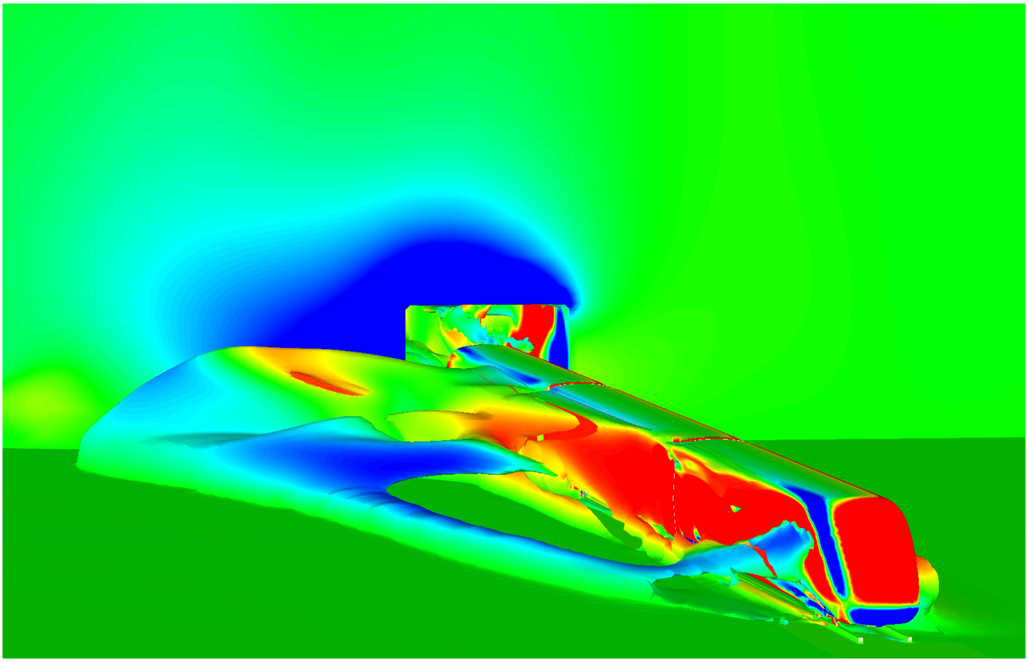
CHALMERS



Title

MARKUS ARONSSON

Department of Stuff
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2011

THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING IN AREA

Title

MARKUS ARONSSON

Department of Stuff
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2011

Title
MARKUS ARONSSON

Thesis for the degree of Licentiate of Engineering 2011:01
ISSN 1652-8565
Department of Stuff
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: +46 (0)31-772 1000

make sure you have
an uptodate
template. They keep
changing.

Cover:
Some explanation

Chalmers Reproservice
Göteborg, Sweden 2011

Title
MARKUS ARONSSON
Department of Stuff
Chalmers University of Technology

to what or who?

refers

shorten & make more specific

too waffly

Abstract

Developing software for embedded systems, in most cases, presents quite the challenge. Not only do they require good knowledge of the targeted hardware, but they are also designed with a very specific task in mind. That is, an embedded system is typically build with the most cost-effective hardware that meets its performance requirements. Developers must therefore ensure they take full advantage of the custom hardware while keeping their computations efficient. Furthermore, designers must often consider several other, non-functional properties, including: software partitioning, a computation should run on the component that best suits it; and memory layout, since resource constraints can be just as limiting as performance.

t

Modern embedded systems often gain their performance from combining several kinds of co-processors, where each processor provides specialized processing capabilities to handle a particular task. Their efficiency does however come at a cost of increased complexity: co-processor specific code is interleaved with regular software code and hardware for the communication between co-processors. As a result, and considering the aforementioned design decisions, embedded systems can be quite difficult to program. Especially so considering that most embedded systems are programmed in low-level languages, like C or even assembler, which severely limits the portability of code.

In this thesis we explore a functional approach to embedded systems development through or co-design language and its associated vector and signal processing languages. The functional approach allows for computations to be passed around as first-class objects and assembled in a modular wary into a larger applications. Also, thanks to the powerful type-systems of functional languages, these computations depend on specific operations rather than an entire language, which alleviates design exploration.

way

/a

enables ?

A functional approach to embedded systems development does address many of the modularity problems with using low-level languages, those languages are mainly used for their fine control over a system's capabilities. The languages we present in this thesis are therefore staged in order to give the designer explicit control over a large portion of the generated code.

sentence, sentence

only one

explain more

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 ~~has~~ 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 26nd 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 section s 3.1 and 3.2. The paper was rewarded 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.

awarded

Contents

Part I
Extended Summary

What is this?

(error of number)

1 Introduction

An

is

Embedded systems are, in brief, any computer system that is part of a larger system but relies on its own microprocessor. It is embedded as part of a larger machine to solve a particular task, and often does so under memory and real-time constraints. Embedded systems have been have been around for decades, and still control many devices in common use today [7]. For example, modern cars contain a number of embedded systems, each controlling a small function of the car; one system might control the brakes, while another displays information to the dashboard. In most cases, these embedded sub-systems are all connected together in a network.

Developing software for embedded systems, in most cases, requires good knowledge about the hardware on which the software is supposed to run. This is because embedded systems are typically designed with a task in mind and consists of the most cost-effective hardware that meets the performance requirements. Developers must therefore ensure that not only are their computations efficient, but also take full advantage of the custom hardware; every line of code counts.

#

As important as computational power is for embedded systems, limiting their power consumption presents an equally important issue in their architectural design [32]—the trend of trading power for performance cannot continue indefinitely. Containing the growth in power requires architectural improvements, with specialized computing for specialized tasks.

Heterogeneous computing represents an interesting development towards the goal of energy efficient computing, and refers to systems making use of more than one kind of processor. These heterogeneous systems gain their performance and energy efficiency not just by combining several processors, but rather by incorporating different kinds of co-processors that provide specialized processing capabilities to handle a particular task. Their efficiency and computational power comes at a cost of increased programming burden in terms of code complexity and portability. Hardware specific code is interleaved with software code to describe its various components and to handle any communication between co-processors.

A substantial amount of research has gone into addressing the challenges of programming for embedded heterogeneous systems, opening them up for programmers without a background in hardware or embedded system design. Hardware description languages are however still the most commonly used tools, together with C dialects for specific co-processors. While such low-level languages are good for extracting maximum performance from a processor, they provide little to no abstractions for alleviate the task of programming for heterogeneous systems.

not sure of this use

that

Another group of languages whom I believe show great promise in describing hardware designs are the functional languages. Higher-order functional languages in particular, where hardware descriptions are first-class objects, offer a particularly ~~useful~~ abstraction mechanism [4, 9, 24]. For instance, a higher-order functional language allows for computations to be passed around as first-class objects. Smaller functions can then be assembled in a modular way into a larger applications. Also, thanks to the powerful type-systems of functional languages, it is possible to precisely record a function's dependencies in its type; functions can depend on the smaller functions they use rather the entire domain

too many particular

too vague

than

not sure what this sentence means

delete?

adoption

they're part of.

Despite the aforementioned benefits of functional programming languages, they are rarely considered for embedded system development. One reason for their low adaptation is that some features, while facilitating the design of hardware ~~descriptions,~~ also makes it difficult to give performance guarantees and resource bounds, especially so for functional languages with lazy evaluation.

description of hardware?

This thesis presents the first steps towards a functional programming language for embedded heterogeneous systems. Instead of taking on the full challenge of heterogeneous programming head on, we are currently exploring a more modest approach: develop a hardware software co-design language, embed it in Haskell, and see how far we can get. The language is staged and utilizes the rich type system of its host language to facilitate design exploration. Furthermore, we have developed a vector and a signal processing language to accompany the co-design language.

As an example of our 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 the products of the corresponding entries of the two vectors:

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

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

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

While the above function is faithful to its corresponding implementation in C, the low-level design has forced us to focus on implementation details rather than the mathematical specification of the dot product. In this situation, where the function can be succinctly expressed as a sum over products, we would be better off using the vector language instead. In fact, the same dot product can be implemented with vectors in a single line:

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

At this point we should note that neither version of the dot functions so far mentions any software or hardware specifics and, as such, the functions could be realized in both languages. Lets assume we wish to put `dotVec` onto hardware. Before we can offload the function, we must first give it a signature that describes its inputs and output. For the dot-product, this means that we declare two input arrays with and a single output:

?

long winded

(delete)

```
dotSig :: Signature (Arr Int32 → Arr Int32 → Sig Int32)
dotSig = inputVec 4 $ λx → inputVec 4 $ λy → returnVal $ dotVec x y
```

The signature of a component lets us inspect a type from within Haskell and, for example, hook it up to an AXI4-lite interconnect which lets us reach the component from software through memory-mapped I/O. This process causes the component to share its address space with the memory of whatever software program they are running in, that is, the component can be reached by simply reading and writing to pointers into its address. The co-design language provides a `compileAXI4lite` function that produces a synthesizable interconnect for a given function's signature.

hard to follow Add a diagram

After the dot-product has been hooked up to an AXI4-lite interconnect, synthesized and put onto hardware, we can write a small software program that interacts with the component and prints its result:

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

Firstly, the software program brings our component into scope by calling the memory-mapping function, `mmap`, with the physical address and signature of the component. Then we then declare two arrays and a reference to hold the output and input values of the component. Finally, we call the component using `call` with our arrays and reference as arguments. The result is then read from the reference and printed to standard output.

2 Background

High demands for efficiency under resource constraints have greatly influenced the development of embedded systems, both in terms of programming practice and architecture. Today, we see embedded systems consisting of everything between general purpose processors (GPPs) and application specific integrated circuits (ASICs).

GPPs and ASICs represent two extremes of available architectures, where field programmable gate arrays (FPGAs) have found a good middle-ground and provide the best of both worlds: they are close to hardware and can be reprogrammed [6]. Modern FPGAs also contain various discrete components and co-processors which, together with their good performance per Watt ratio, have seen them increasingly used in high-performance, computationally intensive systems [31].

A modern FPGA show great promise as a prototypical system for heterogeneous computing, but their adoption has been slowed by the fact that they're difficult to program. The logic blocks of an FPGA are usually programmed in a hardware description language, while the embedded processors are programmed in a low level dialect of C or

#

even assembler. Furthermore, the various components of a modern FPGA may support different intrinsics and therefore have incompatibilities between the code they can execute even if they are programmed in the same language.

One of the benefits of using low level languages for embedded systems is that they give a designer fine control over a system capabilities. However, this control comes at a cost, as the programmers must exercise this right during the entire design process. So the problem of implementing an algorithm has become a problem of implementing an algorithm for a specific component.

The issues with low level languages are magnified for heterogeneous systems, as the developer must specify both its hardware and software parts and how they communicate; ideally she would like to experiment with various choices of what to put in hardware and what in software. Low-level languages provide little support for such design exploration, and rewriting code intended for one component to another is typically a major undertaking.

Many of the issues faced in heterogeneous computing with low-level languages stem from a lack of abstractions. Some of these languages' modularity problems come as a direct result of the fine grained control they provide, as it will inevitably tie programs to the architecture of its system. Other issues of, for instance, functionality and architecture come as a indirect consequence of the lack of abstractions. Ideally, such issues would be treated separately, as they allow for the creation of small, reusable libraries that provide solutions to these issues, and then combined in a modular fashion to solve larger problems.

In the paper "Why functional programming matters" [28], Hughes argues that many of the above modularity problems can be address by making use of functional programming. Particularly the glue code that functional programming languages offer, through higher-order functions and lazy evaluation, enables us to build useful combinators.

The benefits of a functional programming language is however not limited to software development, as Sheeran shows in her paper "Hardware Design and Functional Programming: a Perfect Match" [40]. Where she exemplifies how a functional language can make it easy to explore and analyze hardware designs in a way that traditional hardware description languages would have found difficult, if not impossible.

one sentence

Before we go into combining these benefits for heterogeneous computing, an introduction to functional programming and embedded languages is in order.

nice so far!!

2.1 Functional Programming

Functional programming, as the name implies, is based around the application of a function to its arguments. In this programming style, a program is written as a function that accept input and deliver its result. This 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 functions in a functional programming language and, say, an imperative language like C, is that functions always return the same value when given same arguments. More generally, we say that functional programs have no side effects; functions can safely be evaluated in parallel as long at their data dependencies are satisfied.

the

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 programmers build complex functions from smaller ones. In Haskell, a functional programming language, a number of such higher-order functions that implement common operations are provided by its standard libraries. One such function is `map` and can be defined as follows:

```
map :: (a → b) → [a] → [b]
map f []     = []
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 you by mistake write a program that tries to multiply some integer by a boolean type, it won't even compile and instead warns you about the type error. As for the types themselves, they are a kind of label that every expression has and states what category of operations that the expression belongs in.

A function's type comes after the :: sign, and in the case of `map`, tells us that its first argument is a function `f :: a → b` which, given an argument of type `a`, produces a result of type `b`. In addition to the function, `map` also takes a list `xs :: [a]` of element with type `a`, and returns another list of elements with type `b`. As functions cannot have any side effects, we can already make a guess at what this function does and claim that it applies `f` to every element of `xs`.

The second and third line of `map` validates our earlier guess and lists the full definition of the function. Firstly, it says that given an empty list, shown as `[]`, the result is another empty list—there's simply nothing to apply `f` to. Secondly, in the case where `map` is given a non-empty list `x:xs` where `x` is the list head and `xs` its tail, it applies `f` to `x` and concatenates its result with the list made from a recursive call to itself on `xs`.

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 primitives values and functions for combining them. These combinators allow for complex structures to be built with a small set of verified combinators. For example, the earlier dot-product from section 1 is composed of two such combinators: `zipWith`, a generic way of joining two vectors, and `sum`.

Maybe mention impl. using fold ?

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:

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

where parenthesis around the dot implies that function composition is an infix function. While the size of the intermediate result of `f` could spoil the usefulness of composition, functional programming solves this by only evaluating `f` as much as is needed by `g`. This property is referred to as lazy evaluation.

explain above

The benefits of lazy evaluation is not limited to fusing functions and values, but extends to embedded languages as well. For instance, laziness in our co-design language ensures that only the parts of a program that contribute to the end result will be part of the function, that is, no unnecessary code will generated for a program.

So far, we have looked at a few Haskell functions as an introduction to functional programming and talked a bit about how its beneficial properties can help embedded languages. We have yet to make the distinction between regular and embedded Haskell. The following section introduces the concept of domain specific languages, and explains what explains what it entails to be an embedded domain specific language in Haskell.

2.2 Domain Specific Languages

A domain specific language (DSL) is a special-purpose language, tailored to a certain problem and captures the concepts and operations in its domain. For instance, a hardware designer might write in VHDL, while a web-designer that wants to create an interactive web-page would use JavaScript. Both use a language that is specialized to the particular task they have at hand, and both build programs in a form that is familiar to regular programmers; VHDL and JavaScript are both ~~examples of a~~ DSLs.

explain at first use

Haskell, with its static type system, flexible overloading and lazy semantics, has come to host a range of EDSLs [19]. For instance, popular libraries for parsing, pretty printing, hardware design and testing have all been embedded in Haskell [30, 27, 9]. These DSLs come in two fundamentally different forms: external and internal. An external DSL is a first-class language, with its own compiler or interpreter, and often comes with its own ecosystem. Internal DSLs are embedded in a host language, and are often referred to as embedded domain specific languages (EDSLs). The co-design, vector and signal languages presented in this thesis are all examples of internal DSLs.

unclear You seem to have narrowed down to EDSLs but then you talk about external langs. Reorder?

We illustrate how shallow embeddings work in Haskell through a small example:

first mention (so unclear to reader)

```
type Exp = Int

const :: Int → Exp
const a = a

times :: Exp → Exp → Exp
times a b = a * b
```

Exp is a short-hand for expressions and is defined as a type synonym for integers, making use of Haskell's type keyword. The language around our expressions is defined by two functions, one for integer literals, called const and another for multiplication, called times. Note we could easily have added more functions, as long as they're integral expressions.

Another advantage of Exp is that we can quickly calculate its value:

```
eval :: Exp → Int
eval a = a
```

Shallowly embedded types do however perform quite poorly if we wish to compile the language; functions only return values and provide no way to look at the representation

refer to that famous de Moor paper?

of a program. To compile an embedded language it is better to use an intermediate representation for functions, which sits between Haskell and the compiled code. This technique is known as a deep embedding and its functions return an abstract syntax tree that represents the computed value instead of its result.

We can reimplement our earlier `Exp` type using deep embedding:

```
data Exp = Const Int | Times Exp Exp
```

bar?

Expressions now contain two constructors, one for integer literals and another for multiplication of expressions—Haskell's `data` keyword introduces a new type and its different constructors are separated by a pipe. These constructors forms what is often referred to as a syntax tree, and represents the computations behind an expression. #

Having changed `Exp` we must also change the functions that comes with it: #

```
const :: Int → Exp
const a = Const a

times :: Exp → Exp → Exp
times a b = Times a b
```

`const` and `times` now returns a representation of the result rather than the result itself. As a consequence, we cannot add new functions without first extending the `Exp` type. From a user's perspective these functions are not that different from their shallow counterparts. In fact, embedded languages can have the look and feel of a stand-alone language. #

The syntax tree used for a deeply embedded type like `Exp`, while inflexible compared to its shallow version, is what enables functions to inspect, modify, and interpret an expression in order to support, for example, their evaluation: its

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

Each line of `eval` handles one of the two constructors in `Exp`, translating them into their corresponding Haskell value s. Evaluating an expression to its equivalent Haskell value is however not the only supported interpretation of expressions. We could just as well have compiled the same expression to, say, its corresponding C code.

While the implementation of shallow and deep embedding are usually at odds, there has been work done in order to combine their benefits [45]. In our co-design language, we make use of such a combination of deep and shallow embeddings: the core syntax is implemented as a deep embedding, with user facing libraries as shallow embeddings on top. This mixture of embeddings means that our core language is easy to interpret, while the user-facing libraries are able to provide ~~a~~ nice syntax for their functions.

unclear is it the impls that are at odds!

Now that we have a grasp of functional programming, what domain specific languages in Haskell are and the ideas behind them, the next section will go through a larger example in order to showcase embedded programming in the co-design language.

wildly?

2.3 Embedded Programming in Haskell

As we saw in the section s 2.1 and 2.2, programming in a functional language like Haskell is widely different from the imperative style used in a language like C. In Haskell, users write ~~their~~ a program~~s~~ as a mathematical function from inputs to output, while in C they write ~~them~~ it as a series of sequential steps to execute on some machine.

As an example of the above differences, we'll consider a finite impulse response (FIR) filter, one of the two primary types of digital filters used in digital signal processing applications [37]. 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" as they tap into the input signal at various time instants.

In an imperative language like C, we can implement the FIR filter as:

```
void fir(int N, int L, double *b, double *x, double *y) {
 int j, k;
 double tap[256];
 for(j=0; j<N; j++) tap[j] = 0.0;
 for(j=0; j<L; j++) {
  for(k=N; k>1; k--) tap[k-1] = tap[k-2];
  tap[0] = x[j];
  y[j] = 0.0;
  for(k=0; k<N; k++) y[j] += b[k] * tap[k];
 }
}
```

Here, N is the filter rank, as before, and L is the size of the input—we assume N will be smaller than 256. The three variables b, x, and y point towards arrays containing the coefficient, input, and output, respectively. As for the function~~s~~ body, the first for-loop initializes the taps while the second for-loop goes through all of the inputs and shifts them onto the taps and computes their impulse response.

At first glance the C code seems to be a good representation of the FIR filter. There are however a few problems with the implementation. For instance, the second of the two inner for-loops that calculates the intermediate result v does so by performing a dot-product of the arrays b and tap. Implementing a dot-product in this manner will tie it to our FIR filter, as opposed to a stand-alone function which can be used by many. While it is possible to extract the computation like so:

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

The function is still specialized to values of type *double*, it assumes *b* and *tap* both have at least *N* elements, and it is not readily compositional since the function cannot be merged with the producers of `xs` or `ys` without looking at its implementation.

A dot product can be implemented in our embedded co-design language, as shown in section 1, using a similar imperative, but not idiomatic, style:

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

#

That is, a program in the co-design languages is a monad. These monads acts as a kind of composable computation descriptions, and together with Haskell's `do` notation, allows for instructions to be sequenced very much like they are in C.

This version of a dot product is however not without fault. As with the C version, the function is locked to values of `Float`. We could however tackle this problem by making use of Haskell's `Num` class, changing the function's type while leaving its body intact:

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

The dot-product is now polymorphic in the kind of values it accepts, but limited to numerical values that supports addition and multiplication. Even with its new type, the function is quite still fragile: the for-loop's length and the array indexing are both handled manually. That is, a single typo in any of these two would break the function but not its type, creating an error that would first emerge at run-time.

For purely array based computations like the dot product, the idiomatic approach would instead be to use our vector language. Where users can build larger functions from its library of smaller, pre-verified functions. A vector version can be defined as follows:

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

it is

Here, the dot product is calculated by first joining the two vectors `xs` and `ys` by element-wise multiplication with `zipWith`, and then reducing the resulting list with `sum`.

The above version of the dot product is not only closer to its mathematical specification than the sequential one, but also sturdier in the sense that its harder for users to make an error. Furthermore, Haskell's lazy evaluation ensures that `dotVec` can be merged freely with the producers of `xs` and `ys`. isn't it more complicated?

For a full implementation of the FIR filter, the vector language is a bit outside its comfort zone. Vectors excel at describing array transformations, whereas the filter is described by a recurrence equation where output depends on previous input values. Nevertheless, the vector library does provide a few such recurrence functions, and we use one of them to implement the full filter:

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

Where `recurrenceI` takes an initial buffer, an input vector to iterate over, and a step function that produces one output at a time from the previous inputs and buffer. Recurrence functions can handle the regular feedback of a FIR filter quite well, but struggles for irregular access to earlier inputs.

delete?

odd sentence

While a FIR filter can be described using vectors they are ~~typically used in digital signal processing applications~~, and idiomatic approach for such functions is to instead use our signal processing language. The language is built upon the co-design language and introduces the concept of signals: possibly infinite sequences of values. Like the vector language, idiomatic signal functions are constructed compositionally using smaller functions, but signals also provide a function for introducing unit delays. As an example, we create three signal functions that represents the main components of a FIR filter:

```
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
```

That is, a summation and a multiplication with coefficients, which together form a dot product, and a number of successive delays to form the taps.

one sentence

We should note that `foldr1`, `zipWith`, `map` and `iterate` are the standard Haskell functions for lists. Whereas addition and multiplication are lifted to operate element-wise over signals. The other two functions, `constant` and `delay`, are signal functions that introduces a constant signal and a unit delay, respectively.

A full FIR filter can now be expressed as:

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

w

Which is quite close to the filter's mathematical specification: the input signal is delayed to form the filter's taps, where each "tap" is multiplied with a coefficient and summed.

I need to see the C & compare with that shown earlier

2.4 Summary

Section 1 gave a general introduction to embedded programming and its challenges. Specifically, the problem of extracting performance from an embedded system while keeping its power cost down was discussed. Heterogeneous systems was then introduced as a possible response to these problems, where section 2 mentioned modern FPGAs in particular as a heterogeneous system of interest.

I'm a bit worried about the emphasis on power since you will not measure it

Heterogeneous systems are not without their own challenges, as the presence of multiple processors raises all of the issues involved with parallel, homogeneous systems. Also, the

#

level of heterogeneity in a system can introduce additional challenges with different system capabilities and development between processors: components may support different instructions, leading to incompatibilities between the code they can execute even if they're both programmed in the same language.

Functional languages was then introduced in section 2.1 as a response to the various modularity issues with using lower-level languages like C or VHDL for heterogeneous systems. Particularly the "glue code" of functional languages, that is their higher-order functions, type-system, and lazy evaluation, was shown to be useful for developing reusable components. Section 2.2 showed how these benefits extended to languages embedded in a functional language like Haskell.

Section 2.3 went on to introduce our current attempt at bringing the benefits of functional programming languages to the domain of embedded heterogeneous systems with our hardware software co-design, vector and signal languages. The aim is to have the co-design language serve as a convenient description of imperative program descriptions for both software and hardware, with support for compilation to C and VHDL. Both the software and hardware parts are extensible to account for any differences in the intrinsics of components, and in terms of possible interpretations. The vector and signal languages are both built upon the co-design language. The first serves as a convenient front-end for array based programs, while the other extends it with support for expressing synchronous data-flow computations.

3 Co-Design

In section 2.3 we introduced a simplified version of our co-design, vector and signal languages to illustrate embedded programming in Haskell. We then went on to implement two versions of a FIR filter, one with vectors and one with signals. For the kind of heterogeneous computing our co-design language aims to describe, the simplified types we have shown so far are not enough. Heterogeneous systems typically see hardware code interleaved with software code, and our language therefore needs to be able to describe both. To facilitate design exploration we also need to support descriptions that are not restricted to either language, but rather the operations they require.

C and its dialects are perhaps the most commonly used languages for writing software in embedded systems. The co-design language is no different and compiles its software programs to C as well. For hardware descriptions, we use the VHDL language. This decision is purely based on the fact it is the hardware description language we are most comfortable with. Neither choice of language is final, and the language does support the addition of new target languages.

Starting with a single Haskell program, our co-design library is designed with three main tasks in mind: generate C code for the software parts, VHDL for the hardware parts, and to generate a combination of software and hardware for the transmission of data between components. Furthermore, the software and hardware programs are both extensible in the sense that they support the addition of new operations to account for differences between components.

not clear to me what this means

example?

3.1 Hardware Software Programs

not a sentence

While C and VHDL are different from one another in that one describes software code and the other hardware designs. Nevertheless, both languages use an imperative style of programming. The co-design language is therefore built on a deep embedding of monads, as a representation of imperative programs. Monads can be thought of as composable descriptions of computations, that is, they provide a means to connect smaller programs into a single, larger program.

The general idea behind a monadic embedding is that one can view an imperative program as a sequence of instructions to be executed on some machine—which looks similar to programs written in a stateful monad using Haskell's do-syntax. In fact, a stateful program composed with monadic operations can be directly translated into statements in an imperative language. As an example of these similarities, consider a software program for reversing an array:

```
reverseS :: SArr Int32 → Software ()
reverseS arr =
  for 0 (len ‘div‘ 2) $ λix → do
    aix ← getArr arr ix
    ajx ← getArr arr (len - ix - 1)
    setArr arr ix ajx
    setArr arr (len - ix - 1) aix
  where
    len = length arr
```

modulo

The implementation of `reverseS` certainly has the look and feel of a imperative program, sans a few syntactical differences. Its type signature tells us that it takes an array over 32-bit integers as input and produces a software program—notice that the array type `SArr` is prefixed with an `S` in order to show that it is intended to be used with software statements. The return type is empty since the array is reversed in place.

While the type of `reverseS` is that of a software program, there's nothing software specific about its implementation. For-loops and arrays are both part of most imperative languages, including C and VHDL. We could just as well have implemented reverse as a hardware function. In fact, we can define the hardware version by simply changing the previous function's type signature while keeping its body intact:

```
reverseH :: HArr Int32 → Hardware ()
reverseH arr =
  for 0 (len ‘div‘ 2) $ λix → do
    aix ← getArr arr ix
    ajx ← getArr arr (len - ix - 1)
    setArr arr ix ajx
    setArr arr (len - ix - 1) aix
  where
    len = length arr
```

#

The similarities between `reverseS` and `reverseH` hints that a software or hardware type for reverse is unnecessarily restrictive. A function type that is not limited to either language, but rather the functionality it requires, can be expressed in the co-design language thanks to type classes it provides. These classes are used to ensure a type provides the functionality a program requires, like Num did for the numerical expressions in section 2.3. That is, we can replace the software and hardware types with a generic program type and then add constraints for any functionality we require.

Going back to our reverse example, we can give it the following generic type:

```
reverse :: (Monad m, Arrays m, Control m, TypeM m Int32)
        ⇒ Arr m Int32 → m ()
```

SArr and HArr are substituted for the generic array type Arr, which is parameterized on the monad m. Such an associated type for arrays lets us talk about the array type of m; Arr will turn into either SArr or HArr when m is picked to be their respective monad. one of there ? monads

Three new constraints were introduced for the generic reverse, namely Arrays, Control, and TypeM. The first two ensure that m supports arrays and for-loops, whereas the third one states that 32-bit integers is a valid type in m. Arrays and Control are defined as follows:

```
class Monad m ⇒ Arrays m where
  type Arr m
  newArr :: TypeM m a ⇒ Exp m Length → m (Arr m a)
  getArr :: TypeM m a ⇒ Arr m a → Exp m Index → m a
  setArr :: TypeM m a ⇒ Arr m a → Exp m Index → a → m ()

class Monad m ⇒ Control m where
  for :: (TypeM m a, Integral a) ⇒ Exp m a → Exp m a → (Exp m a → m ())
      → m ()
```

'it provides

Each class lists the functions they provide and in the case of arrays, the type to use with them; Exp represents the expression type associated with m. Seeing as these operations are quite common, we give a short-hand for a collection of them plus references:

```
type MonadComp m = (Monad m, References m, Arrays m, Control m)
```

are　　　　that　　　　are

Just as there's classes of operations both software and hardware support, there's also operations that some languages support but others do not. The type classes therefore form a hierarchy. Monads form the base of the class hierarchy, but functions intended for either software or hardware branches also require that m is an extension of their respective monads. For example, processes in hardware are given by the following type class:

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

d

At this point we should note that types introduces in this section are slightly different from those found in section 2.3. For instance, the array type now has an extra parameter m. The earlier types are in fact synonyms for the software types introduced in this section, and served to give a simplified introduction to the co-design language. We reimplement the dot-product using the new types as a comparison to the old:

```
dot :: (MonadComp m, TypeM m a, Num a) ⇒ Arr m a → Arr m a → m (Exp m a)
dot x y = do
  sum ← initRef 0
  for 0 (min (length x) (length y) $ λix → do
    a ← getArr x ix
    b ← getArr y ix
    modifyRef sum $ λs → s + a * b
  getRef sum
```

The function's type has changed, as expected, but its body remains the same as before.

As an example of a larger function, we also implement the full FIR filter. The filter itself is fairly straightforward: inputs are shifted onto the taps one by one and for each input the current impulse response is calculated using a dot-product:

```
fir :: (MonadComp m, TypeM m a, Num a) ⇒ Arr m a → Arr m a → m (Arr m a)
fir bs xs = do
  taps ← newArr (length bs)
  ys   ← newArr (length xs)
  for 0 (length bs) $ λix → setArr taps ix 0
  for 0 (length xs) $ λix → do
    for 1 (length bs) $ λjx → do
      tmp ← getArr taps (jx - 1)
      setArr taps jx tmp
    x ← getArr xs ix
    setArr taps 0 x
    o ← dot bs taps
    setArr ys ix o
  return ys
```

While the above implementation is suitable for a software, it is perhaps not ideal as a hardware design. Signal processing in C is often done with arrays over chunks of the input, while a hardware is build around signals and processes to drive a continuous filter. Assuming a hardware description is our goal, the necessary change is however quite small: we swap the input and output arrays for signals and change the outer for-loop into a process. Rewriting programs at this scale is a kind of optimization we think developers are comfortable with.

Hmm. What about the nice combinator style fir filter. Now I am a bit confused (*)

3.2 Instructions

give real names too! and then B&B in bracket maybe?

The co-design language is inspired by the work of Björn and Benny [44] and by the Operational Monad [1] and is as such based on a monadic representation of imperative programs. Like our inspiration, 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. Unlike previous work, we also take into account the fact that different languages will support different expressions, types, etc. Some might even share, for instance, a sub-set of each others instructions but include a few of their own primitives—we would rather not copy an entire language just to add a new primitive. We have thus taken extra care to make sure instructions, and to some extent the expressions, can be defined compositionally. The program type is therefore parameterized on its instructions, as before, but also includes a type-level list for the other types associated with the language:

```
data Program instr fs a
```

The general idea behind the program type is that it allows for instructions to be separated from their sequencing, since an instruction's effect will only depend on its interaction with other instructions. That is, Haskell's monadic syntax ensures that any instructions we use in our programs are sequenced correctly. As a consequence of this separation, the task of implementing a language based on programs is the same as writing an interpreter for the language's instructions. In particular, we can define a generic interpreter for programs that maps them to their intended meaning:

```
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`, its possible to implement different "back ends" for programs. For example, interpretation in Haskell's `IO` monad gives a way to *run programs*, while interpretation in a code generation monad can be used to make a *compiler*. The interpretation of an instruction set `instr` to the monad `m` is given by `Interp`:

```
class Interp instr m fs where
  interp :: instr (P2 m fs) a → m a
```

Where `P2` is a synonym for a parameter list of two elements, here `m` and `fs`. Note also that `interpret` also require that its instructions are higher-order functors:

```
class HFunctor h where
  hfmap :: (∀ b . f b → g b) → h (P2 f fs) a → h (P2 g fs) a
```

Higher-order instructions, parameterized on the program they are part of, makes it possible to define instruction sets compositionally using, for instance, a technique like Data Types à La Carte [46]. Extensible instructions sets are particularly useful for our co-design language, as any existing hardware and software instructions can be reused for new systems; we only need to add support for new intrinsic instructions and expressions. That is, to extend either the hardware or software language we only need to add a data type of our new primitive, define a new set of instructions that includes it, and add support for its interpretation.

As an example of how an new instruction can be defined, we introduce the `If` construct:

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

The type parameter `prog` refers to sub-programs, `exp` refers to pure expressions, and `pred` refers to type predicates. Although `pred` is not used in the definition of `If`, it is used by other instructions and therefore included here to keep the parameter lists consistent. `P3` is a synonym for a parameter list of three arguments.

Assuming we have access to a few other instructions like `If`, we can now define a program type that combine several instructions in a Data Types à La Carte fashion:

```
type MyProgram exp pred = Program (If :+: Reference :+: ...) (P2 exp pred)
```

The recursive program type will then set the type of sub-programs in `If` to `MyProgram`.

The final step to our extension is to add support for interpreting `MyProgram`, that is, we must define an instances for `Interp` and `HFunctor`. Fortunately, both classes distribute over `(:+:)`, so we only need define instances for `If`. The `HFunctor` instance is defined as:

swap

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

While interpretation in, for example, the `IO` monad could look as follows:

```
instance Interp ControlCMD IO fs where
  interp (If b tru fls) = if evalExp b then tru else fls
```

W↓ Where `evalExp` is an evaluator for the expression language.

3.3 Expressions

A language embedding based on monads gives us a representation of the statements in an imperative program, but most meaningful programs also include a notion of pure expressions. These expressions contain a combination of one or more values, constants, variables and operators that our co-design library interprets and computes to produce a value. As a small example, consider a function for squaring a value:

```
square :: (Num (exp a), Type exp a) ⇒ exp a → exp a
square a = a * a
```

Note that the `Type` constraint on `a` is slightly different than `TypeM` from section 3.1, as `Type` accepts expressions rather than monads for its first argument.

Programming with expressions in our co-design language is evidently quite similar to how its done in regular Haskell. For instance, applying the squaring function to a value of 5 is equivalent to the mathematical expression $5 * 5$ which evaluates to 25. The expressions used in our co-design language are however deeply embedded, and as we saw in section 2.2, that means evaluation isn't the only interpretation they can support. In fact, by substituting the general `exp` type for either the software or hardware expression type, `SExp` and `HExp`, expressions can be compiled as well.

In simple expressions, like the above squaring function, the resulting value is usually one of the various primitive types, such as signed and unsigned numerical, floating point, or logical. In more elaborate expressions it can however be a complex data type. We can, for example, define an expression that consists of a pair of values:

```
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` computes the distance between two points in a plane, where points are represented as a pair of coordinates. The pairs are simply syntactic suger, and will not appear in the generated code.

In addition to the numerical and logical functions we have seen so far, expressions also provide a number of abstractions that we are accustomed to as functional programmers:

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

Abstractions like the let-binding are one of the hallmarks of functional programming and let users avoid unnecessary detail and mundane operations. At the same time, such abstractions can complicate the compiler and make it harder to generate efficient code. To circumvent this problem, the co-design language makes use of two expression types: one with only primitive operations that is easy to compile, and one with features like let-bindings that is comfortable to program with. Seeing as the feature rich expression type is an extension of the basic one, the two does inevitable share a few primitives and are therefore defined using a technique similar to Data Types à La Carte—like the instructions from section 3.2.

In order to avoid having separate instances of each interpretation for the two expression types, which is a tedious and error-prone task, we provide an elaboration from the feature rich expressions into program snippets over the primitive expressions:

```
elaborateSExp :: SExp a → Program SIns (P2 CExp SType) (CExp a)
```

Here, `SIns` and `SType` are the software instruction set and type predicate, and `CExp` is the primitive software expressions. The program wrapping is necessary in order to translate some expressions as, for instance, let-bindings use references to hold its shared value.

`elaborateSExp` provides a means to elaborate a single expression. For a full program, we need to elaborate every expression its instructions may contain. Programs are fortunately monads, and its thus possible to use the earlier `interpret` function from section 3.2 to elaborate entire programs—assuming we use `elaborateSExp` to give an instance of `Interp` for expressions. For example, a whole software program can be elaborated as:

y

```
elaborateSoft :: Software a → Program SIns (P2 CExp SType) a
elaborateSoft = interpret
```

Where an evaluator for software programs is then defined by combining the elaboration of expressions with an interpreter for its instructions:

corresponds to ?

```
runSoft :: Software a → IO a
runSoft = interpert . elaborateSoft
```

it is

all over the place

This approach to expressions and their interpretation has several benefits: the translation is typed, which rules out many potential errors; it is easier to write than a complete translation into source code; the low-level expression type is reusable and can be shared as an elaboration target between multiple high-level expression types.

O

a program that is...

#

3.4 Components

The ability to write programs that are constrained to its required functionality, rather than a specific language, means that users can easily experiment with putting their functions on different components in a system. Most interesting heterogeneous programs do however include a mixture of software and hardware fragments where the different parts communicate with each other. In the kind of FPGAs with embedded systems that we consider, communication is typically done over an AXI4 or AXI4-lite 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 makes it 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 full AXI4 certainly has its uses, there is no need for such features in the examples we have shown so far. The lighter interconnect offered by AXI4-lite is a better fit for our examples and will be the focus in this section.

maybe just say AXI4-lite implemented so far.

presumably full AXI4 is mentioned in future work

Five channels make up the bulk of the AXI4-lite specification: the read and write address channels, the read and write data channels, and the write acknowledge channel. These five channels are represented in VHDL as signals, driven by processes that implement the associated handshaking and logic for reading and writing. Signals behave very much like the references found in C, and a process is a kind of function that runs automatically once any of its inputs ~~changes~~ change. Both signals and processes are supported by our co-design language and a whole AXI4-lite interconnect is in fact implemented within the language as a function called `axi4lite`. The function takes a hardware component and connects it to the read and write channels of a new interconnect:

```
axi4lite ::
     Component a
  → Component (
          Sig (Bits 32) -- Write address.
       → Sig (Bits 3)  -- Write channel protection type.
       → Sig Bit       -- Write address valid.
       → Sig Bit       -- Write address ready.
       → Sig (Bits 32) -- Write data.
       → Sig (Bits 4)  -- Write strobes.
       → Sig Bit       -- Write valid.
       → Sig Bit       -- Write ready.
       → Sig (Bits 2)  -- Write response.
       → Sig Bit       -- Write response valid.
       → Sig Bit       -- Response ready.
       → Sig (Bits 32) -- Read address.
       → Sig (Bits 3)  -- Protection type.
       → Sig Bit       -- Read address valid.
       → Sig Bit       -- Read address ready.
       → Sig (Bits 32) -- Read data.
       → Sig (Bits 2)  -- Read response.
       → Sig Bit       -- Read valid.
       → Sig Bit       -- Read ready.
       → ())
```

Once the component has been wrapped it can be loaded into a synthesis tool like Vivado [20] to generate a physical design that we put onto, for example, an FPGA's logic blocks. Such hardware components can interface with each other using port-maps as usual, but with the physical address of the component in hand it's also possible to access it from software using memory-mapped I/O. The general idea is that memory-mapping a component causes it to share its address space with the memory of whatever software program is running, that is, a component can be reached from software by simply writing to and reading from pointers to its address. The software function `mmap` preforms the necessary memory-mapping for a component:

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

Note that the resulting memory pointer of `mmap` has “softened” its component's original type. Softening refers to the process of replacing any hardware specific types, like signals, with their corresponding types in software, and is described by the type family `Soften`:

```
type family Soften a where
  Soften (Sig a → b) = SRef a → Soften b
  ...
```

A softened pointer can then be called from software with a matching set of arguments:

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

As `call` goes through the argument list, it writes each value marked as input to the memory pointer—different input signals are reached by simply offsetting the address—and output values are read in a similar manner. The component's output can then be read from the argument matching its output signal. As for the argument list, it is a typed heterogeneous list that ensures we use the expected number of arguments and that all intermediate types match. A list of arguments is constructed with the following functions:

```
nil   :: Argument ()
(:>)  :: SType a ⇒ SRef a → Argument b → Argument (SRef a → b)
(:>>) :: SType a ⇒ SArr a → Argument b → Argument (SArr a → b)
```

`nil` creates an empty argument list, and the two infix functions `(:>)` and `(:>>)` extends an argument list with a reference and an array, respectively.

As an example of offloading a function to hardware, we revisit the earlier dot product from section 3.1 and recall that it had the following type:

```
dot :: (MonadComp m, TypeM m a, Num a) ⇒ Arr m a → Arr m a
  → Program (Exp m a)
```

While we as designers can read the type of dot-products ~~type~~, other functions cannot. So, before we can hook up the dot-product to an AXI4-lite interconnect and offload it, we need to give it a type signature that can be read by other functions. Seeing as the dot-product takes two arrays as input and then produces an expression as output, we can give it the following signature:

to do with ?
C

```
comp :: Component (HArr Int32 → HArr Int32 → Signal Int32 → ())
comp = inputArr 3 $ λa → inputArr 3 $ λb → returnC $ dot a b
```

Where inputArr adds an array of a given length to the signature, and returnC rounds of the type signature with a signal for the dot product's output. The wrapped dot can be hooked up to an interconnect with axi4lite and then be compiled, synthesized and put onto hardware.

In order to communicate with our offloaded dot-product from software we need to get its physical address from the synthesis tool, which is "0x4C300000" in our case. With the address in hand we can now put together a small software program that call s our dot-product and print its result to standard output:

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

3.5 Vectors

A seq. program — # #

Sequential programs in the co-design language makes use of its array type to express array and vector computations with mutable updates. These arrays gives a designer full control over their allocation and assignment, but do so through a low-level and imperative interface. As we saw in section 2.3, some functions are better expressed in a compositional manner than as a sequential program. To facilitate the design of array and vector functions, we provide the vector language.

A typical vector computation starts with a "manifest" vector, that is, a vector which that refers directly to an array 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 various names for "manifest", "pully" and "pushy" vectors draw inspiration from the Pan language [19] and push arrays [16], the ideas of which our vectors are based on.

on

Manifest vectors are more often than not the immutable arrays provided by the co-design language, as such arrays have an representation in memory. A Push vector on the other hand is not stored in memory, but is rather represented as a function: Pull

maybe italics instead of quotes ?

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

A pull 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.

Push vectors go in the opposite direction of pull vectors, and give us control over a vectors evaluation to the producers rather than the consumer. That is pull vectors have a representation that supports nested writes to memory and fusion of operations. Push vectors are represented as:

push

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

A push vector consists of a length, like pull vectors, but their function describes how elements are evaluated rather then how they are fetched. As such, they are parameterized on the type `m` rather than an expression type; `Exp` is an associated type, like `Arr` from section 3, and refers to the expression type associated with `m`. The general idea is to instantiate `m` as either of the co-design's two languages and have the push vector's function write each element to an array. In particular, push arrays implement efficient concatenation and interleaving, which would otherwise introduce unnecessary conditionals had they been implemented with pull vectors.

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

```
squares :: (Num a, Type exp a) ⇒ exp a → exp a
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 from one to n, to which a mapping is applied that squares each element. The vector is then converted into a push vector and summed up into a single value.

Each vector type has a different set of operations associated with it, and these operations are chosen in such a way that each vector type only supports those operations which can be performed efficiently for that type. 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 turns back into a manifest vector. There are however cases where its preferable to "skip" parts of the cycle. For instance, the `squares` function starts with a pull vector rather than a manifest vector.

The functions associated with each kind of vector are overloaded in the kind of vectors they accept: an operation for a pull vector will support the use of any "pully" vector type. For instance, the `sum` function used in the above `squares` is defined as follows:

```
sum :: (Pully exp vec a, Type exp a, Num a) ⇒ vec → a
sum = fold (+) 0
```

3.6 Signal processing

While the imperative style of programming of the co-design language is already convenient for software realization, section 2.3 showed that compositional descriptions can yield a better intuition of what constitutes a program. The vector language was therefore introduced and provides a comfortable syntax for composing sub-components to form larger, array based programs. For functions that involve a notion of time, or are simply better expressed through streaming directly rather than being converted into a function over arrays, we provide the signal language.

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—we cannot construct a signal over other signals.

Programming of signals is done compositionally, and a set of functions is provided to support the composition of new signals from existing ones. That is, a signal program is a collection of mutually recursive signal functions, each built from repeating values or other signals. For instance, some of the supplied combinatorial functions include:

```
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
```

`repeat` creates a signal by repeating some value; `map` applies a function to each value of a signal; `zipWith` joins two signals element-wise using then given function. The general idea is that every ground type in the expression language is lifted to operate element-wise over signals by, for example, using the above functions or type classes like `Num`:

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

The above signal functions are all combinatorial in the sense that their output only depends on the current inputs. Most interesting examples of signal processing does carry some form of state. A sequential operator for unit delays is therefore provided as well:

```
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. Note that `delay` introduces the notion of a *next time step*, making time enumerable.

While these functions may appear innocent, `repeat`, `map` and `zipWith` can express most combinatorial signal networks, while the combination of `delay` with feedback can describe any kind of sequential signal network. For instance, we can define a parity checker 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 of feedback, we implement a infinite impulse response (IIR) filter, which comprises the second primary type of digital filters used in digital signal processing applications and, unlike FIR filter in section 2.3, contains feedback. The IIR filter is typically described and implemented in terms of a difference equation:

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

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 we can see that the IIR filter loosely consists of two FIR filters, where the second filter has an extra delay and is recursively defined on its output. As such, we can define the filter as:

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

With the FIR filter defined as.

```
fir :: (Fractional a, Num (exp a), pred a) ⇒ [exp a]
  → Sig exp pred a → Sig exp pred a
fir as x = P.foldr1 (+)
         $ P.zipWith (*) (P.map repeat as)
         $ P.iterate (delay a) x
```

Where functions prefixed by `P` are Haskell's standard list function rather than signal functions.

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 we have that recursively defined signals introduce feedback, while recursion over other Haskell values like lists can be used to build repeating graphs structures.

This behavior of `delay` implies that we can distinguish values that have and haven't been delayed, which is something that's normally not possible to do in Haskell—as being able to observe the sharing of `y` will, by definition, break any referential transparency. In fact, the internals of `delay` makes use of a restricted form of observable sharing [15, 23]. This allows us to turn signal functions like the above IIR filter, which describes a network of operations, into a directed graph. Any sharing is then visible as edges in the graph, connecting nodes over its operations.

A graph representation of a signal network enables us to check for cycles, order the nodes, and, finally, compile them into programs. A regular program does however not capture the streaming nature of a signal function, as it represents an action we can run in order to produce a value. So instead of compiling to a program directly, the signal language translates a signal into an co-iterative stream [14].

Co-iteration consists of associating to each stream an initial state and a transition function from old state to a pair of a value and a new state. The benefit of this approach

a

is that it allows us to handle infinite data types, like streams, in a strict and efficient way. A slightly modified version of co-iterative streams is provided by the signal language, where state is represented implicitly by the `Program` monad:

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

The outer monad is used to initialize the stream, where the result of the initialization is another program that produces the output values.

The compiler for signal functions is then given as transformation from a function over signals to a function over streams:

```
compileF1 :: (Sig exp pred a → Sig exp pred b)
  → IO (Str instr exp pred a → Str instr exp pred b)
```

Where the `IO` type is a result of observable-sharing.

Now I want to know what happens when you run the examples on the FPGA !! *

4 Concluding Remarks

A substantial amount of research has gone into addressing the challenges of programming for FPGAs, opening them up for programmers without a background in hardware design. However, hardware description languages are still the most commonly used tools for programming FPGAs. These languages have revolutionized hardware design but suffer from a lack of expressiveness and standardization – there is a mismatch between description and synthesized hardware. Designers are therefore looking for alternative solutions, where two of the more well-known approaches are: extending HDLs with features found in modern languages, and synthesis of from high-level languages.

Compiling high-level languages to a hardware description has a great appeal, finding a translation between the two has however proven to be difficult. Tools like Catapult C [25] are able to generate register transfer level code from ordinary C descriptions, but sequential programs are often a bad fit for the parallelism inherent to most hardware architectures. Additional attempts includes creating dialects of the host language, such as System C [22].

Another group of languages whom that have shown success in describing hardware designs are the functional languages [40]. Higher-order functional languages in particular, where hardware descriptions are first-class objects, offer a useful abstraction mechanism which captures many useful design patterns: hardware descriptions can be parameterized and passed around as parameters themselves. An additional attribute of these languages is their purity, enabling formal reasoning about function (de-)composition. Without side-effects, a synthesis tool can derive the inherent parallelism of a functional description as it only has to respect data dependencies.

There has been a number of other approaches to hardware description using functional languages, most of which are focused on describing hardware through structural descriptions rather than sequential algorithms. Sheeran pioneered this approach with μFP [42], which utilizes a set of functional combinators to support the composition of

York Lava dealt with the h/w part of FPGA prog definitely should be mentioned explicitly.

will send paper or link in email

But I think it did not do codesign

larger descriptions from existing ones. Various languages from the Lava family [9, 24] of languages have since built upon these ideas. None has however, to the extent of my knowledge, ventured into the design of heterogeneous architectures, like that of a modern FPGA. The benefits of figuring out how to programme such system is however in no way limited to FPGAs, as most future processors are likely going to be heterogeneous in exactly the same way [41].

The functional approach to hardware description also includes languages outside of the Lava family, like CλASH [4], a compiler capable of translating a subset of Haskell into synthesizable hardware. CλASH is however limited to describing hardware and forgoes support of other components. Bluespec [34], a HDL with strong influence from functional languages, does support both software and hardware descriptions and includes concepts such as higher-order functions and polymorphism. Nevertheless, Bluespec descriptions are written at a clock-cycle granularity and therefore provide a lower level of abstractions than most functional languages.

Lightweight Modular Staging (LMS) has also been explored as an option to ease the construction of a domain-specific High Level Synthesis (HLS) system [21]. 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-level transfer descriptions. Though the language-specific challenges for LMS are different from ours, 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 [11], a DSL for generating JavaScript.

Outside the domain of HLS, examples of DSLs with similar ambitions to ours include the Cryptol DSL for cryptography [12] and Microsoft's Accelerator [47] for programming GPUs and various other platforms—although the latest versions of Cryptol no longer support hardware generation.

The signal processing is based on the synchronous data-flow paradigm and is inspired by similar languages from this domain. Of the synchronous languages, Lucid Synchrone [39, 17] is perhaps our biggest inspiration and it is designed to be used with reactive systems. Initially, the language was introduced as an extension of LUSTRE [26], 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 EDSLs such as our co-design library. Zélus [10], a successor of Lucid Synchrone, has shown that synchronous languages can be extended to model hybrid systems as well, that is, system which consist of both continuous and discrete components.

Another, and rather different approach to modeling signal processing is functional reactive programming. Yampa [18] 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 [35].

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 language. An event is usually a separate entity, modeling the control flow of a system. Some languages do, however, 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.

an ?

4.1 Discussion

Whether measured in the number of operations per seconds a system can handle or its power management size, performance is perhaps the most important feature of embedded systems. Since a embedded system is dedicated to a specific task, developers can optimize it to increase performance. To do so, they take advantage of the low-level intrinsics of the system, often by using a equally low-level programming language. However, this reliance on low-level characteristics of a system also has the unfortunate side-effect of restricting programs to the systems they're initially written for.

Low-level languages provide little in the way of abstractions to address the above modularity problems. In a heterogeneous programming setting, this problem is aggravated as several programming languages are used to describe the different components of a system. Each of these languages could potentially have their own set of intrinsics operations, or even belong to different programming paradigms.

This thesis presents our first steps towards a system in which the entire design process of a heterogeneous system. However, instead of taking on the challenge of heterogeneous system design head on, we are currently exploring a more modest approach; develop a library, embedded in Haskell, for hardware-software co-design, and see how far we can get. The co-design library introduced in this thesis is designed with this goal in mind, and provides a means to write a mixture of software and hardware functions, with a reasonable degree of control over the generated C and VHDL code. Being embedded in Haskell, we exploit its parametric polymorphism to facilitate functions that can be interpreted in software and hardware. This in turn allows the exploration to decide where the boundary between hardware and software should be, to be done entirely in Haskell.

Sentence not finished ?

This has not been demonstrated in the intro

The co-design library is modular in the sense that new instructions and interpretations can be added with relatively little disturbance to the existing system, allowing the library to be used in the development of new languages or simply to extend the current software and hardware language with new intrinsic operations. The library is also modular from the users perspective, as they can write generic code through the use of free abstractions. For instance, the pairs used in section 3.3 let us treat a point in space as a pair of coordinates, and when compiled, had completely disappeared. These pair are an example of a shallowly embedded type, and when paired with a translation into the deeply embedded core language, present a type of abstraction that only resides in Haskell. It is also possible to derive new vector combinators from existing ones without incurring any extra memory usage or traversals, since the vector library guarantees fusion of its operations.

That

means what?

Currently, the co-design library provides two interpretations of its programs: evaluation and compilation. Hardware programs are however often simulated to explore a systems response to some combination of inputs. The final system is intended to support additional interpretations, like simulation, as well. Streaming is also limited in hardware, as the co-design library ~~the~~ only provides an AXI4-lite interconnect. Full AXI4 is however expressed as a regular hardware design, just like AXI4-lite, and we intend to implement one sentence

it within the co-design library. Another possible extension would be to provide access to other components of an embedded system.

Writing the kind of digital signal processing software that is typically used with embedded systems is however still a tedious task. Primarily because of the focus on expressing it algorithms with relatively low-level traversals such as for-loops. The vector library alleviates this a bit, as it provides a number of useful abstractions for expressing array computations. Some algorithms are however better described in a data-flow style, where computations are treated like black boxes, which are connected together in a flow graph. The signal library introduced in this thesis addresses this issue, and provides a means to extend a library that has a notion of pure expressions, like the co-design one, with support for synchronous data-flow operations. The signal library is constructed in a similar fashion as the co-design library, and consists of a deeply embedded core language. As such, it retains much of the elegance and modularity of traditional functional programming for signals.

We intend to further build up the signal library by taking inspiration from related work in synchronous data-flow and introduce clocks as types [8]. This would allow the library to deal correctly with (over-)sampling, and to describe multi-clocked networks. Another interesting direction for signal processing could be to investigate other streaming models, such as the one used in Ziri [43, 13].

a

4.2 Conclusion

have

The co-design library is based on a combination of shallowly and deeply embedded types, which makes the library modular. Furthermore, the software and hardware languages embedded within the library has been made extensible, to support dialects intended for different components. Both languages are also based on the same representation of imperative programs, which is parameterized on the language's instruction set, and their expression and predicate types. So, with the help of this program type, users wishing to define a new imperative language only have to give an implementation of these three types. They get the rest for free.

refer to what?

Hardware and software programs are designed to both be languages with predictable performance, where the translation between design and source code transparent and easy to influence. For example, the monadic style of memory management is used in favor of high-level synthesis, which can have results that are difficult to predict. Testing of software programs is supported through a variety of interpretation functions, one of which compares the result of running the generated C code with the result of Haskell function—which lets us compare a software program to a model implementation in Haskell. For hardware programs, we currently rely on performing simulation and testing in an external synthesis tool.

is? running the

The signal library is based on a deeply embedded type, representing the core operations in synchronous data-flow programming. Type classes are used to lift standard Haskell operations to operate point-wise over streams, letting the library retain much of the elegance and modularity of traditional functional programming.

You need to summarise contributions

& say what is in the upcoming papers

References

[1] H. Apfelmus. *The Operational Monad Tutorial (Blog Post).* http://apfelmus.nfshost.com/articles monad.html. 2017 (cit. on p. 16).

[2] Markus Aronsson, Emil Axelsson, and Mary Sheeran. "Stream Processing for Embedded Domain Specific Languages". *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages.* ACM. 2014, p. 8 (cit. on pp. v, 37).

[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, 39).

[4] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. "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 pp. 3, 27).

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a scala embedded language". *Proceedings of the 49th Annual Design Automation Conference.* ACM. 2012, pp. 1216–1225.

[6] 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).

[7] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools.* " O'Reilly Media, Inc.", 2006 (cit. on p. 3).

[8] 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 (cit. on p. 29).

[9] 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, 8, 27).

[10] Timothy Bourke and Marc Pouzet. "Zélus: A Synchronous Language with ODEs". *16th International Conference on Hybrid Systems: Computation and Control (HSCC'13).* Mar. 2013, pp. 113–118. URL: `http://www.di.ens.fr/~pouzet/bib/hscc13.pdf` (cit. on p. 27).

[11] 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. 27).

[12] 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. 27).

[13] 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 (cit. on p. 29).

[14] 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. 25).

[15] Koen Claessen and David Sands. "Observable sharing for functional circuit description". *Annual Asian Computing Science Conference.* Springer. 1999, pp. 62–73 (cit. on p. 25).

[16] 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 p. 22).

[17] 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. 27).

[18] 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. 27).

[19] Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. *Journal of functional programming* **13**.3 (2003), 455–481 (cit. on pp. 8, 22).

[20] Tom Feist. Vivado design suite. *White Paper* **5** (2012) (cit. on p. 21).

[21] 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. 27).

[22] Frank Ghenassia et al. *Transaction-level modeling with SystemC.* Springer, 2005 (cit. on p. 26).

[23] 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. 25).

[24] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. "Introducing kansas lava". *Implementation and Application of Functional Languages.* Springer, 2010, pp. 18–35 (cit. on pp. 3, 27).

[25] Mentor Graphics. Catapult C synthesis. *Website: http://www. mentor. com* (2008) (cit. on p. 26).

[26] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems.* International Series in Engineering and Computer Science 215. Springer, 1993 (cit. on p. 27).

[27] John Hughes. "The design of a pretty-printing library". *International School on Advanced Functional Programming.* Springer. 1995, pp. 53–96 (cit. on p. 8).

[28] John Hughes. Why functional programming matters. *The computer journal* **32**.2 (1989), 98–107 (cit. on p. 6).

[29] 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.

[30] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world (2002) (cit. on p. 8).

[31] R McMillan. Microsoft supercharges Bing search with programmable chips. *Wired Business* **16** (2014) (cit. on p. 5).

[32] Trevor Mudge. Power: A first-class architectural design constraint. *Computer* **34**.4 (2001), 52–58 (cit. on p. 3).

[33] Cisco Visual Networking Index. Forecast and methodology, 2016-2021, white paper. *San Jose, CA, USA* (2016).

[34] 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. 27).

[35] 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. 27).

[36] W Obile. *Ericsson Mobility Report.* 2016.

[37] 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. 10).

[38] Anders Persson. Towards a functional programming language for baseband signal processing (2014).

[39] 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. 27).

[40] Mary Sheeran. Hardware Design and Functional Programming: a Perfect Match. *J. UCS* **11**.7 (2005), 1135–1158 (cit. on pp. 6, 26).

[41] Mary Sheeran. "Hardware Design and Functional Programming: Still Interesting after All These Years". Presented the 20th International Conference on Functional Programming, 2015 (cit. on p. 27).

[42] 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. 26).

[43] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. Ziria: A DSL for wireless systems programming. *ACM SIGPLAN Notices* **50**.4 (2015), 415–428 (cit. on p. 29).

[44] 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. 16).

[45] 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. 9).

[46] Wouter Swierstra. Data Types à La Carte. *J. Funct. Program.* **18**.4 (July 2008) (cit. on p. 17).

[47] 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. 27).

Part II
Appended Papers

Paper A

Stream Processing for Embedded Domain Specific Languages

Paper B

Hardware software co-design in Haskell