Haskell as a higher order structural hardware description language

Master's thesis

by

Matthijs Kooijman

December 9, 2009

Committee:
Dr. Ir. Jan Kuper
Ir. Marco Gerards
Ir. Bert Molenkamp
Dr. Ir. Sabih Gerez

Computer Architecture for Embedded Systems
Faculty of EEMCS
University of Twente

Abstract

Functional hardware description languages have been around for a while, but never saw adoption on a large scale. Even though advanced features like higher order functions and polymorphism could enable very natural parametrization of hardware descriptions, the conventional hardware description languages VHDL and Verilog are still most widely used.

Cλash is a new functional hardware description language using Haskell's syntax and semantics. It allows structurally describing synchronous hardware, using normal Haskell syntax combined with a selection of built-in functions for operations like addition or list operations. More complex constructions like higher order functions and polymorphism are fully supported.

Cλash supports stateful descriptions through explicit descriptions of a function's state. Every function accepts an argument containing its current state and returns its updated state as a part of its result. This means every function is called exactly once for each cycle, limiting Cλash to synchronous systems with a single clock domain.

A prototype compiler for Cλash has been implemented that can generate an equivalent VHDL description (using mostly structural VHDL). The prototype uses the front-end (parser, type-checker, desugarer) of the existing GHC Haskell compiler. This front-end generates a *Core* version of the description, which is a very small typed functional language. A normalizing system of transformations brings this Core version into a normal form that has any complex parts (higher order functions, polymorphism, complex nested structures) removed. The normal form can then be easily translated to VHDL. This design has proven to be very suitable. In particular the transformation system allows for some theoretical analysis and proofs about the compiler design itself (which have been left as future work).

Using Haskell syntax, semantics and existing tools has enabled the rapid creation of the prototype, but it also became clear that Haskell is not the ideal language for hardware specification. The problems encountered could be solved with syntax extensions and major improvements to Haskell's dependent typing support, or be circumvented entirely by developing a completely new language.

Cλash already allows for implementing real world systems, but has not seen much testing yet. There is much room for improvement, but there is also a clear path forward for further research.

Acknowledgements

This thesis and the research it presents is not the result of just my own work. I have received a lot of ideas, inspiration and comments from other people, who I would like to mention here.

First of all, I would like to thank Christiaan Baaij, which has been my partner in crime during this research. Together we have started a new area of research at the University of Twente, in which I would not have progressed so far on my own.

Then there is my first supervisor, Jan Kuper, whose visions and ideas have sparked this research and who always managed to place my practical views into the proper theoretical background. The rest of the graduation committee has been a great help, providing feedback, comments and challenging questions, each from their own unique point of view.

Finally, I would like to thank Brenda, who has been suffering my working late and general lack of time for just over a year now.

Contents

Introduction

This thesis describes the result and process of my work during my Master's assignment. In these pages, I will introduce the world of hardware descriptions, the world of functional languages and compilers and introduce the hardware description language Cλash that will connect these worlds and puts a step towards making hardware programming on the whole easier, more maintainable and generally more pleasant.

This assignment has been performed in close cooperation with Christiaan Baaij, whose Master's thesis [1] has been completed at the same time as this thesis. Where this thesis focuses on the interpretation of the Haskell language and the compilation process, [1] has a more thorough study of the field, explores more advanced types and provides a case study.

Research goals

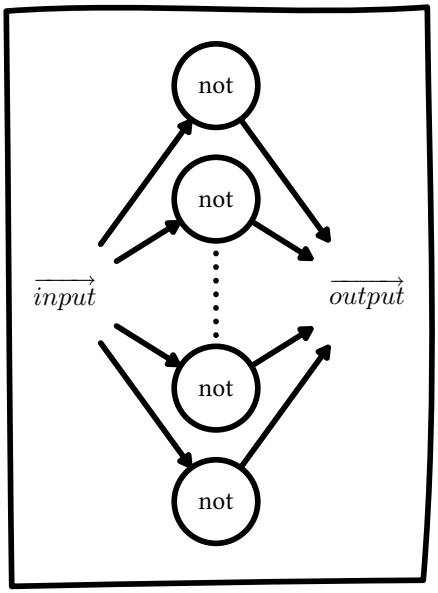
This research started out with the notion that a functional program is very easy to interpret as a hardware description. A functional program typically makes no assumptions about evaluation order and does not have any side effects. This fits hardware nicely, since the evaluation order for hardware is simply everything in parallel.

As a motivating example, consider the simple functional program shown in example 1[1]. This is a very natural way to describe a lot of parallel not ports, that perform a bit-wise not on a bit-vector. The example also shows an image of the architecture described.

Example 1 Simple architecture that inverts a vector of bits.

```
notword :: [Bit] → [Bit]
notword = map not
```

Haskell description of the architecture.



The architecture described by the Haskell description.

Slightly more complicated is the incremental summation of values shown in example 2[1].

In this example we see a recursive function `sum'` that recurses over a list and takes an accumulator argument that stores the sum so far. On each step of the recursion, another number from the input vector is added to the accumulator and each intermediate step returns its result.

This is a nice description of a series of sequential adders that produce the incremental sums of a vector of numbers. For an input list of length 4, the corresponding architecture is show in the example.

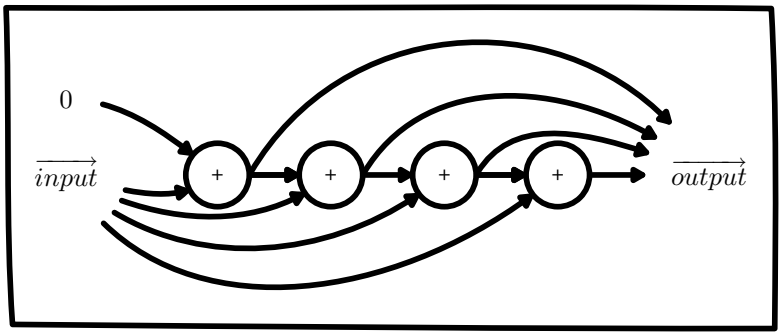
[1] *This example is not in the final Cλash syntax*

Example 2 A recursive description that sums values.

```
sum :: [Int] → [Int]
sum = sum' 0

sum' :: Int → [Int] → [Int]
sum' acc [] = []
sum' acc (x:xs) = acc' : (sum' acc' xs)
  where acc' = x + acc
```

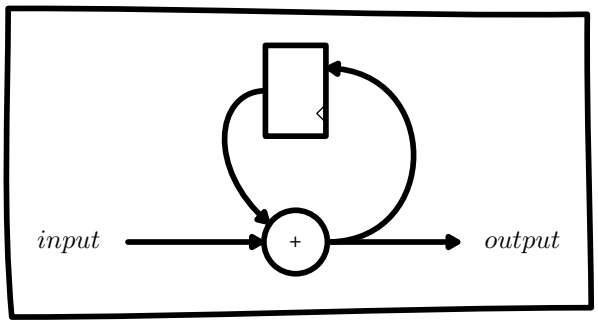
Haskell description of the architecture.



The architecture described by the Haskell description.

Or... is this the description of a single accumulating adder, that will add one element of each input each clock cycle and has a reset value of 0? In that case, we would have described the architecture show in example 3

Example 3 An alternative interpretation of the description in example 2



The distinction in possible interpretations we see here, is an important distinction in this research. In the first figure, the recursion in the code is taken as recursion in space and each recursion step results in a different piece of hardware, all of which are active simultaneously. In the second figure, the recursion in the code is taken as recursion in time and each recursion step is executed sequentially, *on the same piece of hardware.*

In this research we explore how to apply these two interpretations to hardware descriptions. Additionally, we explore how other functional programming concepts can be applied to hardware descriptions to give use an efficient way to describe hardware.

This leads us to the following research question:

> "How can we describe the structural properties of a hardware design, using a functional language?"

We can further split this into sub-questions from a hardware perspective:

- How can we describe a stateful design?
- How can we describe (hierarchical) structure in a design?

And sub-questions from a functional perspective:

- How to interpret recursion in descriptions?
- How to interpret polymorphism?
- How to interpret higher-order descriptions?

In addition to looking at designing a hardware description language, we will also implement a prototype to test ideas. This prototype will translate hardware descriptions written in the Haskell functional language to simple (netlist-like) hardware descriptions in the VHDL language. The reasons for choosing these languages are detailed in section 3.1 and 3.2 respectively.

The result of this research will thus be a prototype compiler and a language that it can compile, to which we will refer to as the Cλash system and Cλash language for short, or simply Cλash.

The name Cλash

The name Cλash more-or-less expands to CAES language for hardware descriptions, where CAES refers to the research chair where this project was undertaken (Computer Architectures for Embedded Systems). The lambda in the name is of course a reference to the lambda abstraction, which is an essential element of most functional languages (and is also prominent in the Haskell logo).
Cλash is pronounced like 'Clash'.

Outline

In the first chapter, we will sketch the context for this research. The current state and history of hardware description languages will be briefly discussed, as well as the state and history of functional programming. Since we are not the first to have merged these approaches, a number of other functional hardware description languages are briefly described.

Chapter two describes the exploratory part of this research: how can we describe hardware using a functional language and how can we use functional concepts for hardware descriptions?

Chapter three talks about the prototype that was created and which has guided most of the research. There is some attention for the language chosen for our hardware descriptions, Haskell. The possibilities and limits of this prototype are further explored.

During the creation of the prototype, it became apparent that some structured way of doing program transformations was required. Doing ad hoc interpretation of the hardware description proved non-scalable. These transformations and their application are the subject of the fourth chapter.

The next chapter sketches ideas for further research, which are many. Some of them have seen some initial exploration and could provide a basis for future work in this area.

Finally, we present our conclusions.

1 Context

An obvious question that arises when starting any research is 'Has this not been done before?'. Using a functional language for describing hardware is not a new idea at all. In fact, there has been research into functional hardware description even before the conventional hardware description languages were created. Examples of these are μFP [4] and Ruby [5]. Functional languages were not nearly as advanced as they are now, and functional hardware description never really got off.

Recently, there have been some renewed efforts, especially using the Haskell functional language. Examples are Lava [6] (an EDSL) and ForSyde [7] (an EDSL using Template Haskell). [1] has a more complete overview of the current field.

We will now have a look at the existing hardware description languages, both conventional and functional to see the fields in which Cλash is operating.

Embedded domain-specific languages (EDSL)

> "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain." [2]

An embedded DSL is a DSL that is embedded in another language. Haskell is commonly used to embed DSLs in, which typically means a number of Haskell functions and types are made available that can be called to construct a large value of some domain-specific datatype (*e.g.*, a circuit datatype). This generated datatype can then be processed further by the EDSL 'compiler' (which runs in the same environment as the EDSL itself. The embedded language is then a, mostly applicative, subset of Haskell where the library functions are the primitives. Sometimes advanced Haskell features such as polymorphism, higher-order values or type classes can be used in the embedded language. [3]

For an EDSL, the definitions of compile-time and run-time are typically fuzzy (and thus hard to define here), since the EDSL 'compiler' is usually compiled by the same Haskell compiler as the EDSL program itself.

1.1 Conventional hardware description languages

Considering that we already have some hardware description languages like VHDL and Verilog, why would we need anything else? By introducing the functional style to hardware description, we hope to obtain a hardware description language that is:

- More concise. Functional programs are known for their conciseness and ability to provide reusable functions that are abstractions of common patterns. This is largely enabled by features like an advanced type system with polymorphism and higher-order functions.
- Type-safer. Functional programs typically have a highly expressive type system, which allows a programmer to more strictly define limitations on values, making it easier to find violations and thus bugs.
- Easy to process. Functional languages have nice properties like purity and single binding behavior, which make it easy to apply program transformations and optimizations and could potentially simplify program verification.

purity p.25

1.2 Existing functional hardware description languages

As noted above, this path has been explored before. However, current embedded functional hardware description languages (in particular those using Haskell) are limited. Below a number of downsides are sketched of the recent attempts using the Haskell language. [1] has a more complete overview of these and other languages.

This list uses Lava and ForSyDe as examples, but tries to generalize the conclusions to the techniques of embedding a DSL and using Template Haskell.

Template Haskell

Template Haskell is an extension to the GHC compiler that allows a program to mark some parts to be evaluated *at compile time*. These *templates* can then access the abstract syntax tree (AST) of the program that is being compiled and generate parts of this AST.
Template Haskell is a very powerful, but also complex mechanism. It was intended to simplify the generation of some repetitive pieces of code, but its introspection features have inspired all sorts of applications, such as hardware description compilers.

- Not all of Haskell's constructs can be captured by embedded domain specific languages. For example, an `if` or `case` expression is typically executed once when the Haskell description is processed and only the result is reflected in the generated data-structure (and thus in the final program). In Lava, for example, conditional assignment can only be described by using explicit multiplexer components, not using any of Haskell's compact mechanisms (such as `case` expressions or pattern matching).

  Also, sharing of variables (*e.g.*, using the same variable twice while only calculating it once) and cycles in circuits are non-trivial to properly and safely translate (though there is some work to fix this, but that has not been possible in a completely reliable way yet [8]).

- Template Haskell makes descriptions verbose. Template Haskell needs extra annotations to move things around between TH and the normal program. When TH is combined with an EDSL approach, it can get confusing when to use TH and when not to.

- Function hierarchies cannot be observed in an EDSL. For example, Lava generates a single flat VHDL architecture, which has no structure whatsoever. Even though this is strictly correct, it offers no support to the synthesis software about which parts of the system can best be placed together and makes debugging the system very hard.

  It is possible to add explicit annotation to overcome this limitation (ForSyDe does this using Template Haskell), but this adds extra verbosity again.

- Processing in Template Haskell can become very complex, since it works on the Haskell AST directly. This means that every part of the Haskell syntax that is used must be supported explicitly by the Template Haskell code. Chapter 3 shows that working on a smaller AST is much more efficient.

2 Hardware description

In this chapter an overview will be provided of the hardware description language that was created and the issues that have arisen in the process. The focus will be on the issues of the language, not the implementation. The prototype implementation will be discussed in chapter 3.

To translate Haskell to hardware, every Haskell construct needs a translation to VHDL. There are often multiple valid translations possible. When faced with choices, the most obvious choice has been chosen wherever possible. In a lot of cases, when a programmer looks at a functional hardware description it is completely clear what hardware is described. We want our translator to generate exactly that hardware whenever possible, to make working with CλaSH as intuitive as possible.

In this chapter we describe how to interpret a Haskell program from a hardware perspective. We provide a description of each Haskell language element that needs translation, to provide a clear picture of what is supported and how.

Reading the examples

In this thesis, a lot of functional code will be presented. Part of this will be valid CλaSH code, but others will just be small Haskell or Core snippets to illustrate a concept.
In these examples, some functions and types will be used, without properly defining every one of them. These functions (like `and`, `not`, `add`, `+`, etc.) and types (like `Bit`, `Word`, `Bool`, etc.) are usually pretty self-explanatory.
The special type `[t]` means 'list of `t`'s', where `t` can be any other type.
Of particular note is the use of the `::` operator. It is used in Haskell to explicitly declare the type of function or let binding. In these examples and the text, we will occasionally use this operator to show the type of arbitrary expressions, even where this would not normally be valid. Just reading the `::` operator as 'and also note that *this* expression has *this* type' should work out.

2.1 Function application

The basic syntactic elements of a functional program are functions and function application. These have a single obvious VHDL translation: each top level function becomes a hardware component, where each argument is an input port and the result value is the (single) output port. This output port can have a complex type (such as a tuple), so having just a single output port does not pose a limitation.

Each function application in turn becomes component instantiation. Here, the result of each argument expression is assigned to a signal, which is mapped to the corresponding input port. The output port of the function is also mapped to a signal, which is used as the result of the application.

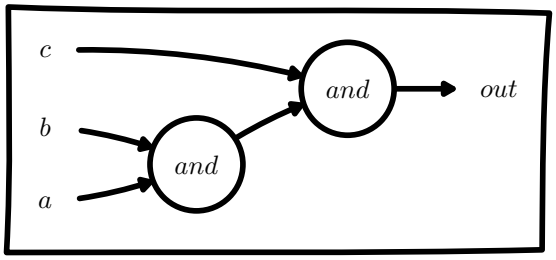
Since every top level function generates its own component, the hierarchy of of function calls is reflected in the final VHDL output as well, creating a hierarchical VHDL description of the hardware. This separation in different components makes the resulting VHDL output easier to read and debug.

Example 2.1 shows a simple program using only function application and the corresponding architecture.

Example 2.1 Simple three input and gate.

```
-- A simple function that returns
-- conjunction of three bits
and3 :: Bit -> Bit -> Bit -> Bit
and3 a b c = and (and a b) c
```

Haskell description using function applications.



The architecture described by the Haskell description.

Example 2.2 VHDL generated for `and3` from example 2.1

```
entity and3Component_0 is
     port (\azMyG2\ : in std_logic;
           \bzMyI2\ : in std_logic;
           \czMyK2\ : in std_logic;
           \foozMySzMyS2\ : out std_logic;
           clock : in std_logic;
           resetn : in std_logic);
end entity and3Component_0;


architecture structural of and3Component_0 is
     signal \argzMyMzMyM2\ : std_logic;
begin
     \argzMyMzMyM2\ <= \azMyG2\ and \bzMyI2\;

     \foozMySzMyS2\ <= \argzMyMzMyM2\ and \czMyK2\;
end architecture structural;
```

2.2 Choice

Although describing components and connections allows us to describe a lot of hardware designs already, there is an obvious thing missing: choice. We need some way to be able to choose between values based on another value. In Haskell, choice is achieved by `case` expressions, `if` expressions, pattern matching and guards.

An obvious way to add choice to our language without having to recognize any of Haskell's syntax, would be to add a primitive '`if`' function. This function would take three arguments: the condition, the value to return when the condition is true and the value to return when the condition is false.

This `if` function would then essentially describe a multiplexer and allows us to describe any architecture that uses multiplexers.

However, to be able to describe our hardware in a more convenient way, we also want to translate Haskell's choice mechanisms. The easiest of these are of course case expressions (and `if` expressions, which can be very directly translated to `case` expressions). A `case` expression can in turn simply be translated to a conditional assignment, where the conditions use equality comparisons against the constructors in the `case` expressions.

In example 2.3 two versions of an inverter are shown. The first uses a simple `case` expression, scrutinizing a Boolean value. The corresponding architecture has a comparator to determine which of the constructors is on the `in` input. There is a multiplexer to select the output signal (which is just a conditional assignment in the generated VHDL). The two options for the output signals are just constants, but these could have been more complex expressions (in which case also both of them would be working in parallel, regardless of which output would be chosen eventually). The VHDL generated for (both versions of) this inverter is shown in example 2.4.

If we would translate a Boolean to a bit value, we could of course remove the comparator and directly feed 'in' into the multiplexer (or even use an inverter instead of a multiplexer). However, we will try to make a general translation, which works for all possible `case` expressions. Optimizations such as these are left for the VHDL synthesizer, which handles them very well.

Top level binders and functions

A *top level binder* is any binder (variable) that is declared in the 'global' scope of a Haskell program (as opposed to a binder that is bound inside a function. The binder together with its body is referred to as a *top level binding*.
In Haskell, there is no sharp distinction between a variable and a function: a function is just a variable (binder) with a function type. This means that a *top level function* is just any top level binder with a function type. This also means that sometimes top level function will be used when top level binder is really meant.
As an example, consider the following Haskell snippet:

```
foo :: Int -> Int
foo x = inc x
  where
    inc = \y -> y + 1
```

Here, `foo` is a top level binder, whereas `inc` is a function (since it is bound to a lambda extraction, indicated by the backslash) but is not a top level binder or function. Since the type of `foo` is a function type, namely `Int -> Int`, it is also a top level function.

Arguments / results vs. inputs / outputs

Due to the translation chosen for function application, there is a very strong relation between arguments, results, inputs and outputs. For clarity, the former two will always refer to the arguments and results in the functional description (either Haskell or Core). The latter two will refer to input and output ports in the generated VHDL.
Even though these concepts seem to be nearly identical, when stateful functions are introduces we will see arguments and results that will not get translated into input and output ports, making this distinction more important.

A slightly more complex (but very powerful) form of choice is pattern matching. A function can be defined in multiple clauses, where each clause specifies a pattern. When the arguments match the pattern, the corresponding clause will be used.

Example 2.3 also shows an inverter that uses pattern matching. The architecture it describes is of course the same one as the description with a case expression. The general interpretation of pattern matching is also similar to that of `case` expressions: generate hardware for each of the clauses (like each of the clauses of a `case` expression) and connect them to the function output through (a number of nested) multiplexers. These multiplexers are driven by comparators and other logic, that check each pattern in turn.

In these examples we have seen only binary case expressions and pattern matches (*i.e.*, with two alternatives). In practice, case expressions can choose between more than two values, resulting in a number of nested multiplexers.

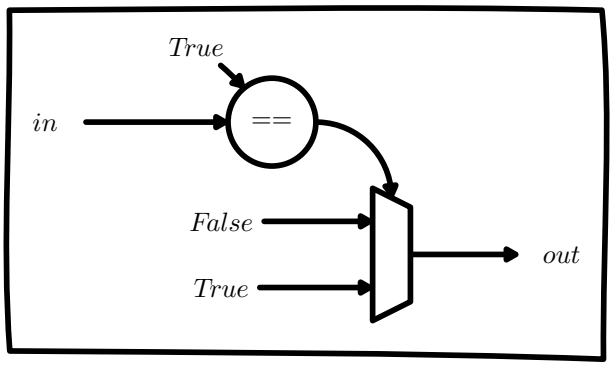
Example 2.3 Simple inverter.

```
inv :: Bool -> Bool
inv x = case x of
  True -> False
  False -> True
```

Haskell description
using a Case expression.

```
inv :: Bool -> Bool
inv True = False
inv False = True
```

Haskell description using
Pattern matching expression.



The architecture described by the Haskell descriptions.

2.3 Types

Translation of two most basic functional concepts has been discussed: function application and choice. Before looking further into less obvious concepts like higher-order expressions and polymorphism, the possible types that can be used in hardware descriptions will be discussed.

Some way is needed to translate every values used to its hardware equivalents. In particular, this means a hardware equivalent for every *type* used in a hardware description is needed

Since most functional languages have a lot of standard types that are hard to translate (integers without a fixed size, lists without a static length, etc.), a number of 'built-in' types will be defined first. These types are built-in in the sense that our compiler will have a fixed VHDL type for these. User defined types, on the other

Example 2.4 VHDL generated for (both versions of) `inv` from example 2.3

```
entity invComponent_0 is
     port (\xzAMo2\ : in boolean;
           \reszAMuzAMu2\ : out boolean;
           clock : in std_logic;
           resetn : in std_logic);
end entity invComponent_0;


architecture structural of invComponent_0 is
begin
     \reszAMuzAMu2\ <= false when \xzAMo2\ = true else
                       true;
end architecture structural;
```

hand, will have their hardware type derived directly from their Haskell declaration automatically, according to the rules sketched here.

2.3.1 Built-in types

The language currently supports the following built-in types. Of these, only the `Bool` type is supported by Haskell out of the box (the others are defined by the CλaSH package, so they are user-defined types from Haskell's point of view).

`Bit` This is the most basic type available. It is mapped directly onto the `std_logic` VHDL type. Mapping this to the `bit` type might make more sense (since the Haskell version only has two values), but using `std_logic` is more standard (and allowed for some experimentation with don't care values)

`Bool` This is the only built-in Haskell type supported and is translated exactly like the Bit type (where a value of `True` corresponds to a value of `High`). Supporting the Bool type is particularly useful to support `if ... then ... else ...` expressions, which always have a `Bool` value for the condition.

A `Bool` is translated to a `std_logic`, just like `Bit`.

`SizedWord`, `SizedInt` These are types to represent integers. A `SizedWord` is unsigned, while a `SizedInt` is signed. These types are parametrized by a length type, so you can define an unsigned word of 32 bits wide as follows:

```
type Word32 = SizedWord D32
```

Here, a type synonym `Word32` is defined that is equal to the `SizedWord` type constructor applied to the type `D32`. `D32` is the *type level representation* of the decimal number 32, making the `Word32` type a 32-bit unsigned word.

These types are translated to the VHDL `unsigned` and `signed` respectively.

`Vector` This is a vector type, that can contain elements of any other type and has a fixed length. It has two type parameters: its length and the type of the elements contained in it. By putting the length parameter in the type, the length of a vector can be determined at compile time, instead of only at run-time for conventional lists.

The `Vector` type constructor takes two type arguments: the length of the vector and the type of the elements contained in it. The state type of an 8 element register bank would then for example be:

```
type RegisterState = Vector D8 Word32
```

Here, a type synonym `RegisterState` is defined that is equal to the `Vector` type constructor applied to the types `D8` (The type level representation of the decimal number 8) and `Word32` (The 32 bit word type as defined above). In other words, the `RegisterState` type is a vector of 8 32-bit words.

A fixed size vector is translated to a VHDL array type.

`RangedWord` This is another type to describe integers, but unlike the previous two it has no specific bit-width, but an upper bound. This means that its range is not limited to powers of two, but can be any number. A `RangedWord` only has an upper bound, its lower bound is implicitly zero. There is a lot of added implementation complexity when adding a lower bound and having just an upper bound was enough for the primary purpose of this type: type-safely indexing vectors.

To define an index for the 8 element vector above, we would do:

```
type RegisterIndex = RangedWord D7
```

Here, a type synonym `RegisterIndex` is defined that is equal to the `RangedWord` type constructor applied to the type `D7`. In other words, this defines an unsigned word with values from 0 to 7 (inclusive). This word can be be used to index the 8 element vector `RegisterState` above.

This type is translated to the `unsigned` VHDL type.

The integer and vector built-in types are discussed in more detail in [1].

2.3.2 User-defined types

There are three ways to define new types in Haskell: algebraic data-types with the `data` keyword, type synonyms with the `type` keyword and type renamings with the `newtype` keyword. GHC offers a few more advanced ways to introduce types (type families, existential typing, GADTs, etc.) which are not standard Haskell. These will be left outside the scope of this research.

Only an algebraic datatype declaration actually introduces a completely new type, for which we provide the VHDL translation below. Type synonyms and renamings only define new names for existing types (where synonyms are completely interchangeable and renamings need explicit conversion). Therefore, these do not need any particular VHDL translation, a synonym or renamed type will just use the same representation as the original type. The distinction between a renaming and a synonym does no longer matter in hardware and can be disregarded in the generated VHDL.

For algebraic types, we can make the following distinction:

Product types A product type is an algebraic datatype with a single constructor with two or more fields, denoted in practice like (a,b), (a,b,c), etc. This is essentially a way to pack a few values together in a record-like structure. In fact, the built-in tuple types are just algebraic product types (and are thus supported in exactly the same way).

The 'product' in its name refers to the collection of values belonging to this type. The collection for a product type is the Cartesian product of the collections for the types of its fields.

These types are translated to VHDL record types, with one field for every field in the constructor. This translation applies to all single constructor algebraic data-types, including those with just one field (which are technically not a product, but generate a VHDL record for implementation simplicity).

Enumerated types An enumerated type is an algebraic datatype with multiple constructors, but none of them have fields. This is essentially a way to get an enumeration-like type containing alternatives.

Note that Haskell's `Bool` type is also defined as an enumeration type, but we have a fixed translation for that.

These types are translated to VHDL enumerations, with one value for each constructor. This allows references to these constructors to be translated to the corresponding enumeration value.

Sum types A sum type is an algebraic datatype with multiple constructors, where the constructors have one or more fields. Technically, a type with more than one field per constructor is a sum of products type, but for our purposes this distinction does not really make a difference, so this distinction is note made.

The 'sum' in its name refers again to the collection of values belonging to this type. The collection for a sum type is the union of the the collections for each of the constructors.

Sum types are currently not supported by the prototype, since there is no obvious VHDL alternative. They can easily be emulated, however, as we will see from an example:

```
data Sum = A Bit Word | B Word
```

An obvious way to translate this would be to create an enumeration to distinguish the constructors and then create a big record that contains all the fields of all the constructors. This is the same translation that would result from the following enumeration and product type (using a tuple for clarity):

```
data SumC = A | B
type Sum = (SumC, Bit, Word, Word)
```

Here, the `SumC` type effectively signals which of the latter three fields of the `Sum` type are valid (the first two if `A`, the last one if `B`), all the other ones have no useful value.

An obvious problem with this naive approach is the space usage: the example above generates a fairly big VHDL type. Since we can be sure that the two `Word`s in the `Sum` type will never be valid at the same time, this is a waste of space.

Obviously, duplication detection could be used to reuse a particular field for another constructor, but this would only partially solve the problem. If two fields would be, for example, an array of 8 bits and an 8 bit unsigned word, these are different types and could not be shared. However, in the final hardware, both of these types would simply be 8 bit connections, so we have a 100% size increase by not sharing these.

Another interesting case is that of recursive types. In Haskell, an algebraic datatype can be recursive: any of its field types can be (or contain) the type being defined. The most well-known recursive type is probably the list type, which is defined is:

```
data List t = Empty | Cons t (List t)
```

Note that `Empty` is usually written as `[]` and `Cons` as `:`, but this would make the definition harder to read. This immediately shows the problem with recursive types: what hardware type to allocate here?

If the naive approach for sum types described above would be used, a record would be created where the first field is an enumeration to distinguish `Empty` from `Cons`. Furthermore, two more fields would be added: one with the (VHDL equivalent of) type `t` (assuming this type is actually known at compile time, this should not be a problem) and a second one with type `List t`. The latter one is of course a problem: this is exactly the type that was to be translated in the first place.

The resulting VHDL type will thus become infinitely deep. In other words, there is no way to statically determine how long (deep) the list will be (it could even be infinite).

In general, recursive types can never be properly translated: all recursive types have a potentially infinite value (even though in practice they will have a bounded value, there is no way for the compiler to automatically determine an upper bound on its size).

2.3.3 Partial application

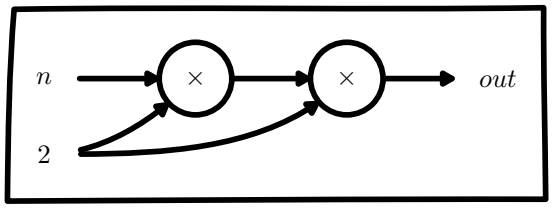
Now the translation of application, choice and types has been discussed, a more complex concept can be considered: partial applications. A *partial application* is any application whose (return) type is (again) a function type.

From this, it should be clear that the translation rules for full application does not apply to a partial application: there are not enough values for all the input ports in the resulting VHDL. Example 2.5 shows an example use of partial application and the corresponding architecture.

Example 2.5 Simple three port and.

```
-- Multiply the input word by four.
quadruple :: Word -> Word
quadruple n = mul (mul n)
  where
    mul = (*) 2
```

Haskell description using function applications.



The architecture described by the Haskell description.

Here, the definition of mul is a partial function application: it applies the function `(*) :: Word -> Word -> Word` to the value `2 :: Word`, resulting in the expression `(*) 2 :: Word -> Word`. Since this resulting expression is again a function, hardware cannot be generated for it directly. This is because the hardware to generate for `mul` depends completely on where and how it is used. In this example, it is even used twice.

However, it is clear that the above hardware description actually describes valid hardware. In general, any partial applied function must eventually become completely applied, at which point hardware for it can be generated using the rules for function application given in section 2.1. It might mean that a partial application is passed around quite a bit (even beyond function boundaries), but eventually, the partial application will become completely applied. An example of this principle is given in section 4.3.6.2.

2.4 Costless specialization

Each (complete) function application in our description generates a component instantiation, or a specific piece of hardware in the final design. It is interesting to note that each application of a function generates a *separate* piece of hardware. In the final design, none of the hardware is shared between applications, even when the applied function is the same (of course, if a particular value, such as the result of a function application, is used twice, it is not calculated twice).

This is distinctly different from normal program compilation: two separate calls to the same function share the same machine code. Having more machine code has implications for speed (due to less efficient caching) and memory usage. For normal compilation, it is therefore important to keep the amount of functions limited and maximize the code sharing (though there is a trade-off between speed and memory usage here).

When generating hardware, this is hardly an issue. Having more 'code sharing' does reduce the amount of VHDL output (Since different component instantiations still share the same component), but after synthesis, the amount of hardware generated is not affected. This means there is no trade-off between speed and memory (or rather, chip area) usage.

In particular, if we would duplicate all functions so that there is a separate function for every application in the program (*e.g.*, each function is then only applied exactly once), there would be no increase in hardware size whatsoever.

Because of this, a common optimization technique called *specialization* can be applied to hardware generation without any performance or area cost (unlike for software).

2.4.1 Specialization

Given some function that has a *domain* D (*e.g.*, the set of all possible arguments that the function could be applied to), we create a specialized function with exactly the same behavior, but with a domain $D' \subset D$. This subset can be chosen in all sorts of ways. Any subset is valid for the general definition of specialization, but in practice only some of them provide useful optimization opportunities.

Common subsets include limiting a polymorphic argument to a single type (*i.e.*, removing polymorphism) or limiting an argument to just a single value (*i.e.*, cross-function constant propagation, effectively removing the argument).

Since we limit the argument domain of the specialized function, its definition can often be optimized further (since now more types or even values of arguments are already known). By replacing any application of the function that falls within the reduced domain by an application of the specialized version, the code gets faster (but the code also gets bigger, since we now have two versions instead of one). If we apply this technique often enough, we can often replace all applications of a function by specialized versions, allowing the original function to be removed (in some cases, this can even give a net reduction of the code compared to the non-specialized version).

Specialization is useful for our hardware descriptions for functions that contain arguments that cannot be translated to hardware directly (polymorphic or higher-order arguments, for example). If we can create

specialized functions that remove the argument, or make it translatable, we can use specialization to make the original, untranslatable, function obsolete.

2.5 Higher order values

What holds for partial application, can be easily generalized to any higher-order expression. This includes partial applications, plain variables (e.g., a binder referring to a top level function), lambda expressions and more complex expressions with a function type (a `case` expression returning lambda's, for example).

Each of these values cannot be directly represented in hardware (just like partial applications). Also, to make them representable, they need to be applied: function variables and partial applications will then eventually become complete applications, applied lambda expressions disappear by applying β-reduction, etc.

So any higher-order value will be 'pushed down' towards its application just like partial applications. Whenever a function boundary needs to be crossed, the called function can be specialized.

2.6 Polymorphism

In Haskell, values can be *polymorphic*: they can have multiple types. For example, the function `fst :: (a, b) -> a` is an example of a polymorphic function: it works for tuples with any two element types. Haskell type classes allow a function to work on a specific set of types, but the general idea is the same. The opposite of this is a *monomorphic* value, which has a single, fixed, type.

When generating hardware, polymorphism cannot be easily translated. How many wires will you lay down for a value that could have any type? When type classes are involved, what hardware components will you lay down for a class method (whose behavior depends on the type of its arguments)? Note that CλaSH currently does not allow user-defined type classes, but does partly support some of the built-in type classes (like `Num`).

Fortunately, we can again use the principle of specialization: since every function application generates a separate piece of hardware, we can know the types of all arguments exactly. Provided that existential typing (which is a GHC extension) is not used typing, all of the polymorphic types in a function must depend on the types of the arguments (In other words, the only way to introduce a type variable is in a lambda abstraction).

If a function is monomorphic, all values inside it are monomorphic as well, so any function that is applied within the function can only be applied to monomorphic values. The applied functions can then be specialized to work just for these specific types, removing the polymorphism from the applied functions as well.

The entry function must not have a polymorphic type (otherwise no hardware interface could be generated for the entry function).

By induction, this means that all functions that are (indirectly) called by our top level function (meaning all functions that are translated in the final hardware) become monomorphic.

2.7 State

A very important concept in hardware designs is *state*. In a stateless (or, *combinational*) design, every output is directly and solely dependent on the inputs. In a stateful design, the outputs can depend on the history of inputs, or the *state*. State is usually stored in *registers*, which retain their value during a clock cycle, and are typically updated at the start of every clock cycle. Since the updating of the state is tightly coupled (synchronized) to the clock signal, these state updates are often called *synchronous* behavior.

To make Cλash useful to describe more than simple combinational designs, it needs to be able to describe state in some way.

2.7.1 Approaches to state

In Haskell, functions are always pure (except when using unsafe functions with the `IO` monad, which is not supported by Cλash). This means that the output of a function solely depends on its inputs. If you evaluate a given function with given inputs, it will always provide the same output.

This is a perfect match for a combinational circuit, where the output also solely depends on the inputs. However, when state is involved, this no longer holds. Of course this purity constraint cannot just be removed from Haskell. But even when designing a completely new (hardware description) language, it does not seem to be a good idea to remove this purity. This would that all kinds of interesting properties of the functional language get lost, and all kinds of transformations and optimizations are no longer be meaning preserving.

So our functions must remain pure, meaning the current state has to be present in the function's arguments in some way. There seem to be two obvious ways to do this: adding the current state as an argument, or including the full history of each argument.

Purity

A function is said to be pure if it satisfies two conditions:

I. When a pure function is called with the same arguments twice, it should return the same value in both cases.

II. When a pure function is called, it should have not observable side-effects.

Purity is an important property in functional languages, since it enables all kinds of mathematical reasoning and optimizations with pure functions, that are not guaranteed to be correct for impure functions.
An example of a pure function is the square root function `sqrt`. An example of an impure function is the `today` function that returns the current date (which of course cannot exist like this in Haskell).

2.7.1.1 Stream arguments and results

Including the entire history of each input (*e.g.*, the value of that input for each previous clock cycle) is an obvious way to make outputs depend on all previous input. This is easily done by making every input a list instead of a single value, containing all previous values as well as the current value.

An obvious downside of this solution is that on each cycle, all the previous cycles must be resimulated to obtain the current state. To do this, it might be needed to have a recursive helper function as well, which might be hard to be properly analyzed by the compiler.

A slight variation on this approach is one taken by some of the other functional HDLs in the field: Make functions operate on complete streams. This means that a function is no longer called on every cycle, but just once. It takes stream as inputs instead of values, where each stream contains all the values for every clock cycle since system start. This is easily modeled using an (infinite) list, with one element for each clock cycle. Since the function is only evaluated once, its output must also be a stream. Note that, since we are working with infinite lists and still want to be able to simulate the system cycle-by-cycle, this relies heavily on the lazy semantics of Haskell.

Since our inputs and outputs are streams, all other (intermediate) values must be streams. All of our primitive operators (*e.g.*, addition, subtraction, bit-wise operations, etc.) must operate on streams as well (note that changing a single-element operation to a stream operation can done with `map`, `zipwith`, etc.).

This also means that primitive operations from an existing language such as Haskell cannot be used directly (including some syntax constructs, like `case` expressions and `if` expressions). This makes this approach well suited for use in EDSLs, since those already impose these same limitations. EDSL p.13

Note that the concept of *state* is no more than having some way to communicate a value from one cycle to the next. By introducing a `delay` function, we can do exactly that: delay (each value in) a stream so that we

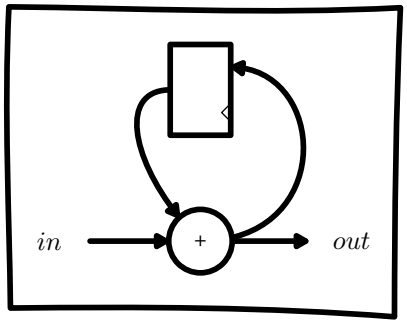
can "look into" the past. This `delay` function simply outputs a stream where each value is the same as the input value, but shifted one cycle. This causes a 'gap' at the beginning of the stream: what is the value of the delay output in the first cycle? For this, the `delay` function has a second input, of which only a single value is used.

Example 2.6 shows a simple accumulator expressed in this style.

Example 2.6 Simple accumulator architecture.

```
acc :: Stream Word -> Stream Word
acc in = out
  where
    out = (delay out 0) + in
```

Haskell description using streams.



The architecture described by the Haskell description.

This notation can be confusing (especially due to the loop in the definition of out), but is essentially easy to interpret. There is a single call to delay, resulting in a circuit with a single register, whose input is connected to `out` (which is the output of the adder), and its output is the expression `delay out 0` (which is connected to one of the adder inputs).

2.7.1.2 Explicit state arguments and results

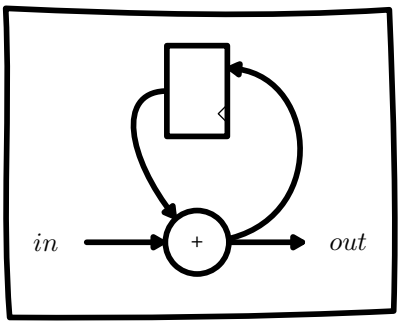
A more explicit way to model state, is to simply add an extra argument containing the current state value. This allows an output to depend on both the inputs as well as the current state while keeping the function pure (letting the result depend only on the arguments), since the current state is now an argument.

In Haskell, this would look like example 2.7[1].

Example 2.7 Simple accumulator architecture.

```
-- input -> current state -> (new state, output)
acc :: Word -> Word -> (Word, Word)
acc in s = (s', out)
  where
    out = s + in
    s'  = out
```

Haskell description using explicit state arguments.



The architecture described by the Haskell description.

[1] *This example is not in the final Cλash syntax*

This approach makes a function's state very explicit, which state variables are used by a function can be completely determined from its type signature (as opposed to the stream approach, where a function looks the same from the outside, regardless of what state variables it uses or whether it is stateful at all).

This approach to state has been one of the initial drives behind this research. Unlike a stream based approach it is well suited to completely use existing code and language features (like `if` and `case` expressions) because it operates on normal values. Because of these reasons, this is the approach chosen for Cλash. It will be examined more closely below.

2.7.2 Explicit state specification

The concept of explicit state has been introduced with some examples above, but what are the implications of this approach?

2.7.2.1 Substates

Since a function's state is reflected directly in its type signature, if a function calls other stateful functions (*e.g.*, has sub-circuits), it has to somehow know the current state for these called functions. The only way to do this, is to put these *substates* inside the caller's state. This means that a function's state is the sum of the states of all functions it calls, and its own state. This sum can be obtained using something simple like a tuple, or possibly custom algebraic types for clarity.

This also means that the type of a function (at least the "state" part) is dependent on its own implementation and of the functions it calls.

This is the major downside of this approach: the separation between interface and implementation is limited. However, since Cλash is not very suitable for separate compilation this is not a big problem in practice.

Additionally, when using a type synonym for the state type of each function, we can still provide explicit type signatures while keeping the state specification for a function near its definition only.

2.7.2.2 Which arguments and results are stateful?

We need some way to know which arguments should become input ports and which argument(s?) should become the current state (*e.g.*, be bound to the register outputs). This does not hold just for the top level function, but also for any sub-function. Or could we perhaps deduce the statefulness of sub-functions by analyzing the flow of data in the calling functions?

To explore this matter, the following observation is interesting: we get completely correct behavior when we put all state registers in the top level entity (or even outside of it). All of the state arguments and results on sub-functions are treated as normal input and output ports. Effectively, a stateful function results in a stateless hardware component that has one of its input ports connected to the output of a register and one of its output ports connected to the input of the same register.

Of course, even though the hardware described like this has the correct behavior, unless the layout tool does smart optimizations, there will be a lot of extra wire in the design (since registers will not be close to the component that uses them). Also, when working with the generated VHDL code, there will be a lot of extra ports just to pass on state values, which can get quite confusing.

To fix this, we can simply 'push' the registers down into the sub-circuits. When we see a register that is connected directly to a sub-circuit, we remove the corresponding input and output port and put the register

inside the sub-circuit instead. This is slightly less trivial when looking at the Haskell code instead of the resulting circuit, but the idea is still the same.

However, when applying this technique, we might push registers down too far. When you intend to store a result of a stateless sub-function in the caller's state and pass the current value of that state variable to that same function, the register might get pushed down too far. It is impossible to distinguish this case from similar code where the called function is in fact stateful. From this we can conclude that we have to either:

- accept that the generated hardware might not be exactly what we intended, in some specific cases. In most cases, the hardware will be what we intended.
- explicitly annotate state arguments and results in the input description.

The first option causes (non-obvious) exceptions in the language interpretation. Also, automatically determining where registers should end up is easier to implement correctly with explicit annotations, so for these reasons we will look at how this annotations could work.

2.7.3 Explicit state annotation

To make our stateful descriptions unambiguous and easier to translate, we need some way for the developer to describe which arguments and results are intended to become stateful.

Roughly, we have two ways to achieve this:

I. Use some kind of annotation method or syntactic construction in the language to indicate exactly which argument and (part of the) result is stateful. This means that the annotation lives 'outside' of the function, it is completely invisible when looking at the function body.

II. Use some kind of annotation on the type level, *i.e.* give stateful arguments and stateful (parts of) results a different type. This has the potential to make this annotation visible inside the function as well, such that when looking at a value inside the function body you can tell if it is stateful by looking at its type. This could possibly make the translation process a lot easier, since less analysis of the program flow might be required.

From these approaches, the type level 'annotations' have been implemented in CλaSH. Section 3.6 expands on the possible ways this could have been implemented.

2.8 Recursion

An important concept in functional languages is recursion. In its most basic form, recursion is a definition that is described in terms of itself. A recursive function is thus a function that uses itself in its body. This usually requires multiple evaluations of this function, with changing arguments, until eventually an evaluation of the function no longer requires itself.

Given the notion that each function application will translate to a component instantiation, we are presented with a problem. A recursive function would translate to a component that contains itself. Or, more precisely, that contains an instance of itself. This instance would again contain an instance of itself, and again, into infinity. This is obviously a problem for generating hardware.

This is expected for functions that describe infinite recursion. In that case, we cannot generate hardware that shows correct behavior in a single cycle (at best, we could generate hardware that needs an infinite number of cycles to complete).

However, most recursive definitions will describe finite recursion. This is because the recursive call is done conditionally. There is usually a `case` expression where at least one alternative does not contain the recursive call, which we call the "base case". If, for each call to the recursive function, we would be able to detect at compile time which alternative applies, we would be able to remove the `case` expression and leave only the base case when it applies. This will ensure that expanding the recursive functions will terminate after a bounded number of expansions.

This does imply the extra requirement that the base case is detectable at compile time. In particular, this means that the decision between the base case and the recursive case must not depend on run-time data.

`null`, `head` and `tail`

The functions `null`, `head` and `tail` are common list functions in Haskell. The `null` function simply checks if a list is empty. The `head` function returns the first element of a list. The `tail` function returns containing everything *except* the first element of a list.
In CλaSH, there are vector versions of these functions, which do exactly the same.

2.8.1 List recursion

The most common deciding factor in recursion is the length of a list that is passed in as an argument. Since we represent lists as vectors that encode the length in the vector type, it seems easy to determine the base case. We can simply look at the argument type for this. However, it turns out that this is rather non-trivial to write down in Haskell already, not even looking at translation. As an example, we would like to write down something like this:

```
sum :: Vector n Word -> Word
sum xs = case null xs of
  True -> 0
  False -> head xs + sum (tail xs)
```

However, the Haskell type-checker will now use the following reasoning. For simplicity, the element type of a vector is left out, all vectors are assumed to have the same element type. Below, we write conditions on type variables before the `=>` operator. This is not completely valid Haskell syntax, but serves to illustrate the type-checker reasoning. Also note that a vector can never have a negative length, so `Vector n` implicitly means `(n >= 0) => Vector n`.

- tail has the type `(n > 0) => Vector n -> Vector (n - 1)`
- This means that xs must have the type `(n > 0) => Vector n`
- This means that sum must have the type `(n > 0) => Vector n -> a` (The type `a` is can be anything at this stage, we will not try to finds its actual type in this example).
- sum is called with the result of tail as an argument, which has the type `Vector n` (since `(n > 0) => Vector (n - 1)` is the same as `(n >= 0) => Vector n`, which is the same as just `Vector n`).

- This means that sum must have the type `Vector n -> a`
- This is a contradiction between the type deduced from the body of sum (the input vector must be non-empty) and the use of sum (the input vector could have any length).

As you can see, using a simple `case` expression at value level causes the type checker to always type-check both alternatives, which cannot be done. The type-checker is unable to distinguish the two case alternatives (this is partly possible using GADTs, but that approach faced other problems [1]).

This is a fundamental problem, that would seem perfectly suited for a type class. Considering that we need to switch between to implementations of the sum function, based on the type of the argument, this sounds like the perfect problem to solve with a type class. However, this approach has its own problems (not the least of them that you need to define a new type class for every recursive function you want to define).

2.8.2 General recursion

Of course there are other forms of recursion, that do not depend on the length (and thus type) of a list. For example, simple recursion using a counter could be expressed, but only translated to hardware for a fixed number of iterations. Also, this would require extensive support for compile time simplification (constant propagation) and compile time evaluation (evaluation of constant comparisons), to ensure non-termination. Supporting general recursion will probably require strict conditions on the input descriptions. Even then, it will be hard (if not impossible) to really guarantee termination, since the user (or GHC desugarer) might use some obscure notation that results in a corner case of the simplifier that is not caught and thus non-termination.

Evaluating all possible (and non-possible) ways to add recursion to our descriptions, it seems better to limit the scope of this research to exclude recursion. This allows for focusing on other interesting areas instead. By including (built-in) support for a number of higher-order functions like `map` and `fold`, we can still express most of the things we would use (list) recursion for.

3 Prototype

An important step in this research is the creation of a prototype compiler. Having this prototype allows us to apply the ideas from the previous chapter to actual hardware descriptions and evaluate their usefulness. Having a prototype also helps to find new techniques and test possible interpretations.

Obviously the prototype was not created after all research ideas were formed, but its implementation has been interleaved with the research itself. Also, the prototype described here is the final version, it has gone through a number of design iterations which we will not completely describe here.

3.1 Input language

When implementing this prototype, the first question to ask is: Which (functional) language will be used to describe our hardware? (Note that this does not concern the *implementation language* of the compiler, just the language *translated by* the compiler).

Initially, we have two choices:

- Create a new functional language from scratch. This has the advantage of having a language that contains exactly those elements that are convenient for describing hardware and can contain special constructs that allows our hardware descriptions to be more powerful or concise.

- Use an existing language and create a new back-end for it. This has the advantage that existing tools can be reused, which will speed up development.

Considering that we required a prototype which should be working quickly, and that implementing parsers, semantic checkers and especially type-checkers is not exactly the core of this research (but it is lots and lots of work, using an existing language is the obvious choice. This also has the advantage that a large set of language features is available to experiment with and it is easy to find which features apply well and which do not. Another important advantage of using an existing language, is that simulation of the code becomes trivial. Since there are existing compilers and interpreters that can run the hardware description directly, it can be simulated without also having to write an interpreter for the new language.

> **No EDSL or Template Haskell**
>
> Note that in this consideration, embedded domain-specific languages (EDSL) and Template Haskell (TH) approaches have not been included. As we have seen in section 1.2, these approaches have all kinds of limitations on the description language that we would like to avoid.

A possible second prototype could use a custom language with just the useful features (and possibly extra features that are specific to the domain of hardware description as well).

The second choice to be made is which of the many existing languages to use. As mentioned before, the chosen language is Haskell. This choice has not been the result of a thorough comparison of languages, for the simple reason that the requirements on the language were completely unclear at the start of this research. The fact that Haskell is a language with a broad spectrum of features, that it is commonly used in research projects and that the primary compiler, GHC, provides a high level API to its internals, made Haskell an obvious choice.

3.2 Output format

The second important question is: what will be our output format? This output format should at least allow for programming the hardware design into a field-programmable gate array (FPGA). The choice of output format is thus limited by what hardware synthesis and programming tools can process.

Looking at other tools in the industry, the Electronic Design Interchange Format (EDIF) is commonly used for storing intermediate *netlists* (lists of components and connections between these components) and is commonly the target for VHDL and Verilog compilers.

However, EDIF is not completely tool-independent. It specifies a meta-format, but the hardware components that can be used vary between various tool and hardware vendors, as well as the interpretation of the EDIF standard. [9]

This means that when working with EDIF, our prototype would become technology dependent (*e.g.* only work with FPGAs of a specific vendor, or even only with specific chips). This limits the applicability of our prototype. Also, the tools we would like to use for verifying, simulating and draw pretty pictures of our output (like Precision, or QuestaSim) are designed for VHDL or Verilog input.

For these reasons, we will not use EDIF, but VHDL as our output language. We choose VHDL over Verilog simply because we are familiar with VHDL already. The differences between VHDL and Verilog are on the higher level, while we will be using VHDL mainly to write low level, netlist-like descriptions anyway.

An added advantage of using VHDL is that we can profit from existing optimizations in VHDL synthesizers. A lot of optimizations are done on the VHDL level by existing tools. These tools have been under development for years, so it would not be reasonable to assume we could achieve a similar amount of optimization in our prototype (nor should it be a goal, considering this is just a prototype).

Note that we will be using VHDL as our output language, but will not use its full expressive power. Our output will be limited to using simple, structural descriptions, without any complex behavioral descriptions like arbitrary sequential statements (which might not be supported by all tools). This ensures that any tool that works with VHDL will understand our output. This also leaves open the option to switch to EDIF in the future, with minimal changes to the prototype.

Translation vs. compilation vs. synthesis

In this thesis the words *translation*, *compilation* and sometimes *synthesis* will be used interchangeably to refer to the process of translating the hardware description from the Haskell language to the VHDL language.
Similarly, the prototype created is referred to as both the *translator* as well as the *compiler*.
The final part of this process is usually referred to as VHDL *generation*.

3.3 Simulation and synthesis

As mentioned above, by using the Haskell language, we get simulation of our hardware descriptions almost for free. The only thing that is needed is to provide a Haskell implementation of all built-in functions that can be used by the Haskell interpreter to simulate them.

The main topic of this thesis is therefore the path from the Haskell hardware descriptions to FPGA synthesis, focusing of course on the VHDL generation. Since the VHDL generation process preserves the meaning of the Haskell description exactly, any simulation done in Haskell *should* produce identical results as the synthesized hardware.

3.4 Prototype design

As suggested above, we will use the Glasgow Haskell Compiler (GHC) to implement our prototype compiler. To understand the design of the prototype, we will first dive into the GHC compiler a bit. Its compilatprototype consists of the following steps (slightly simplified):

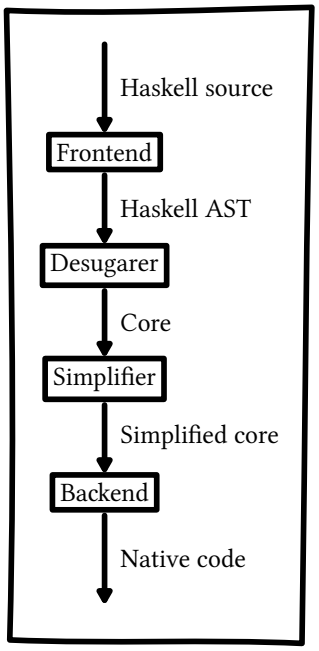
Frontend This step takes the Haskell source files and parses them into an abstract syntax tree (AST). This AST can express the complete Haskell language and is thus a very complex one (in contrast with the Core AST, later on). All identifiers in this AST are resolved by the renamer and all types are checked by the type-checker.

Desugaring This step takes the full AST and translates it to the *Core* language. Core is a very small functional language with lazy semantics, that can still express everything Haskell can express. Its simpleness makes Core very suitable for further simplification and translation. Core is the language we will be working with as well.

Simplification Through a number of simplification steps (such as inlining, common sub-expression elimination, etc.) the Core program is simplified to make it faster or easier to process further.

Backend This step takes the simplified Core program and generates an actual runnable program for it. This is a big and complicated step we will not discuss it any further, since it is not relevant to our prototype.

Figure 3.1 GHC compiler pipeline



In this process, there are a number of places where we can start our work. Assuming that we do not want to deal with (or modify) parsing, type-checking and other front end business and that native code is not really a useful format anymore, we are left with the choice between the full Haskell AST, or the smaller (simplified) Core representation.

The advantage of taking the full AST is that the exact structure of the source program is preserved. We can see exactly what the hardware description looks like and which syntax constructs were used. However, the full AST is a very complicated data-structure. If we are to handle everything it offers, we will quickly get a big compiler.

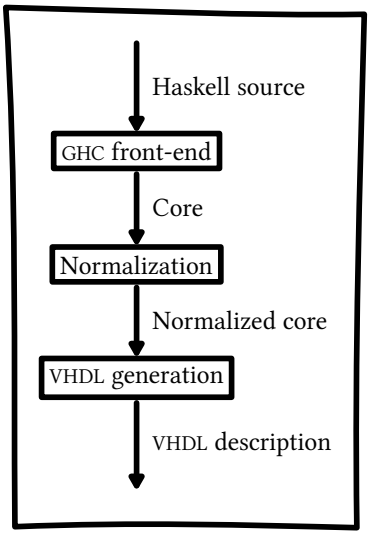
Using the Core representation gives us a much more compact data-structure (a Core expression only uses 9 constructors). Note that this does not mean that the Core representation itself is smaller, on the contrary. Since the Core language has less constructs, most Core expressions are larger than the equivalent versions in Haskell.

However, the fact that the Core language is so much smaller, means it is a lot easier to analyze and translate it into something else. For the same reason, GHC runs its simplifications and optimizations on the Core representation as well [10].

We will use the normal Core representation, not the simplified Core. Even though the simplified Core version is an equivalent, but simpler definition, some problems were encountered with it in practice. The simplifier restructures some (stateful) functions in a way the normalizer and the VHDL generation cannot handle, leading to uncompilable programs (whereas the non-simplified version more closely resembles the original program, allowing the original to be written in a way that can be handled). This problem is further discussed in section 4.4.4.

The final prototype roughly consists of three steps:

Figure 3.2 Cλash compiler pipeline



Frontend This is exactly the front-end from the GHC pipeline, that translates Haskell sources to a typed Core representation.

Normalization This is a step that transforms the Core representation into a normal form. This normal form is still expressed in the Core language, but has to adhere to an additional set of constraints. This normal form is less expressive than the full Core language (e.g., it can have limited higher-order expressions, has a specific structure, etc.), but is also very close to directly describing hardware.

VHDL **generation** The last step takes the normal formed Core representation and generates VHDL for it. Since the normal form has a specific, hardware-like structure, this final step is very straightforward.

The most interesting step in this process is the normalization step. That is where more complicated functional constructs, which have no direct hardware interpretation, are removed and translated into hardware constructs. This step is described in a lot of detail at chapter 4.

Translation of a hardware description always starts at a single function, which is referred to as the *entry function*. VHDL is generated for this function first, followed by any functions used by the entry functions (recursively).

3.5 The Core language

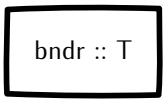
Most of the prototype deals with handling the program in the Core language. In this section we will show what this language looks like and how it works.

The Core language is a functional language that describes *expressions*. Every identifier used in Core is called a *binder*, since it is bound to a value somewhere. On the highest level, a Core program is a collection of functions, each of which bind a binder (the function name) to an expression (the function value, which has a function type).

The Core language itself does not prescribe any program structure (like modules, declarations, imports, etc.), only expression structure. In the GHC compiler, the Haskell module structure is used for the resulting Core code as well. Since this is not so relevant for understanding the Core language or the Normalization process, we will only look at the Core expression language here.

Each Core expression consists of one of these possible expressions.

Variable reference



This is a reference to a binder. It is written down as the name of the binder that is being referred to along with its type. The binder name should of course be bound in a containing scope (including top level

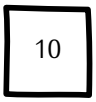
scope, so a reference to a top level function is also a variable reference). Additionally, constructors from algebraic data-types also become variable references (*e.g.* `True`).

In our examples, binders will commonly consist of a single characters, but they can have any length.

The value of this expression is the value bound to the given binder.

Each binder also carries around its type (explicitly shown above), but this is usually not shown in the Core expressions. Only when the type is relevant (when a new binder is introduced, for example) will it be shown. In other cases, the type is either not relevant, or easily derived from the context of the expression.

Literal



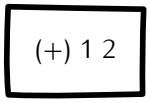
This is a literal. Only primitive types are supported, like chars, strings, integers and doubles. The types of these literals are the 'primitive', unboxed versions, like `Char#` and `Word#`, not the normal Haskell versions (but there are built-in conversion functions). Without going into detail about these types, note that a few conversion functions exist to convert these to the normal (boxed) Haskell equivalents. See section 4.3.6.3 for an example.

Application

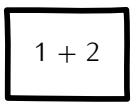
This is function application. Each application consists of two parts: the function part and the argument part. Applications are used for normal function 'calls', but also for applying type abstractions and data constructors.

The value of an application is the value of the function part, with the first argument binder bound to the argument part.

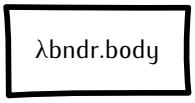
In Core, there is no distinction between an operator and a function. This means that, for example the addition of two numbers looks like the following in Core:



Where the function '(+)' is applied to the numbers 1 and 2. However, to increase readability, an application of an operator like (+) is sometimes written infix. In this case, the parenthesis are also left out, just like in Haskell. In other words, the following means exactly the same as the addition above:



Lambda abstraction

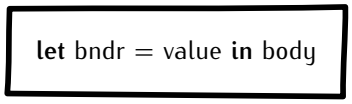


This is the basic lambda abstraction, as it occurs in lambda calculus. It consists of a binder part and a body part. A lambda abstraction creates a function, that can be applied to an argument. The binder is usually a value binder, but it can also be a *type binder* (or *type variable*). The latter case introduces a new polymorphic variable, which can be used in types later on. See section 3.5.1 for details.

The body of a lambda abstraction extends all the way to the end of the expression, or the closing bracket surrounding the lambda. In other words, the lambda abstraction 'operator' has the lowest priority of all.

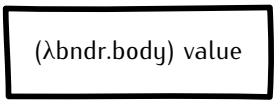
The value of an application is the value of the body part, with the binder bound to the value the entire lambda abstraction is applied to.

Non-recursive let expression



A let expression allows you to bind a binder to some value, while evaluating to some other value (for which that binder is in scope). This allows for sharing of sub-expressions (you can use a binder twice) and explicit 'naming' of arbitrary expressions. A binder is not in scope in the value bound it is bound to, so it is not possible to make recursive definitions with a non-recursive let expression (see the recursive form below).

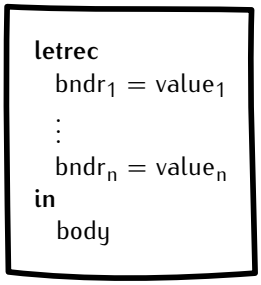
Even though this let expression is an extension on the basic lambda calculus, it is easily translated to a lambda abstraction. The let expression above would then become:



This notion might be useful for verifying certain properties on transformations, since a lot of verification work has been done on lambda calculus already.

The value of a let expression is the value of the body part, with the binder bound to the value.

Recursive let expression



This is the recursive version of the let expression. In GHC's Core implementation, non-recursive and recursive lets are not so distinct as we present them here, but this provides a clearer overview.

The main difference with the normal let expression is that it can contain multiple bindings (or even none) and each of the binders is in scope in each of the values, in addition to the body. This allows for self-recursive or mutually recursive definitions.

It is also possible to express a recursive let expression using normal lambda calculus, if we use the *least fixed-point operator*, Y (but the details are too complicated to help clarify the let expression, so this will not be explored further).

Case expression

case scrutinee **of** bndr
  DEFAULT → defaultbody
  C_0 $bndr_{0,0}$ ... $bndr_{0,m}$ → $body_0$
  ⋮
  C_n $bndr_{n,0}$ ... $bndr_{n,m}$ → $body_n$

A case expression is the only way in Core to choose between values. All `if` expressions and pattern matchings from the original Haskell program have been translated to case expressions by the desugarer.

Weak head normal form (WHNF)

An expression is in weak head normal form if it is either an constructor application or lambda abstraction. [11]
Without going into detail about the differences with head normal form and normal form, note that evaluating the scrutinee of a case expression to normal form (evaluating any function applications, variable references and case expressions) is sufficient to decide which case alternatives should be chosen.

A case expression evaluates its scrutinee, which should have an algebraic datatype, into weak head normal form (WHNF) and (optionally) binds it to bndr. If bndr is wild, it is left out. Every alternative lists a single constructor (C_0 ... C_n). Based on the actual constructor of the scrutinee, the corresponding alternative is chosen. The binders in the chosen alternative ($bndr_{0,0}$ .... $bndr_{0,m}$ are bound to the actual arguments to the constructor in the scrutinee.

wild binders p.77

This is best illustrated with an example. Assume there is an algebraic datatype declared as follows[1]:

```
data D = A Word | B Bit
```

This is an algebraic datatype with two constructors, each getting a single argument. A case expression scrutinizing this datatype could look like the following:

case s **of**
  A word → High
  B bit → bit

[1] *This datatype is not supported by the current Cλash implementation, but serves well to illustrate the case expression*

What this expression does is check the constructor of the scrutinee `s`. If it is `A`, it always evaluates to `High`. If the constructor is `B`, the binder `bit` is bound to the argument passed to `B` and the case expression evaluates to this bit.

If none of the alternatives match, the `DEFAULT` alternative is chosen. A case expression must always be exhaustive, *i.e.* it must cover all possible constructors that the scrutinee can have (if all of them are covered explicitly, the `DEFAULT` alternative can be left out).

Since we can only match the top level constructor, there can be no overlap in the alternatives and thus order of alternatives is not relevant (though the `DEFAULT` alternative must appear first for implementation efficiency).

To support strictness, the scrutinee is always evaluated into WHNF, even when there is only a `DEFAULT` alternative. This allows application of the strict function `f` to the argument `a` to be written like:

```
f (case a of arg DEFAULT → arg)
```

According to the GHC documentation, this is the only use for the extra binder to which the scrutinee is bound. When not using strictness annotations (which is rather pointless in hardware descriptions), GHC seems to never generate any code making use of this binder. In fact, GHC has never been observed to generate code using this binder, even when strictness was involved. Nonetheless, the prototype handles this binder as expected.

Note that these case expressions are less powerful than the full Haskell case expressions. In particular, they do not support complex patterns like in Haskell. Only the constructor of an expression can be matched, complex patterns are implemented using multiple nested case expressions.

Case expressions are also used for unpacking of algebraic data-types, even when there is only a single constructor. For examples, to add the elements of a tuple, the following Core is generated:

```
sum = λtuple.case tuple of
  (,) a b → a + b
```

Here, there is only a single alternative (but no `DEFAULT` alternative, since the single alternative is already exhaustive). When its body is evaluated, the arguments to the tuple constructor `(,)` (*e.g.*, the elements of the tuple) are bound to `a` and `b`.

Cast expression

```
body ▸ targettype
```

A cast expression allows you to change the type of an expression to an equivalent type. Note that this is not meant to do any actual work, like conversion of data from one format to another, or force a complete type change. Instead, it is meant to change between different representations of the same type, *e.g.* switch between types that are provably equal (but look different).

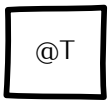
In our hardware descriptions, we typically see casts to change between a Haskell newtype and its contained type, since those are effectively different types (so a cast is needed) with the same representation (but no work is done by the cast).

More complex are types that are proven to be equal by the type-checker, but look different at first glance. To ensure that, once the type-checker has proven equality, this information sticks around, explicit casts are added. In our notation we only write the target type, but in reality a cast expressions carries around a *coercion*, which can be seen as a proof of equality.

The value of a cast is the value of its body, unchanged. The type of this value is equal to the target type, not the type of its body.

Note The Core language in GHC allows adding *notes*, which serve as hints to the inliner or add custom (string) annotations to a Core expression. These should not be generated normally, so these are not handled in any way in the prototype.

Type



It is possibly to use a Core type as a Core expression. To prevent confusion between types and values, the @ sign is used to explicitly mark a type that is used in a Core expression.

For the actual types supported by Core, see section 3.5.1. This 'lifting' of a type into the value domain is done to allow for type abstractions and applications to be handled as normal lambda abstractions and applications above. This means that a type expression in Core can only ever occur in the argument position of an application, and only if the type of the function that is applied to expects a type as the first argument. This happens in applications of all polymorphic functions. Consider the fst function:

```
fst :: ∀ t1. ∀ t2. (t1, t2) →t1
fst = λt1.λt2.λ(tup :: (t1, t2)). case tup of (,) a b → a

fstint :: (Int, Int) → Int
fstint = λa.λb.fst @Int @Int a b
```

The type of fst has two universally quantified type variables. When fst is applied in fstint, it is first applied to two types. (which are substituted for t_1 and t_2 in the type of fst, so the actual type of arguments and result of fst can be found: fst @Int @Int :: (Int, Int) → Int).

3.5.1 Core type system

Whereas the expression syntax of Core is very simple, its type system is a bit more complicated. It turns out it is harder to 'desugar' Haskell's complex type system into something more simple. Most of the type system is thus very similar to that of Haskell.

We will slightly limit our view on Core's type system, since the more complicated parts of it are only meant to support Haskell's (or rather, GHC's) type extensions, such as existential types, GADTs, type families and other non-standard Haskell stuff which we do not (plan to) support.

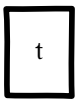
The `id` function

A function that is probably present in every functional language, is the *identity* function. This is the function that takes a single argument and simply returns it unmodified. In Haskell this function is called `id` and can take an argument of any type (*i.e.*, it is polymorphic).
The `id` function will be used in the examples every now and then.

In Core, every expression is typed. The translation to Core happens after the type-checker, so types in Core are always correct as well (though you could of course construct invalidly typed expressions through the GHC API).

Any type in Core is one of the following:

A type variable

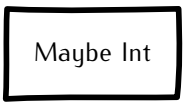


This is a reference to a type defined elsewhere. This can either be a polymorphic type (like the latter two t's in id :: ∀ t. t → t), or a type constructor (like Bool in not :: Bool → Bool). Like in Haskell, polymorphic type variables always start with a lowercase letter, while type constructors always start with an uppercase letter.

A special case of a type constructor is the *function type constructor*, →. This is a type constructor taking two arguments (using application below). The function type constructor is commonly written inline, so we write a → b when we really mean → a b, the function type constructor applied to a and b.

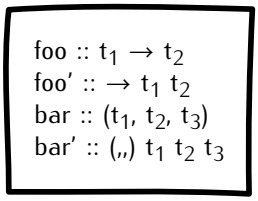
Polymorphic type variables can only be defined by a lambda abstraction, see the forall type below.

A type application

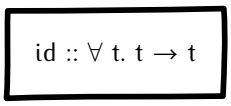


This applies some type to another type. This is particularly used to apply type variables (type constructors) to their arguments.

As mentioned above, applications of some type constructors have special notation. In particular, these are applications of the *function type constructor* and *tuple type constructors*:



The forall type



The forall type introduces polymorphism. It is the only way to introduce new type variables, which are completely unconstrained (Any possible type can be assigned to it). Constraints can be added later using predicate types, see below.

A forall type is always (and only) introduced by a type lambda expression. For example, the Core translation of the id function is:

```
id = λt.λ(x :: t).x
```

Here, the type of the binder x is t, referring to the binder in the topmost lambda.

When using a value with a forall type, the actual type used must be applied first. For example Haskell expression `id True` (the function `id` applied to the data-constructor `True`) translates to the following Core:

```
id @Bool True
```

Here, id is first applied to the type to work with. Note that the type then changes from id :: ∀ t. t → t to id @Bool :: Bool → Bool. Note that the type variable a has been substituted with the actual type.

In Haskell, forall types are usually not explicitly specified (The use of a lowercase type variable implicitly introduces a forall type for that variable). In fact, in standard Haskell there is no way to explicitly specify forall types. Through a language extension, the `forall` keyword is available, but still optional for normal forall types (it is needed for *existentially quantified types*, which CλAsh does not support).

Predicate type

```
show :: ∀ t. Show t ⇒ t → String
```

A predicate type introduces a constraint on a type variable introduced by a forall type (or type lambda). In the example above, the type variable t can only contain types that are an *instance* of the *type class* Show.

There are other sorts of predicate types, used for the type families extension, which we will not discuss here.

A predicate type is introduced by a lambda abstraction. Unlike with the forall type, this is a value lambda abstraction, that must be applied to a value. We call this value a *dictionary*.

Without going into the implementation details, a dictionary can be seen as a lookup table all the methods for a given (single) type class instance. This means that all the dictionaries for the same type class look the same (*e.g.* contain methods with the same names). However, dictionaries for different instances of the same class contain different methods, of course.

A dictionary is introduced by GHC whenever it encounters an instance declaration. This dictionary, as well as the binder introduced by a lambda that introduces a dictionary, have the predicate type as their type. These binders are usually named starting with a $. Usually the name of the type concerned is not reflected in the name of the dictionary, but the name of the type class is. The Haskell expression `show True` thus becomes:

```
show @Bool $dShow True
```

Using this set of types, all types in basic Haskell can be represented.

3.6 State annotations in Haskell

As noted in section 2.7.3, CλaSH needs some way to let the programmer explicitly specify which of a function's arguments and which part of a function's result represent the function's state.
Using the Haskell type systems, there are a few ways we can tackle this.

3.6.1 Type synonyms

Haskell provides type synonyms as a way to declare a new type that is equal to an existing type (or rather, a new name for an existing type). This allows both the original type and the synonym to be used interchangeably in a Haskell program. This means no explicit conversion is needed. For example, a simple accumulator would become:

```
-- This type synonym would become part of CaSH, it is shown here
-- just for clarity.
type State s = s

acc :: Word -> State Word -> (State Word, Word)
acc i s = let sum = s + i in (sum, sum)
```

This looks nice in Haskell, but turns out to be hard to implement. There is no explicit conversion in Haskell, but not in Core either. This means the type of a value might be shown as `State Word` in some places, but `Word` in others (and this can even change due to transformations). Since every binder has an explicit type associated with it, the type of every function type will be properly preserved and could be used to track down the statefulness of each value by the compiler. However, this would make the implementation a lot more complicated than when using type renamings as described in the next section.

3.6.2 Type renaming (`newtype`)

Haskell also supports type renamings as a way to declare a new type that has the same (run-time) representation as an existing type (but is in fact a different type to the type-checker). With type renaming, explicit conversion between values of the two types is needed. The accumulator would then become:

```
-- This type renaming would become part of CaSH, it is shown here
-- just for clarity.
newtype State s = State s

acc :: Word -> State Word -> (State Word, Word)
acc i (State s) = let sum = s + i in (State sum, sum)
```

The `newtype` line declares a new type `State` that has one type argument, `s`. This type contains one 'constructor' `State` with a single argument of type `s`. It is customary to name the constructor the same as the type, which is allowed (since types can never cause name collisions with values). The difference with the type synonym example is in the explicit conversion between the `State Word` and `Word` types by pattern matching and by using the explicit the `State` constructor.

This explicit conversion makes the VHDL generation easier: whenever we remove (unpack) the `State` type, this means we are accessing the current state (*i.e.*, accessing the register output). Whenever we are adding (packing) the `State` type, we are producing a new value for the state (*i.e.*, providing the register input).

When dealing with nested states (a stateful function that calls stateful functions, which might call stateful functions, etc.) the state type could quickly grow complex because of all the `State` type constructors needed. For example, consider the following state type (this is just the state type, not the entire function type):

```
State (State Bit, State (State Word, Bit), Word)
```

We cannot leave all these `State` type constructors out, since that would change the type (unlike when using type synonyms). However, when using type synonyms to hide away sub-states (see section 3.6.3 below), this disadvantage should be limited.

3.6.2.1 Different input and output types

An alternative could be to use different types for input and output state (*i.e.* current and updated state). The accumulator example would then become something like:

```
-- These type renamings would become part of Cash, it is shown
-- here just for clarity.
newtype StateIn s = StateIn s
newtype StateOut s = StateOut s

acc :: Word -> StateIn Word -> (StateIn Word, Word)
acc i (StateIn s) = let sum = s + i in (StateIn sum, sum)
```

This could make the implementation easier and the hardware descriptions less error-prone (you can no longer 'forget' to unpack and repack a state variable and just return it directly, which can be a problem in the current prototype). However, it also means we need twice as many type synonyms to hide away sub-states, making this approach a bit cumbersome. It also makes it harder to compare input and output state types, possible reducing the type-safety of the descriptions.

3.6.3 Type synonyms for sub-states

As noted above, when using nested (hierarchical) states, the state types of the 'upper' functions (those that call other functions, which call other functions, etc.) quickly become complicated. Also, when the state type of one of the 'lower' functions changes, the state types of all the upper functions changes as well. If the state

type for each function is explicitly and completely specified, this means that a lot of code needs updating whenever a state type changes.

To prevent this, it is recommended (but not enforced) to use a type synonym for the state type of every function. Every function calling other functions will then use the state type synonym of the called functions in its own type, requiring no code changes when the state type of a called function changes. This approach is used in example 3.3 below. The `AccState` and `AvgState` are examples of such state type synonyms.

3.6.4 Chosen approach

To keep implementation simple, the current prototype uses the type renaming approach, with a single type for both input and output states. In the future, it might be worthwhile to revisit this approach if more complicated flow analysis is implemented for state variables. This analysis is needed to add proper error checking anyway and might allow the use of type synonyms without losing any expressivity.

3.6.4.1 Example

As an example of the used approach, a simple averaging circuit is shown in example 3.3. This circuit lets the accumulation of the inputs be done by a sub-component, `acc`, but keeps a count of value accumulated in its own state.[1]

3.7 VHDL generation for state

Now its clear how to put state annotations in the Haskell source, there is the question of how to implement this state translation. As we have seen in section 3.4, the translation to VHDL happens as a simple, final step in the compilation process. This step works on a Core expression in normal form. The specifics of normal form will be explained in chapter 4, but the examples given should be easy to understand using the definition of Core given above. The conversion to and from the `State` type is done using the cast operator, `▶`.

3.7.1 State in normal form

Before describing how to translate state from normal form to VHDL, we will first see how state handling looks in normal form. How must their use be limited to guarantee that proper VHDL can be generated?

We will formulate a number of rules about what operations are allowed with state variables. These rules apply to the normalized Core representation, but will in practice apply to the original Haskell hardware description as well. Ideally, these rules would become part of the intended normal form definition , but this is not the case right now. This can cause some problems, which are detailed in section 4.4.4.

intended normal form definition p.56

In these rules we use the terms *state variable* to refer to any variable that has a State type. A *state-containing variable* is any variable whose type contains a State type, but is not one itself (like (AccState, Word) in the example, which is a tuple type, but contains AccState, which is again equal to State Word).

We also use a distinction between *input* and *output (state) variables* and *sub-state variables*, which will be defined in the rules themselves.

[1] *Currently, the prototype is not able to compile this example, since there is no built-in function for division.*

Example 3.3 Simple stateful averaging circuit.

```
-- This type renaming would become part of Cash, it is shown
-- here just for clarity
newtype State s = State s

-- The accumulator state type
type AccState = State Word
-- The accumulator
acc :: Word -> AccState -> (AccState, Word)
acc i (State s) = let sum = s + i in (State sum, sum)

-- The averaging circuit state type
type AvgState = State (AccState, Word)
-- The averaging circuit
avg :: Word -> AvgState -> (AvgState, Word)
avg i (State s) = (State s', o)
  where
    (accs, count) = s
    -- Pass our input through the accumulator, which outputs a sum
    (accs', sum) = acc i accs
    -- Increment the count (which will be our new state)
    count' = count + 1
    -- Compute the average
    o = sum / count'
    s' = (accs', count')
```

These rules describe everything that can be done with state variables and state-containing variables. Everything else is invalid. For every rule, the corresponding part of example 3.4 is shown.

State variables can appear as an argument.

```
avg = λi.λspacked. ...
```

Any lambda that binds a variable with a state type, creates a new input state variable.

Input state variables can be unpacked.

```
s = spacked ▸ (AccState, Word)
```

An input state variable may be unpacked using a cast operation. This removes the State type renaming and the result has no longer a State type.

If the result of this unpacking does not have a state type and does not contain state variables, there are no limitations on its use (this is the function's own state). Otherwise if it does not have a state type but does

Example 3.4 Normalized version of example 3.3

```
acc = λi.λspacked.
  let
    -- Remove the State newtype
    s = spacked ▸ Word
    sum = s + i
    -- Add the State newtype again
    spacked' = sum ▸ State Word
    res = (spacked', sum)
  in
    res

avg = λi.λspacked.
  let
    s = spacked ▸ (AccState, Word)
    accs = case s of (a, b) → a
    count = case s of (c, d) → d
    accres = acc i accs
    accs' = case accres of (e, f) → e
    sum = case accres of (g, h) → h
    count' = count + 1
    o = sum / count'
    s' = (accs', count')
    spacked' = s' ▸ State (AccState, Word)
    res = (spacked', o)
  in
    res
```

contain sub-states, we refer to it as a *state-containing input variable* and the limitations below apply. If it has a state type itself, we refer to it as an *input sub-state variable* and the below limitations apply as well.

It may seem strange to consider a variable that still has a state type directly after unpacking, but consider the case where a function does not have any state of its own, but does call a single stateful function. This means it must have a state argument that contains just a sub-state. The function signature of such a function could look like:

```
type FooState = State AccState
```

Which is of course equivalent to State (State Word).

Variables can be extracted from state-containing input variables.

```
accs = case s of (a, b) → a
```

A state-containing input variable is typically a tuple containing multiple elements (like the current function's state, sub-states or more tuples containing sub-states). All of these can be extracted from an input variable using an extractor case (or possibly multiple, when the input variable is nested).

If the result has no state type and does not contain any state variables either, there are no further limitations on its use (this is the function's own state). If the result has no state type but does contain state variables we refer to it as a *state-containing input variable* and this limitation keeps applying. If the variable has a state type itself, we refer to it as an *input sub-state variable* and below limitations apply.

Input sub-state variables can be passed to functions.

```
accres = acc i accs
accs' = case accres of (e, f) → e
```

An input sub-state variable can (only) be passed to a function. Additionally, every input sub-state variable must be used in exactly *one* application, no more and no less.

The function result should contain exactly one state variable, which can be extracted using (multiple) case expressions. The extracted state variable is referred to the *output sub-state*

The type of this output sub-state must be identical to the type of the input sub-state passed to the function.

Variables can be inserted into a state-containing output variable.

```
s' = (accs', count')
```

A function's output state is usually a tuple containing its own updated state variables and all output sub-states. This result is built up using any single-constructor algebraic datatype (possibly nested).

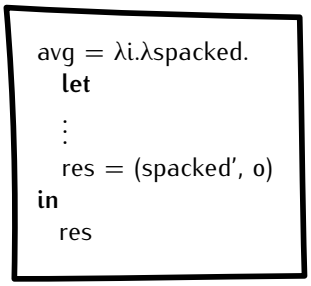
The result of these expressions is referred to as a *state-containing output variable*, which are subject to these limitations.

State containing output variables can be packed.

```
spacked' = s' ▸ State (AccState, Word)
```

As soon as all a functions own update state and output sub-state variables have been joined together, the resulting state-containing output variable can be packed into an output state variable. Packing is done by casting to a state type.

Output state variables can appear as (part of) a function result.



When the output state is packed, it can be returned as a part of the function result. Nothing else can be done with this value (or any value that contains it).

There is one final limitation that is hard to express in the above itemization. Whenever sub-states are extracted from the input state to be passed to functions, the corresponding output sub-states should be inserted into the output state in the same way. In other words, each pair of corresponding sub-states in the input and output states should be passed to / returned from the same called function.

The prototype currently does not check much of the above conditions. This means that if the conditions are violated, sometimes a compile error is generated, but in other cases output can be generated that is not valid VHDL or at the very least does not correspond to the input.

3.7.2 Translating to VHDL

As noted above, the basic approach when generating VHDL for stateful functions is to generate a single register for every stateful function. We look around the normal form to find the let binding that removes the State type renaming (using a cast). We also find the let binding that adds a State type. These are connected to the output and the input of the generated let binding respectively. This means that there can only be one let binding that adds and one that removes the State type. It is easy to violate this constraint. This problem is detailed in section 4.4.4.

This approach seems simple enough, but will this also work for more complex stateful functions involving sub-states? Observe that any component of a function's state that is a sub-state, *i.e.* passed on as the state of another function, should have no influence on the hardware generated for the calling function. Any state-specific VHDL for this component can be generated entirely within the called function. So, we can completely ignore sub-states when generating VHDL for a function.

From this observation it might seem logical to remove the sub-states from a function's states altogether and leave only the state components which are actual states of the current function. While doing this would not remove any information needed to generate VHDL from the function, it would cause the function definition to become invalid (since we will not have any sub-state to pass to the functions anymore). We could solve the syntactic problems by passing `undefined` for state variables, but that would still break the code on the semantic level (*i.e.*, the function would no longer be semantically equivalent to the original input).

To keep the function definition correct until the very end of the process, we will not deal with (sub)states until we get to the VHDL generation. Then, we are translating from Core to VHDL, and we can simply generate no VHDLfor sub-states, effectively removing them altogether.

But, how will we know what exactly is a sub-state? Since any state argument or return value that represents state must be of the `State` type, we can look at the type of a value. However, we must be careful to ignore only *sub-states*, and not a function's own state.

For example 3.4 above, we should generate a register with its output connected to `s` and its input connected to `s'`. However, `s'` is build up from both `accs'` and `count'`, while only `count'` should end up in the register. `accs'` is a sub-state for the `acc` function, for which a register will be created when generating VHDL for the `acc` function.

Fortunately, the `accs'` variable (and any other sub-state) has a property that we can easily check: it has a `State` type. This means that whenever VHDL is generated for a tuple (or other algebraic type), we can simply leave out all elements that have a `State` type. This will leave just the parts of the state that do not have a `State` type themselves, like `count'`, which is exactly a function's own state. This approach also means that the state part of the result (*e.g.* `s'` in `res`) is automatically excluded when generating the output port, which is also required.

We can formalize this translation a bit, using the following rules.

- A state unpack operation should not generate any VHDL. The binder to which the unpacked state is bound should still be declared, this signal will become the register and will hold the current state.

- A state pack operation should not generate any VHDL. The binder to which the packed state is bound should not be declared. The binder that is packed is the signal that will hold the new state.

- Any values of a State type should not be translated to VHDL. In particular, State elements should be removed from tuples (and other data-types) and arguments with a state type should not generate ports.

- To make the state actually work, a simple VHDL (sequential) process should be generated. This process updates the state at every clock cycle, by assigning the new state to the current state. This will be recognized by synthesis tools as a register specification.

When applying these rules to the function `avg` from example 3.4, we be left with the description below. All the parts that do not generate any VHDL directly are crossed out, leaving just the actual flow of values in the final hardware. To illustrate the change of the types of `s` and `s'`, their types are also shown.

```
avg = iλ.λspacked.
  let
    s :: (AccState, Word)
    s = spacked ▸ (AccState, Word)
    accs = case s of (a, b) → a
    count = case s of (c, d) → d
    accres = acc i accs
    accs' = case accres of (e, f) → e
    sum = case accres of (g, h) → h
    count' = count + 1
    o = sum / count'
    s' :: (AccState, Word)
    s' = (accs', count')
    spacked' = s' ▸ State (AccState, Word)
    res = (spacked', o)
  in
    res
```

When we actually leave out the crossed out parts, we get a slightly weird program: there is a variable s which has no value, and there is a variable s′ that is never used. But together, these two will form the state process of the function. s contains the "current" state, s′ is assigned the "next" state. So, at the end of each clock cycle, s′ should be assigned to s.

As an illustration of the result of this function, example 3.6 and example 3.7 show the the VHDL that is generated by Cλash from the examples is this section.

Example 3.5 VHDL types generated for `acc` and `avg` from example 3.3

```
package types is
     subtype \unsigned_31\ is unsigned (0 to 31);

     type \(,)unsigned_31\ is record
           A : \unsigned_31\;
     end record;
end package types;
```

3.8 Prototype implementation

The prototype has been implemented using Haskell as its implementation language, just like GHC. This allows the prototype do directly use parts of GHC through the API it exposes (which essentially taps directly into the internals of GHC, making this API not really a stable interface).

Cλash can be run from a separate library, but has also been integrated into `ghci` [1]. The latter does requires a custom GHC build, however.

The latest version and all history of the CλaSH code can be browsed on-line or retrieved using the `git` program.

http://git.stderr.nl/gitweb?p=matthijs/projects/clash.git

Example 3.6 VHDL generated for `acc` from example 3.3

```
entity accComponent_1 is
     port (\izAob3\ : in \unsigned_31\;
           \foozAoBzAoB2\ : out \(,)unsigned_31\;
           clock : in std_logic;
           resetn : in std_logic);
end entity accComponent_1;

architecture structural of accComponent_1 is
     signal \szAod3\ : \unsigned_31\;
     signal \reszAonzAor3\ : \unsigned_31\;
begin
     \reszAonzAor3\ <= \szAod3\ + \izAob3\;

     \foozAoBzAoB2\.A <= \reszAonzAor3\;

     state : process (clock, resetn)
     begin
          if resetn = '0' then
          elseif rising_edge(clock) then
               \szAod3\ <= \reszAonzAor3\;
          end if;
     end process state;
end architecture structural;
```

Example 3.7 VHDL generated for avg from example 3.3

```
entity avgComponent_0 is
     port (\izAlE2\ : in \unsigned_31\;
           \foozAo1zAo12\ : out \(,)unsigned_31\;
           clock : in std_logic;
           resetn : in std_logic);
end entity avgComponent_0;


architecture structural of avgComponent_0 is
     signal \szAlG2\ : \(,)unsigned_31\;
     signal \countzAlW2\ : \unsigned_31\;
     signal \dszAm62\ : \(,)unsigned_31\;
     signal \sumzAmk3\ : \unsigned_31\;
     signal \reszAnCzAnM2\ : \unsigned_31\;
     signal \foozAnZzAnZ2\ : \unsigned_31\;
     signal \reszAnfzAnj3\ : \unsigned_31\;
     signal \s'zAmC2\ : \(,)unsigned_31\;
begin
     \countzAlW2\ <= \szAlG2\.A;

     \comp_ins_dszAm62\ : entity accComponent_1
                               port map (\izAob3\ => \izAlE2\,
                                         \foozAoBzAoB2\ => \dszAm62\,
                                         clock => clock,
                                         resetn => resetn);

     \sumzAmk3\ <= \dszAm62\.A;

     \reszAnCzAnM2\ <= to_unsigned(1, 32);

     \foozAnZzAnZ2\ <= \countzAlW2\ + \reszAnCzAnM2\;

     \reszAnfzAnj3\ <= \sumzAmk3\ * \foozAnZzAnZ2\;

     \s'zAmC2\.A <= \foozAnZzAnZ2\;

     \foozAo1zAo12\.A <= \reszAnfzAnj3\;

     state : process (clock, resetn)
     begin
          if resetn = '0' then
          elseif rising_edge(clock) then
               \szAlG2\ <= \s'zAmC2\;
          end if;
     end process state;
end architecture structural;
```

4 Normalization

The first step in the Core to VHDL translation process, is normalization. We aim to bring the Core description into a simpler form, which we can subsequently translate into VHDL easily. This normal form is needed because the full Core language is more expressive than VHDL in some areas (higher-order expressions, limited polymorphism using type classes, etc.) and because Core can describe expressions that do not have a direct hardware interpretation.

4.1 Normal form

The transformations described here have a well-defined goal: to bring the program in a well-defined form that is directly translatable to VHDL, while fully preserving the semantics of the program. We refer to this form as the *normal form* of the program. The formal definition of this normal form is quite simple:

Definition 4.1

A program is in *normal form* if none of the transformations from this chapter apply.

Of course, this is an 'easy' definition of the normal form, since our program will end up in normal form automatically. The more interesting part is to see if this normal form actually has the properties we would like it to have.

But, before getting into more definitions and details about this normal form, let us try to get a feeling for it first. The easiest way to do this is by describing the things that are unwanted in the intended normal form.

- Any *polymorphism* must be removed. When laying down hardware, we cannot generate any signals that can have multiple types. All types must be completely known to generate hardware.
- All *higher-order* constructions must be removed. We cannot generate a hardware signal that contains a function, so all values, arguments and return values used must be first order.
- All complex *nested scopes* must be removed. In the VHDL description, every signal is in a single scope. Also, full expressions are not supported everywhere (in particular port maps can only map signal names and constants, not complete expressions). To make the VHDL generation easy, a separate binder must be bound to ever application or other expression.

A very simple example of a program in normal form is given in example 4.2. As you can see, all arguments to the function (which will become input ports in the generated VHDL) are at the outer level. This means that the body of the inner lambda abstraction is never a function, but always a plain value.

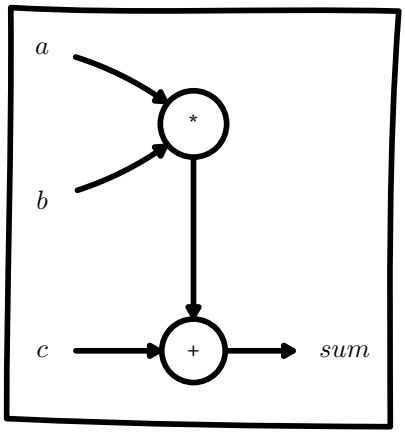
As the body of the inner lambda abstraction, we see a single (recursive) let expression, that binds two variables (mul and sum). These variables will be signals in the generated VHDL, bound to the output port of the $*$ and $+$ components.

The final line (the 'return value' of the function) selects the sum signal to be the output port of the function. This 'return value' can always only be a variable reference, never a more complex expression.

Example 4.2 showed a function that just applied two other functions (multiplication and addition), resulting in a simple architecture with two components and some connections. There is of course also some mechanism for choice in the normal form. In a normal Core program, the *case* expression can be used in a few different ways to describe choice. In normal form, this is limited to a very specific form.

Example 4.2 Simple architecture consisting of a multiplier and a subtractor.

```
alu :: Bit → Word → Word → Word
alu = λa.λb.λc.
    let
      mul = (*) a b
      sum = (+) mul c
    in
      sum
```



Core description in normal form.

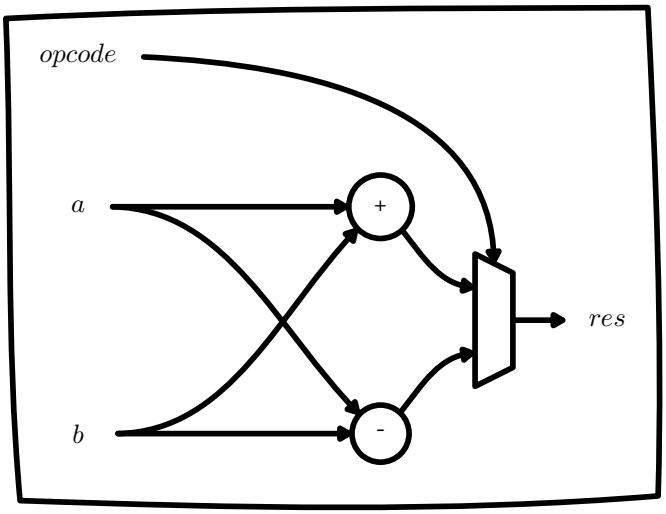
The architecture described by the normal form.

Example 4.3 shows an example describing a simple ALU, which chooses between two operations based on an opcode bit. The main structure is similar to example 4.2, but this time the res variable is bound to a case expression. This case expression scrutinizes the variable opcode (and scrutinizing more complex expressions is not supported). The case expression can select a different variable based on the constructor of opcode.

case expression p.37

Example 4.3 Simple ALU supporting two operations.

```
alu :: Bit → Word → Word → Word
alu = λopcode.λa.λb.
    let
      res1 = (+) a b
      res2 = (-) a b
      res = case opcode of
        Low → res1
        High → res2
    in
      res
```



Core description in normal form.

The architecture described by the normal form.

As a more complete example, consider example 4.4. This example shows everything that is allowed in normal form, except for built-in higher-order functions (like map). The graphical version of the architecture contains a slightly simplified version, since the state tuple packing and unpacking have been left out. Instead, two separate registers are drawn. Most synthesis tools will further optimize this architecture by removing the multiplexers at the register input and instead use the write enable port of the register (when it is available), but we want to show the architecture as close to the description as possible.

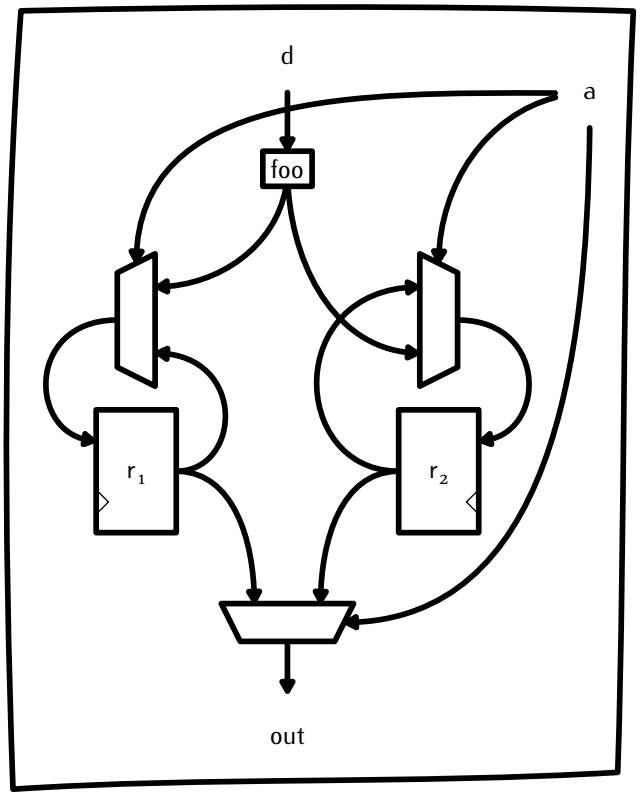
As you can see from the previous examples, the generation of the final architecture from the normal form is straightforward. In each of the examples, there is a direct match between the normal form structure, the generated VHDL and the architecture shown in the images.

Example 4.4 Simple architecture consisting of an adder and a subtractor.

```
regbank :: Bit
        → Word
        → State (Word, Word)
        → (State (Word, Word), Word)

-- All arguments are an initial lambda
-- (address, data, packed state)
regbank = λa.λd.λsp.
-- There are nested let expressions at top level
let
  -- Unpack the state by coercion (e.g., cast from
  -- State (Word, Word) to (Word, Word))
  s = sp ▸ (Word, Word)
  -- Extract both registers from the state
  r1 = case s of (a, b) → a
  r2 = case s of (a, b) → b
  -- Calling some other user-defined function.
  d' = foo d
  -- Conditional connections
  out = case a of
    High → r1
    Low → r2
  r1' = case a of
    High → d'
    Low → r1
  r2' = case a of
    High → r2
    Low → d'
  -- Packing a tuple
  s' = (,) r1' r2'
  -- pack the state by coercion (e.g., cast from
  -- (Word, Word) to State (Word, Word))
  sp' = s' ▸ State (Word, Word)
  -- Pack our return value
  res = (,) sp' out
in
  -- The actual result
  res
```

Core description in normal form.



The architecture described by the normal form.

4.1.1 Intended normal form definition

Now we have some intuition for the normal form, we can describe how we want the normal form to look like in a slightly more formal manner. The EBNF-like description in definition 4.5 captures most of the intended structure (and generates a subset of GHC's Core format).

There are two things missing from this definition: cast expressions are sometimes allowed by the prototype, but not specified here and the below definition allows uses of state that cannot be translated to VHDL properly. These two problems are discussed in section 4.4.3 and section 4.4.4 respectively.

Some clauses have an expression listed behind them in parentheses. These are conditions that need to apply to the clause. The predicates used there (lvar(), representable(), gvar()) will be defined in section 4.2.2.1.

An expression is in normal form if it matches the first definition, *normal*.

Definition 4.5 Definition of the intended normal form using an EBNF-like syntax.

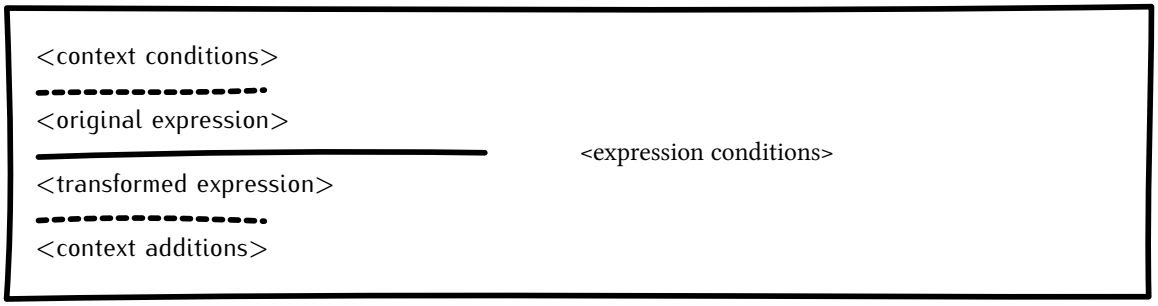
normal := *lambda*
lambda := λvar.*lambda* (representable(var))
| *toplet*
toplet := **letrec** [*binding*...] **in** var (representable(var))
binding := var = *rhs* (representable(rhs))
-- *State packing and unpacking by coercion*
| var_0 = var_1 ▸ State ty ($lvar(var_1)$)
| var_0 = var_1 ▸ ty (var_1 :: State ty ∧ $lvar(var_1)$)
rhs := *userapp*
| *builtinapp*
-- *Extractor* **case**
| **case** var **of** C a_0 ... a_n → a_i (lvar(var))
-- *Selector* **case**
| **case** var **of** (lvar(var))
[DEFAULT → var] (lvar(var))
C_0 $w_{0,0}$ ... $w_{0,n}$ → var_0
⋮
C_m $w_{m,0}$ ... $w_{m,n}$ → var_m (∀i ∀j, $w_{i,j} \neq var_i$, $lvar(var_i)$)
userapp := *userfunc*
| *userapp* userarg
userfunc := var (gvar(var))
userarg := var (lvar(var))
builtinapp := *builtinfunc*
| *builtinapp builtinarg*
built-infunc := var (bvar(var))
built-inarg := var (representable(var) ∧ lvar(var))
| *partapp* (partapp :: a → b)
| *coreexpr* (¬representable(coreexpr) ∧ ¬(coreexpr :: a → b))
partapp := *userapp*
| *builtinapp*

When looking at such a program from a hardware perspective, the top level lambda abstractions (*lambda*) define the input ports. Lambda abstractions cannot appear anywhere else. The variable reference in the body of the recursive let expression (*toplet*) is the output port. Most binders bound by the let expression define a component instantiation (*userapp*), where the input and output ports are mapped to local signals (*userarg*). Some of the others use a built-in construction (*e.g.* the **case** expression) or call a built-in function (*builtinapp*) such as + or map. For these, a hard-coded VHDL translation is available.

4.2 Transformation notation

To be able to concisely present transformations, we use a specific format for them. It is a simple format, similar to one used in logic reasoning.

Such a transformation description looks like the following.



This format describes a transformation that applies to <original expression> and transforms it into <transformed expression>, assuming that all conditions are satisfied. In this format, there are a number of placeholders in pointy brackets, most of which should be rather obvious in their meaning. Nevertheless, we will more precisely specify their meaning below:

<original expression> The expression pattern that will be matched against (sub-expressions of) the expression to be transformed. We call this a pattern, because it can contain *placeholders* (variables), which match any expression or binder. Any such placeholder is said to be *bound* to the expression it matches. It is convention to use an uppercase letter (*e.g.* M or E) to refer to any expression (including a simple variable reference) and lowercase letters (*e.g.* v or b) to refer to (references to) binders.

For example, the pattern a + B will match the expression v + (2 ∗ w) (binding a to v and B to (2 ∗ w)), but not (2 ∗ w) + v.

<expression conditions> These are extra conditions on the expression that is matched. These conditions can be used to further limit the cases in which the transformation applies, commonly to prevent a transformation from causing a loop with itself or another transformation.

Only if these conditions are *all* satisfied, the transformation applies.

<context conditions> These are a number of extra conditions on the context of the function. In particular, these conditions can require some (other) top level function to be present, whose value matches the pattern given here. The format of each of these conditions is: binder = <pattern>.

Typically, the binder is some placeholder bound in the <original expression>, while the pattern contains some placeholders that are used in the transformed expression.

Only if a top level binder exists that matches each binder and pattern, the transformation applies.

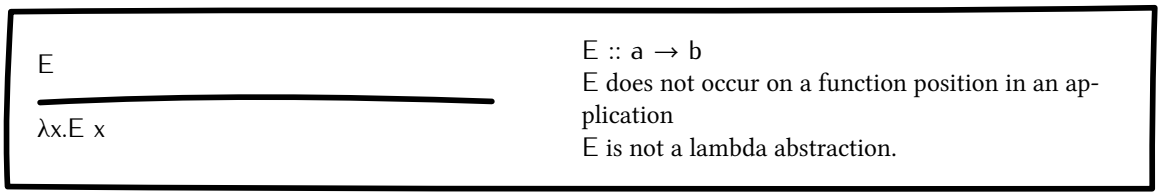
<transformed expression> This is the expression template that is the result of the transformation. If, looking at the above three items, the transformation applies, the <original expression> is completely replaced by the <transformed expression>. We call this a template, because it can contain placeholders, referring to any placeholder bound by the <original expression> or the <context conditions>. The resulting expression will have those placeholders replaced by the values bound to them.

Any binder (lowercase) placeholder that has no value bound to it yet will be bound to (and replaced with) a fresh binder.

<context additions> These are templates for new functions to be added to the context. This is a way to let a transformation create new top level functions.

Each addition has the form binder = template. As above, any placeholder in the addition is replaced with the value bound to it, and any binder placeholder that has no value bound to it yet will be bound to (and replaced with) a fresh binder.

To understand this notation better, the step by step application of the η-expansion transformation to a simple ALU will be shown. Consider η-expansion, which is a common transformation from lambda calculus, described using above notation as follows:



η-expansion is a well known transformation from lambda calculus. What this transformation does, is take any expression that has a function type and turn it into a lambda expression (giving an explicit name to the argument). There are some extra conditions that ensure that this transformation does not apply infinitely (which are not necessarily part of the conventional definition of η-expansion).

Consider the following function, in Core notation, which is a fairly obvious way to specify a simple ALU (Note that it is not yet in normal form, but example 4.3 shows the normal form of this function). The parentheses around the + and - operators are commonly used in Haskell to show that the operators are used as normal functions, instead of *infix* operators (*e.g.*, the operators appear before their arguments, instead of in between).

```
alu :: Bit → Word → Word → Word
alu = λopcode. case opcode of
  Low → (+)
  High → (-)
```

There are a few sub-expressions in this function to which we could possibly apply the transformation. Since the pattern of the transformation is only the placeholder E, any expression will match that. Whether the transformation applies to an expression is thus solely decided by the conditions to the right of the transformation.

We will look at each expression in the function in a top down manner. The first expression is the entire expression the function is bound to.

```
λopcode. case opcode of
  Low → (+)
  High → (-)
```

As said, the expression pattern matches this. The type of this expression is Bit → Word → Word → Word, which matches a → b (Note that in this case a = Bit and b = Word → Word → Word).

Since this expression is at top level, it does not occur at a function position of an application. However, The expression is a lambda abstraction, so this transformation does not apply.

The next expression we could apply this transformation to, is the body of the lambda abstraction:

```
case opcode of
  Low → (+)
  High → (-)
```

The type of this expression is Word → Word → Word, which again matches a → b. The expression is the body of a lambda expression, so it does not occur at a function position of an application. Finally, the expression is not a lambda abstraction but a case expression, so all the conditions match. There are no context conditions to match, so the transformation applies.

By now, the placeholder E is bound to the entire expression. The placeholder x, which occurs in the replacement template, is not bound yet, so we need to generate a fresh binder for that. Let us use the binder a. This results in the following replacement expression:

```
λa.(case opcode of
  Low → (+)
  High → (-)) a
```

Continuing with this expression, we see that the transformation does not apply again (it is a lambda expression). Next we look at the body of this lambda abstraction:

```
(case opcode of
  Low → (+)
  High → (-)) a
```

Here, the transformation does apply, binding E to the entire expression (which has type Word → Word) and binding x to the fresh binder b, resulting in the replacement:

```
λb.(case opcode of
  Low → (+)
  High → (-)) a b
```

The transformation does not apply to this lambda abstraction, so we look at its body. For brevity, we will put the case expression on one line from now on.

```
(case opcode of Low → (+); High → (-)) a b
```

The type of this expression is Word, so it does not match a → b and the transformation does not apply. Next, we have two options for the next expression to look at: the function position and argument position of the application. The expression in the argument position is b, which has type Word, so the transformation does not apply. The expression in the function position is:

```
(case opcode of Low → (+); High → (-)) a
```

Obviously, the transformation does not apply here, since it occurs in function position (which makes the second condition false). In the same way the transformation does not apply to both components of this expression (case opcode of Low → (+); High → (-) and a), so we will skip to the components of the case expression: the scrutinee and both alternatives. Since the opcode is not a function, it does not apply here.

The first alternative is (+). This expression has a function type (the operator still needs two arguments). It does not occur in function position of an application and it is not a lambda expression, so the transformation applies.

We look at the <original expression> pattern, which is E. This means we bind E to (+). We then replace the expression with the <transformed expression>, replacing all occurrences of E with (+). In the <transformed expression>, the This gives us the replacement expression: λx.(+) x (A lambda expression binding x, with a body that applies the addition operator to x).

The complete function then becomes:

```
(case opcode of Low → λa1.(+) a1; High → (-)) a
```

Now the transformation no longer applies to the complete first alternative (since it is a lambda expression). It does not apply to the addition operator again, since it is now in function position in an application. It does, however, apply to the application of the addition operator, since that is neither a lambda expression

nor does it occur in function position. This means after one more application of the transformation, the function becomes:

```
(case opcode of Low → λa1.λb1.(+) a1 b1; High → (-)) a
```

The other alternative is left as an exercise to the reader. The final function, after applying η-expansion until it does no longer apply is:

```
alu :: Bit → Word → Word → Word
alu = λopcode.λa.b. (case opcode of
  Low → λa1.λb1 (+) a1 b1
  High → λa2.λb2 (-) a2 b2) a b
```

4.2.1 Transformation application

In this chapter we define a number of transformations, but how will we apply these? As stated before, our normal form is reached as soon as no transformation applies anymore. This means our application strategy is to simply apply any transformation that applies, and continuing to do that with the result of each transformation.

In particular, we define no particular order of transformations. Since transformation order should not influence the resulting normal form, this leaves the implementation free to choose any application order that results in an efficient implementation. Unfortunately this is not entirely true for the current set of transformations. See section 4.4.2 for a discussion of this problem.

When applying a single transformation, we try to apply it to every (sub)expression in a function, not just the top level function body. This allows us to keep the transformation descriptions concise and powerful.

4.2.2 Definitions

A *global variable* is any variable (binder) that is bound at the top level of a program, or an external module. A *local variable* is any other variable (*e.g.*, variables local to a function, which can be bound by lambda abstractions, let expressions and pattern matches of case alternatives). This is a slightly different notion of global versus local than what GHC uses internally, but for our purposes the distinction GHC makes is not useful.

A *hardware representable* (or just *representable*) type or value is (a value of) a type that we can generate a signal for in hardware. For example, a bit, a vector of bits, a 32 bit unsigned word, etc. Values that are not run-time representable notably include (but are not limited to): types, dictionaries, functions.

A *built-in function* is a function supplied by the CλaSH framework, whose implementation is not used to generate VHDL. This is either because it is no valid CλaSH (like most list functions that need recursion) or because a CλaSH implementation would be unwanted (for the addition operator, for example, we would rather use the VHDLaddition operator to let the synthesis tool decide what kind of adder to use instead of explicitly describing one in CλaSH).

These are functions like map, hwor, + and length.

For these functions, CλaSH has a *built-in hardware translation*, so calls to these functions can still be translated. Built-in functions must have a valid Haskell implementation, of course, to allow simulation.

A *user-defined* function is a function for which no built-in translation is available and whose definition will thus need to be translated to CλaSH.

4.2.2.1 Predicates

Here, we define a number of predicates that can be used below to concisely specify conditions.

gvar(expr) is true when *expr* is a variable that references a global variable. It is false when it references a local variable.

lvar(expr) is the complement of *gvar*; it is true when *expr* references a local variable, false when it references a global variable.

representable(expr) is true when *expr* is *representable*.

4.2.3 Binder uniqueness

A common problem in transformation systems, is binder uniqueness. When not considering this problem, it is easy to create transformations that mix up bindings and cause name collisions. Take for example, the following Core expression:

```
(λa.λb.λc. a ∗ b ∗ c) x c
```

By applying β-reduction (see section 4.3.1.1) once, we can simplify this expression to:

```
(λb.λc. x ∗ b ∗ c) c
```

Now, we have replaced the a binder with a reference to the x binder. No harm done here. But note that we see multiple occurrences of the c binder. The first is a binding occurrence, to which the second refers. The last, however refers to *another* instance of c, which is bound somewhere outside of this expression. Now, if we would apply beta reduction without taking heed of binder uniqueness, we would get:

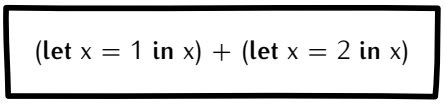
```
λc. x ∗ c ∗ c
```

This is obviously not what was supposed to happen! The root of this problem is the reuse of binders: identical binders can be bound in different, but overlapping scopes. Any variable reference in those overlapping scopes then refers to the variable bound in the inner (smallest) scope. There is not way to refer to the variable in the outer scope. This effect is usually referred to as *shadowing*: when a binder is bound in a scope where the binder already had a value, the inner binding is said to *shadow* the outer binding. In the example above, the c binder was bound outside of the expression and in the inner lambda expression. Inside that lambda expression, only the inner c can be accessed.

There are a number of ways to solve this. GHC has isolated this problem to their binder substitution code, which performs *de-shadowing* during its expression traversal. This means that any binding that shadows another binding on a higher level is replaced by a new binder that does not shadow any other binding. This non-shadowing invariant is enough to prevent binder uniqueness problems in GHC.

In our transformation system, maintaining this non-shadowing invariant is a bit harder to do (mostly due to implementation issues, the prototype does not use GHC's substitution code). Also, the following points can be observed.

- De-shadowing does not guarantee overall uniqueness. For example, the following (slightly contrived) expression shows the identifier x bound in two separate places (and to different values), even though no shadowing occurs.

- In our normal form (and the resulting VHDL), all binders (signals) within the same function (entity) will end up in the same scope. To allow this, all binders within the same function should be unique.

- When we know that all binders in an expression are unique, moving around or removing a sub-expression will never cause any binder conflicts. If we have some way to generate fresh binders, introducing new sub-expressions will not cause any problems either. The only way to cause conflicts is thus to duplicate an existing sub-expression.

Given the above, our prototype maintains a unique binder invariant. This means that in any given moment during normalization, all binders *within a single function* must be unique. To achieve this, we apply the following technique.

- Before starting normalization, all binders in the function are made unique. This is done by generating a fresh binder for every binder used. This also replaces binders that did not cause any conflict, but it does ensure that all binders within the function are generated by the same unique supply.

- Whenever a new binder must be generated, we generate a fresh binder that is guaranteed to be different from *all binders generated so far*. This can thus never introduce duplication and will maintain the invariant.

- Whenever (a part of) an expression is duplicated (for example when inlining), all binders in the expression are replaced with fresh binders (using the same method as at the start of normalization). These fresh binders can never introduce duplication, so this will maintain the invariant.

- Whenever we move part of an expression around within the function, there is no need to do anything special. There is obviously no way to introduce duplication by moving expressions around. Since we know that each of the binders is already unique, there is no way to introduce (incorrect) shadowing either.

4.3 Transform passes

In this section we describe the actual transforms.

Each transformation will be described informally first, explaining the need for and goal of the transformation. Then, we will formally define the transformation using the syntax introduced in section 4.2.

4.3.1 General cleanup

Substitution notation

In some of the transformations in this chapter, we need to perform substitution on an expression. Substitution means replacing every occurrence of some expression (usually a variable reference) with another expression.
There have been a lot of different notations used in literature for specifying substitution. The notation that will be used in this report is the following:

E[A⇒B]

This means expression E with all occurrences of A replaced with B.

These transformations are general cleanup transformations, that aim to make expressions simpler. These transformations usually clean up the mess left behind by other transformations or clean up expressions to expose new transformation opportunities for other transformations.

Most of these transformations are standard optimizations in other compilers as well. However, in our compiler, most of these are not just optimizations, but they are required to get our program into intended normal form.

4.3.1.1 β-reduction

β-reduction is a well known transformation from lambda calculus, where it is the main reduction step. It reduces applications of lambda abstractions, removing both the lambda abstraction and the application.

In our transformation system, this step helps to remove unwanted lambda abstractions (basically all but the ones at the top level). Other transformations (application propagation, non-representable inlining) make sure that most lambda abstractions will eventually be reducible by β-reduction.

Note that β-reduction also works on type lambda abstractions and type applications as well. This means the substitution below also works on type variables, in the case that the binder is a type variable and the expression applied to is a type.

$$\frac{(\lambda x.E)\ M}{E[x \Rightarrow M]}$$

Example 4.6 β-reduction

Original program	Transformed program
(λa. 2 ∗ a) (2 ∗ b)	2 ∗ (2 ∗ b)

Example 4.7 β-reduction for type abstractions

Original program | Transformed program

4.3.1.2 Unused let binding removal

This transformation removes let bindings that are never used. Occasionally, GHC's desugarer introduces some unused let bindings.

This normalization pass should really be not be necessary to get into intended normal form (since the intended normal form definition does not require that every binding is used), but in practice the desugarer or simplifier emits some bindings that cannot be normalized (e.g., calls to a `Control.Exception.Base.patError`) but are not used anywhere either. To prevent the VHDL generation from breaking on these artifacts, this transformation removes them.

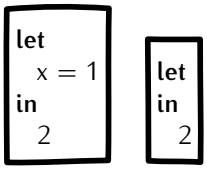
intended normal form definition p.56

$$\frac{\begin{array}{l}\textbf{letrec}\\ \quad a_0 = E_0\\ \quad \vdots\\ \quad a_i = E_i\\ \quad \vdots\\ \quad a_n = E_n\\ \textbf{in}\\ \quad M\end{array}}{\begin{array}{l}\textbf{letrec}\\ \quad a_0 = E_0\\ \quad \vdots\\ \quad a_{i-1} = E_{i-1}\\ \quad a_{i+1} = E_{i+1}\\ \quad \vdots\\ \quad a_n = E_n\\ \textbf{in}\\ \quad M\end{array}} \quad a_i \text{ does not occur free in } M \;\; \forall\, j,\, 0 \leq j \leq n,\, j \neq i \; (a_i \text{ does not occur free in } E_j)$$

4.3.1.3 Empty let removal

This transformation is simple: it removes recursive lets that have no bindings (which usually occurs when unused let binding removal removes the last binding from it).

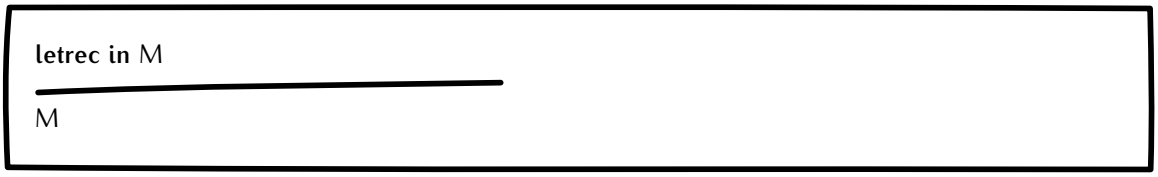
Example 4.8 Unused let binding removal



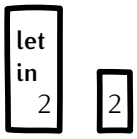
Original program

Transformed program

Note that there is no need to define this transformation for non-recursive lets, since they always contain exactly one binding.



Example 4.9 Empty let removal



Original program

Transformed program

4.3.1.4 Simple let binding removal

This transformation inlines simple let bindings, that bind some binder to some other binder instead of a more complex expression (*i.e.* a = b).

This transformation is not needed to get an expression into intended normal form (since these bindings are part of the intended normal form), but makes the resulting VHDL a lot shorter.

substitution notation p.64

letrec
  $a_0 = E_0$
  ⋮
  $a_i = b$
  ⋮
  $a_n = E_n$
in
  M

b is a variable reference
$a_i \neq b$

---

letrec
  $a_0 = E_0\ [a_i \Rightarrow b]$
  ⋮
  $a_{i-1} = E_{i-1}\ [a_i \Rightarrow b]$
  $a_{i+1} = E_{i+1}\ [a_i \Rightarrow b]$
  ⋮
  $a_n = E_n\ [a_i \Rightarrow b]$
in
  $M[a_i \Rightarrow b]$

4.3.1.5 Cast propagation / simplification

This transform pushes casts down into the expression as far as possible. This transformation has been added to make a few specific corner cases work, but it is not clear yet if this transformation handles cast expressions completely or in the right way. See section 4.4.3.

(**let** binds **in** E) ▸ T

---

let binds **in** (E ▸ T)

```
(case S of
  p0 → E0
  ⋮
  pn → En
) ▸ T
──────────────────────────────
case S of
  p0 → E0 ▸ T
  ⋮
  pn → En ▸ T
```

4.3.1.6 Top level binding inlining

top level binding p.17

This transform takes simple top level bindings generated by the GHC compiler. GHC sometimes generates very simple 'wrapper' bindings, which are bound to just a variable reference, or contain just a (partial) function application with the type and dictionary arguments filled in (such as the (+) in the example below).

Note that this transformation is completely optional. It is not required to get any function into intended normal form, but it does help making the resulting VHDL output easier to read (since it removes components that do not add any real structure, but do hide away operations and cause extra clutter).

This transform takes any top level binding generated by GHC, whose normalized form contains only a single let binding.

```
x = λa0 ... λan.let y = E in y
- - - - - - - - - - - - - - -
x
──────────────────────────────   x is generated by the compiler
λa0 ... λan.let y = E in y
```

Example 4.10 Top level binding inlining

```
(+) :: Word → Word → Word
(+) = GHC.Num.(+) @Word $dNum
- - - - - - - - - - - - - - -
(+) a b
```

Original program

```
GHC.Num.(+) @ Alu.Word $dNum a b
```

Transformed program

Example 4.10 shows a typical application of the addition operator generated by GHC. The type and dictionary arguments used here are described in Section 3.5.1.

Without this transformation, there would be a (+) entity in the VHDL which would just add its inputs. This generates a lot of overhead in the VHDL, which is particularly annoying when browsing the generated

RTL schematic (especially since most non-alphanumerics, like all characters in (+), are not allowed in VHDL architecture names[1], so the entity would be called 'w7aA7f' or something similarly meaningless and auto-generated).

4.3.2 Program structure

These transformations are aimed at normalizing the overall structure into the intended form. This means ensuring there is a lambda abstraction at the top for every argument (input port or current state), putting all of the other value definitions in let bindings and making the final return value a simple variable reference.

4.3.2.1 η-expansion

This transformation makes sure that all arguments of a function-typed expression are named, by introducing lambda expressions. When combined with β-reduction and non-representable binding inlining, all function-typed expressions should be lambda abstractions or global identifiers.

$$\frac{E}{\lambda x.E\ x}$$

$E :: a \to b$
E does not occur on a function position in an application
E is not a lambda abstraction.

Example 4.11 η-expansion

```
foo = λa.case a of
  True → λb.mul b b
  False → id
```

Original program

```
foo = λa.λx.(case a of
    True → λb.mul b b
    False → λy.id y) x
```

Transformed program

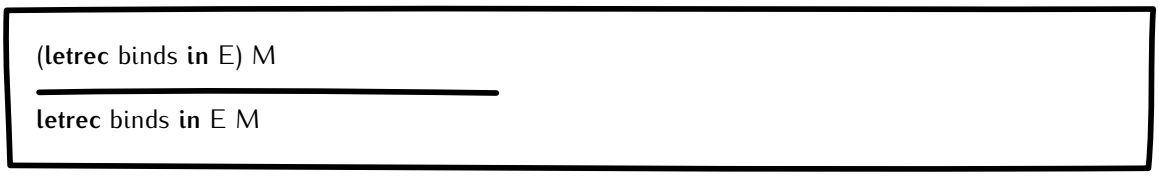
4.3.2.2 Application propagation

This transformation is meant to propagate application expressions downwards into expressions as far as possible. This allows partial applications inside expressions to become fully applied and exposes new transformation opportunities for other transformations (like β-reduction and specialization).

Since all binders in our expression are unique (see section 4.2.3), there is no risk that we will introduce unintended shadowing by moving an expression into a lower scope. Also, since only move expression

[1] *Technically, it is allowed to use non-alphanumerics when using extended identifiers, but it seems that none of the tooling likes extended identifiers in file names, so it effectively does not work.*

into smaller scopes (down into our expression), there is no risk of moving a variable reference out of the scope in which it is defined.



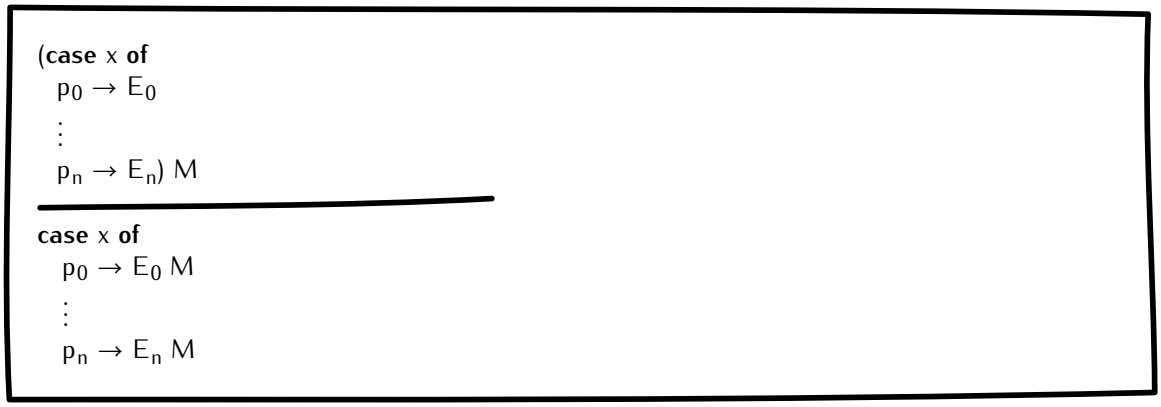
Example 4.12 Application propagation for a let expression

```
( letrec
    val = 1
  in
    add val
) 3
```

Original program

```
letrec
  val = 1
in
  add val 3
```

Transformed program



Example 4.13 Application propagation for a case expression

```
( case x of
    True → id
    False → neg
) 1
```

Original program

```
case x of
  True → id 1
  False → neg 1
```

Transformed program

4.3.2.3 Let recursification

This transformation makes all non-recursive lets recursive. In the end, we want a single recursive let in our normalized program, so all non-recursive lets can be converted. This also makes other transformations simpler: they only need to be specified for recursive let expressions (and simply will not apply to non-recursive let expressions until this transformation has been applied).

```
let
  a = E
in
  M
```

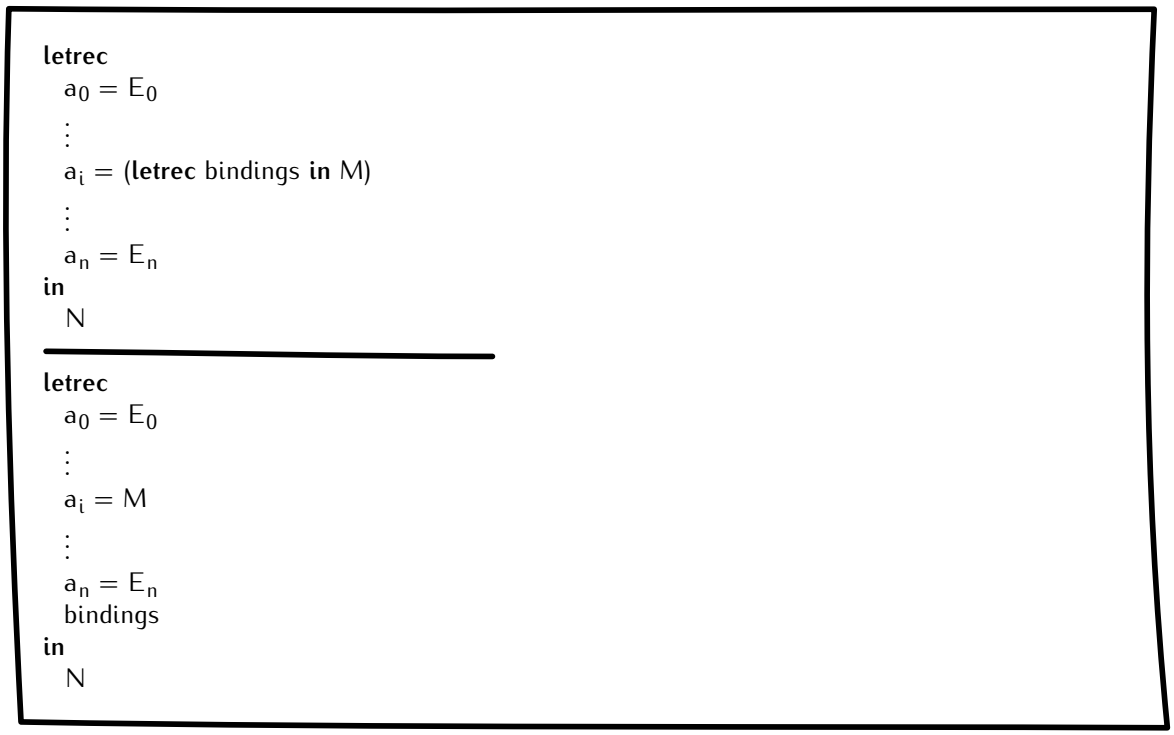
```
letrec
  a = E
in
  M
```

4.3.2.4 Let flattening

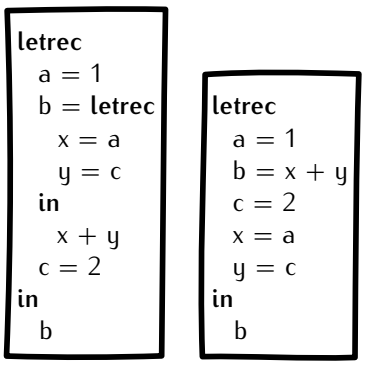
This transformation puts nested lets in the same scope, by lifting the binding(s) of the inner let into the outer let. Eventually, this will cause all let bindings to appear in the same scope.

This transformation only applies to recursive lets, since all non-recursive lets will be made recursive (see section 4.3.2.3).

Since we are joining two scopes together, there is no risk of moving a variable reference out of the scope where it is defined.



Example 4.14 Let flattening



Original program

Transformed program

4.3.2.5 Return value simplification

This transformation ensures that the return value of a function is always a simple local variable reference.

The basic idea of this transformation is to take the body of a function and bind it with a let expression (so the body of that let expression becomes a variable reference that can be used as the output port).

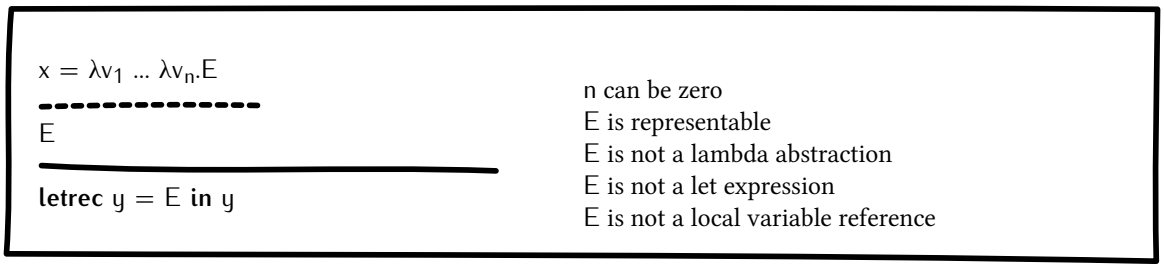
If the body of the function happens to have lambda abstractions at the top level (which is allowed by the intended normal form), we take the body of the inner lambda instead. If that happens to be a let expression already (which is allowed by the intended normal form), we take the body of that let (which is not allowed to be anything but a variable reference according the the intended normal form).

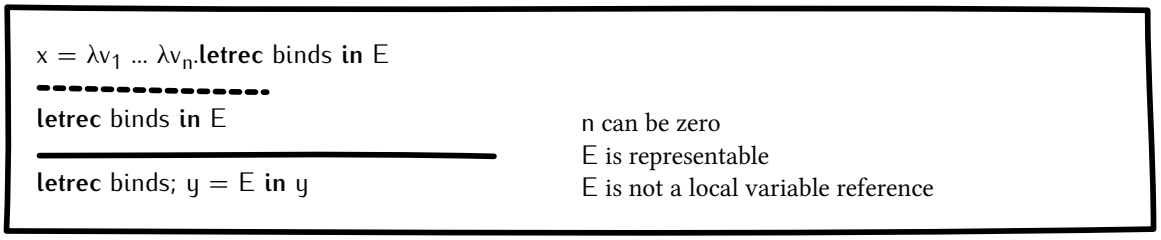
intended normal form definition p.56

This transformation uses the context conditions in a special way. These contexts, like $x = \lambda v_1 \ldots \lambda v_n.E$, are above the dotted line and provide a condition on the environment (*i.e.* they require a certain top level binding to be present). These ensure that expressions are only transformed when they are in the functions 'return value' directly. This means the context conditions have to interpreted in the right way: not 'if there is any function x that binds E, any E can be transformed', but we mean only the E that is bound by x).

Be careful when reading the transformations: Not the entire function from the context is transformed, just a part of it.

Note that the return value is not simplified if it is not representable. Otherwise, this would cause a loop with the inlining of unrepresentable bindings in section 4.3.6.4. If the return value is not representable because it has a function type, η-expansion should make sure that this transformation will eventually apply. If the value is not representable for other reasons, the function result itself is not representable, meaning this function is not translatable anyway.





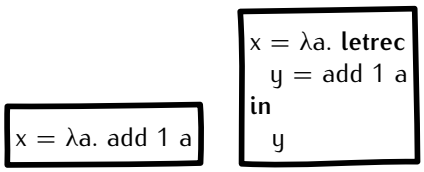
Example 4.15 Return value simplification

Original program	Transformed program
`x = add 1 2`	`x = letrec y = add 1 2 in y`

4.3.3 Representable arguments simplification

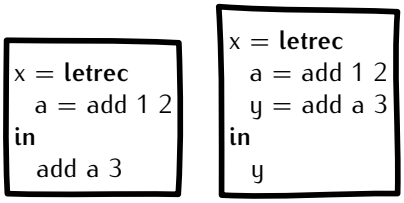
This section contains just a single transformation that deals with representable arguments in applications. Non-representable arguments are handled by the transformations in section 4.3.6.

Example 4.16 Return value simplification with a lambda abstraction



Original program | Transformed program

Example 4.17 Return value simplification with a let expression



Original program | Transformed program

This transformation ensures that all representable arguments will become references to local variables. This ensures they will become references to local signals in the resulting VHDL, which is required due to limitations in the component instantiation code in VHDL (one can only assign a signal or constant to an input port). By ensuring that all arguments are always simple variable references, we always have a signal available to map to the input ports.

To reduce a complex expression to a simple variable reference, we create a new let expression around the application, which binds the complex expression to a new variable. The original function is then applied to this variable.

global variable p.61

Note that references to *global variables* (like a top level function without arguments, but also an argumentless data-constructors like True) are also simplified. Only local variables generate signals in the resulting architecture. Even though argumentless data-constructors generate constants in generated VHDL code and could be mapped to an input port directly, they are still simplified to make the normal form more regular.

representable p.61

$$\frac{M\ N}{\textbf{letrec}\ x = N\ \textbf{in}\ M\ x}$$

N is representable
N is not a local variable reference

Example 4.18 Argument simplification

Original program Transformed program

local variable p.61

4.3.4 Built-in functions

This section deals with (arguments to) built-in functions. In the intended normal form definitionwe can see that there are three sorts of arguments a built-in function can receive.

intended normal form definition p.56

I. A representable local variable reference. This is the most common argument to any function. The argument simplification transformation described in section 4.3.3 makes sure that *any* representable argument to *any* function (including built-in functions) is turned into a local variable reference.

II. (A partial application of) a top level function (either built-in on user-defined). The function extraction transformation described in this section takes care of turning every function-typed argument into (a partial application of) a top level function.

III. Any expression that is not representable and does not have a function type. Since these can be any expression, there is no transformation needed. Note that this category is exactly all expressions that are not transformed by the transformations for the previous two categories. This means that *any* Core expression that is used as an argument to a built-in function will be either transformed into one of the above categories, or end up in this category. In any case, the result is in normal form.

As noted, the argument simplification will handle any representable arguments to a built-in function. The following transformation is needed to handle non-representable arguments with a function type, all other non-representable arguments do not need any special handling.

4.3.4.1 Function extraction

This transform deals with function-typed arguments to built-in functions. Since built-in functions cannot be specialized (see section 4.3.6.5) to remove the arguments, these arguments are extracted into a new global function instead. In other words, we create a new top level function that has exactly the extracted argument as its body. This greatly simplifies the translation rules needed for built-in functions, since they only need to handle (partial applications of) top level functions.

Any free variables occurring in the extracted arguments will become parameters to the new global function. The original argument is replaced with a reference to the new function, applied to any free variables from the original argument.

This transformation is useful when applying higher-order built-in functions like `map` to a lambda abstraction, for example. In this case, the code that generates VHDL for `map` only needs to handle top level

functions and partial applications, not any other expression (such as lambda abstractions or even more complicated expressions).

$$\frac{M\ N}{M\ (x\ f_0\ \ldots\ f_n)}\quad x = \lambda f_0\ \ldots\ \lambda f_n.N$$

M is (a partial application of) a built-in function.
f_0 ... f_n are all free local variables of N
N :: a → b
N is not a (partial application of) a top level function

Example 4.19 Function extraction

```
addList = λb.λxs.map (λa . add a b) xs
```

Original program

```
addList = λb.λxs.map (f b) xs
-----------------------------
f = λb.λa.add a b
```

Transformed program

Note that the function f will still need normalization after this.

4.3.5 Case normalization

The transformations in this section ensure that case statements end up in normal form.

4.3.5.1 Scrutinee simplification

This transform ensures that the scrutinee of a case expression is always a simple variable reference.

$$\frac{\textbf{case}\ E\ \textbf{of}\ alts}{\textbf{letrec}\ x = E\ \textbf{in}\ \textbf{case}\ x\ \textbf{of}\ alts}$$

E is not a local variable reference

Example 4.20 Case normalization

```
case (foo a) of
  True → a
  False → b
```

Original program

```
letrec x = foo a in
  case x of
    True → a
    False → b
```

Transformed program

4.3.5.2 Scrutinee binder removal

This transformation removes (or rather, makes wild) the binder to which the scrutinee is bound after evaluation. This is done by replacing the bndr with the scrutinee in all alternatives. To prevent duplication of work, this transformation is only applied when the scrutinee is already a simple variable reference (but the previous transformation ensures this will eventually be the case). The scrutinee binder itself is replaced by a wild binder (which is no longer displayed).

Note that one could argue that this transformation can change the meaning of the Core expression. In the regular Core semantics, a case expression forces the evaluation of its scrutinee and can be used to implement strict evaluation. However, in the generated VHDL, evaluation is always strict. So the semantics we assign to the Core expression (which differ only at this particular point), this transformation is completely valid.

Wild binders

In a functional expression, a *wild binder* refers to any binder that is never referenced. This means that even though it will be bound to a particular value, that value is never used. The Haskell syntax offers the underscore as a wild binder that cannot even be referenced (It can be seen as introducing a new, anonymous, binder every time it is used).
In these transformations, the term wild binder will sometimes be used to indicate that a binder must not be referenced.

```
case x of bndr
  alts
------------------------------   x is a local variable reference
case x of
  alts[bndr⇒x]
```

Example 4.21 Scrutinee binder removal

```
case x of y
  True → y
  False → not y
```

Original program

```
case x of
  True → x
  False → not x
```

Transformed program

4.3.5.3 Case normalization

This transformation ensures that all case expressions get a form that is allowed by the intended normal form. This means they will become one of:

- An extractor case with a single alternative that picks a field from a datatype, *e.g.* `case x of (a, b) → a`.
- A selector case with multiple alternatives and only wild binders, that makes a choice between expressions based on the constructor of another expression, *e.g.* `case x of Low → a; High → b`.

For an arbitrary case, that has n alternatives, with m binders in each alternatives, this will result in $m * n$ extractor case expression to get at each variable, n let bindings for each of the alternatives' value and a single selector case to select the right value out of these.

Technically, the definition of this transformation would require that the constructor for every alternative has exactly the same amount (m) of arguments, but of course this transformation also applies when this is not the case.

$$
\begin{array}{l}
\textbf{case}\ E\ \textbf{of} \\
\quad C_0\ v_{0,0}\ \ldots\ v_{0,m} \to E_0 \\
\quad \vdots \\
\quad C_n\ v_{n,0}\ \ldots\ v_{n,m} \to E_n \\
\hline
\textbf{letrec} \\
\quad v_{0,0} = \textbf{case}\ E\ \textbf{of}\ C_0\ x_{0,0}\ ..\ x_{0,m} \to x_{0,0} \\
\quad \vdots \\
\quad v_{0,m} = \textbf{case}\ E\ \textbf{of}\ C_0\ x_{0,0}\ ..\ x_{0,m} \to x_{0,m} \\
\quad \vdots \\
\quad v_{n,m} = \textbf{case}\ E\ \textbf{of}\ C_n\ x_{n,0}\ ..\ x_{n,m} \to x_{n,m} \\
\quad y_0 = E_0 \\
\quad \vdots \\
\quad y_n = E_n \\
\textbf{in} \\
\quad \textbf{case}\ E\ \textbf{of} \\
\quad\quad C_0\ w_{0,0}\ \ldots\ w_{0,m} \to y_0 \\
\quad\quad \vdots \\
\quad\quad C_n\ w_{n,0}\ \ldots\ w_{n,m} \to y_n
\end{array}
$$

$\forall\ i\ \forall\ j,\ 0 \leq i \leq n,\ 0 \leq i < m$ ($w_{i,j}$ is a wild (unused) binder)
The case expression is not an extractor case
The case expression is not a selector case

Note that this transformation applies to case expressions with any scrutinee. If the scrutinee is a complex expression, this might result in duplication of work (hardware). An extra condition to only apply this transformation when the scrutinee is already simple (effectively causing this transformation to be only applied after the scrutinee simplification transformation) might be in order.

Example 4.22 Selector case simplification

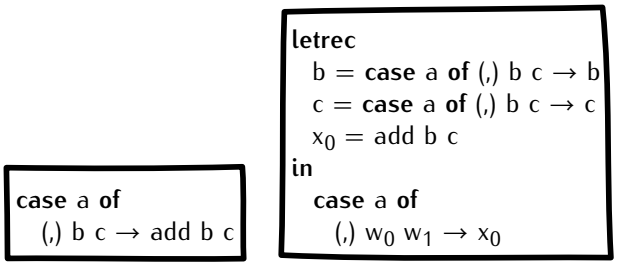
```
case a of
  True → add b 1
  False → add b 2
```

Original program

```
letrec
  x0 = add b 1
  x1 = add b 2
in
  case a of
    True → x0
    False → x1
```

Transformed program

Example 4.23 Extractor case simplification



Original program Transformed program

In example 4.23 the case expression is expanded into multiple case expressions, including a pretty useless expression (that is neither a selector or extractor case). This case can be removed by the Case removal transformation in section 4.3.5.4.

selector case p.77

4.3.5.4 Case removal

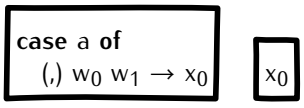
This transform removes any case expression with a single alternative and only wild binders.

wild binders p.77

These "useless" case expressions are usually leftovers from case simplification on extractor case (see the previous example).

$$\frac{\texttt{case } x \texttt{ of } C\ v_0 \ldots v_m \rightarrow E}{E} \quad \forall\, i,\ 0 \leq i \leq m\ (v_i \text{ does not occur free in E})$$

Example 4.24 Case removal



Original program Transformed program

4.3.6 Removing unrepresentable values

The transformations in this section are aimed at making all the values used in our expression representable. There are two main transformations that are applied to *all* unrepresentable let bindings and function arguments. These are meant to address three different kinds of unrepresentable values: polymorphic values, higher-order values and literals. The transformation are described generically: they apply to all non-representable values. However, non-representable values that do not fall into one of

these three categories will be moved around by these transformations but are unlikely to completely disappear. They usually mean the program was not valid in the first place, because unsupported types were used (for example, a program using strings).

Each of these three categories will be detailed below, followed by the actual transformations.

4.3.6.1 Removing Polymorphism

As noted in section 3.5.1, polymorphism is made explicit in Core through type and dictionary arguments. To remove the polymorphism from a function, we can simply specialize the polymorphic function for the particular type applied to it. The same goes for dictionary arguments. To remove polymorphism from let bound values, we simply inline the let bindings that have a polymorphic type, which should (eventually) make sure that the polymorphic expression is applied to a type and/or dictionary, which can then be removed by β-reduction (section 4.3.1.1).

representable p.61

Since both type and dictionary arguments are not representable, the non-representable argument specialization and non-representable let binding inlining transformations below take care of exactly this.

There is one case where polymorphism cannot be completely removed: built-in functions are still allowed to be polymorphic (Since we have no function body that we could properly specialize). However, the code that generates VHDL for built-in functions knows how to handle this, so this is not a problem.

4.3.6.2 Defunctionalization

These transformations remove higher-order expressions from our program, making all values first-order. The approach used for defunctionalization uses a combination of specialization, inlining and some cleanup transformations, was also proposed in parallel research by Neil Mitchell [12].

Higher order values are always introduced by lambda abstractions, none of the other Core expression elements can introduce a function type. However, other expressions can *have* a function type, when they have a lambda expression in their body.

For example, the following expression is a higher-order expression that is not a lambda expression itself:

id function p.40

```
case x of
  High → id
  Low → λx.x
```

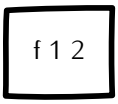
The reference to the id function shows that we can introduce a higher-order expression in our program without using a lambda expression directly. However, inside the definition of the id function, we can be sure that a lambda expression is present.

Looking closely at the definition of our normal form in section 4.1.1, we can see that there are three possibilities for higher-order values to appear in our intended normal form:

I. Lambda abstractions can appear at the highest level of a top level function. These lambda abstractions introduce the arguments (input ports / current state) of the function.

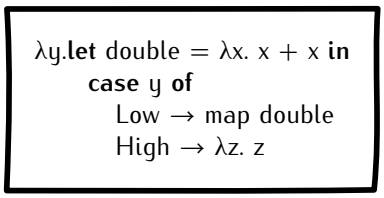
II. (Partial applications of) top level functions can appear as an argument to a built-in function.

III. (Partial applications of) top level functions can appear in function position of an application. Since a partial application cannot appear anywhere else (except as built-in function arguments), all partial applications are applied, meaning that all applications will become complete applications. However, since application of arguments happens one by one, in the expression:



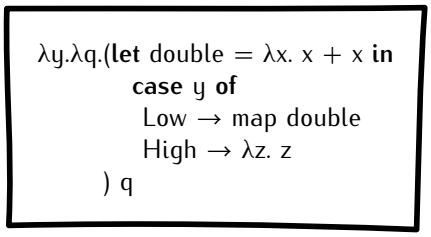
the sub-expression f 1 has a function type. But this is allowed, since it is inside a complete application.

We will take a typical function with some higher-order values as an example. The following function takes two arguments: a Bit and a list of numbers. Depending on the first argument, each number in the list is doubled, or the list is returned unmodified. For the sake of the example, no polymorphism is shown. In reality, at least map would be polymorphic.



This example shows a number of higher-order values that we cannot translate to VHDL directly. The double binder bound in the let expression has a function type, as well as both of the alternatives of the case expression. The first alternative is a partial application of the map built-in function, whereas the second alternative is a lambda abstraction.

To reduce all higher-order values to one of the above items, a number of transformations we have already seen are used. The η-expansion transformation from section 4.3.2.1 ensures all function arguments are introduced by lambda abstraction on the highest level of a function. These lambda arguments are allowed because of item I above. After η-expansion, our example becomes a bit bigger:



η-expansion also introduces extra applications (the application of the let expression to q in the above example). These applications can then propagated down by the application propagation transformation

(section 4.3.2.2). In our example, the q and r variable will be propagated into the let expression and then into the case expression:

```
λy.λq.let double = λx. x + x in
        case y of
         Low → map double q
         High → (λz. z) q
```

This propagation makes higher-order values become applied (in particular both of the alternatives of the case now have a representable type). Completely applied top level functions (like the first alternative) are now no longer invalid (they fall under item III above). (Completely) applied lambda abstractions can be removed by β-expansion. For our example, applying β-expansion results in the following:

```
λy.λq.let double = λx. x + x in
        case y of
         Low → map double q
         High → q
```

As you can see in our example, all of this moves applications towards the higher-order values, but misses higher-order functions bound by let expressions. The applications cannot be moved towards these values (since they can be used in multiple places), so the values will have to be moved towards the applications. This is achieved by inlining all higher-order values bound by let applications, by the non-representable binding inlining transformation below. When applying it to our example, we get the following:

```
λy.λq.case y of
        Low → map (λx. x + x) q
        High → q
```

We have nearly eliminated all unsupported higher-order values from this expressions. The one that is remaining is the first argument to the map function. Having higher-order arguments to a built-in function like map is allowed in the intended normal form, but only if the argument is a (partial application) of a top level function. This is easily done by introducing a new top level function and put the lambda abstraction inside. This is done by the function extraction transformation from section 4.3.4.1.

```
λy.λq.case y of
        Low → map func q
        High → q
```

This also introduces a new function, that we have called func:

```
func = λx. x + x
```

Note that this does not actually remove the lambda, but now it is a lambda at the highest level of a function, which is allowed in the intended normal form.

There is one case that has not been discussed yet. What if the map function in the example above was not a built-in function but a user-defined function? Then extracting the lambda expression into a new function would not be enough, since user-defined functions can never have higher-order arguments. For example, the following expression shows an example:

```
twice :: (Word → Word) → Word → Word
twice = λf.λa.f (f a)

main = λa.app (λx. x + x) a
```

This example shows a function twice that takes a function as a first argument and applies that function twice to the second argument. Again, we have made the function monomorphic for clarity, even though this function would be a lot more useful if it was polymorphic. The function main uses twice to apply a lambda expression twice.

When faced with a user defined function, a body is available for that function. This means we could create a specialized version of the function that only works for this particular higher-order argument (*i.e.*, we can just remove the argument and call the specialized function without the argument). This transformation is detailed below. Applying this transformation to the example gives:

```
twice′ :: Word → Word
twice′ = λb.(λf.λa.f (f a)) (λx. x + x) b

main = λa.app′ a
```

The main function is now in normal form, since the only higher-order value there is the top level lambda expression. The new twice′ function is a bit complex, but the entire original body of the original twice function is wrapped in a lambda abstraction and applied to the argument we have specialized for (λx. x + x) and the other arguments. This complex expression can fortunately be effectively reduced by repeatedly applying β-reduction:

```
twice′ :: Word → Word
twice′ = λb.(b + b) + (b + b)
```

This example also shows that the resulting normal form might not be as efficient as we might hope it to be (it is calculating (b + b) twice). This is discussed in more detail in section 4.4.1.

4.3.6.3 Literals

enumerated types p.21

There are a limited number of literals available in Haskell and Core. When using (enumerating) algebraic data-types, a literal is just a reference to the corresponding data constructor, which has a representable type (the algebraic datatype) and can be translated directly. This also holds for literals of the `Bool` Haskell type, which is just an enumerated type.

There is, however, a second type of literal that does not have a representable type: integer literals. CλaSH supports using integer literals for all three integer types supported (`SizedWord`, `SizedInt` and `RangedWord`). This is implemented using Haskell's `Num` type class, which offers a `fromInteger` method that converts any `Integer` to the CλaSH data-types.

When GHC sees integer literals, it will automatically insert calls to the `fromInteger` method in the resulting Core expression. For example, the following expression in Haskell creates a 32 bit unsigned word with the value 1. The explicit type signature is needed, since there is no context for GHC to determine the type from otherwise.

```
1 :: SizedWord D32
```

This Haskell code results in the following Core expression:

```
fromInteger @(SizedWord D32) $dNum (smallInteger 10)
```

The literal 10 will have the type GHC.Prim.Int#, which is converted into an Integer by smallInteger. Finally, the fromInteger function will finally convert this into a SizedWord D_{32}.

Both the GHC.Prim.Int# and Integer types are not representable, and cannot be translated directly. Fortunately, there is no need to translate them, since fromInteger is a built-in function that knows how to handle these values. However, this does require that the fromInteger function is directly applied to these non-representable literal values, otherwise errors will occur. For example, the following expression is not in the intended normal form, since one of the let bindings has an unrepresentable type (Integer):

```
let l = smallInteger 10 in fromInteger @(SizedWord D32) $dNum l
```

By inlining these let-bindings, we can ensure that unrepresentable literals bound by a let binding end up in an application of the appropriate built-in function, where they are allowed. Since it is possible that the application of that function is in a different function than the definition of the literal value, we will

always need to specialize away any unrepresentable literals that are used as function arguments. The following two transformations do exactly this.

4.3.6.4 Non-representable binding inlining

This transform inlines let bindings that are bound to a non-representable value. Since we can never generate a signal assignment for these bindings (we cannot declare a signal assignment with a non-representable type, for obvious reasons), we have no choice but to inline the binding to remove it.

As we have seen in the previous sections, inlining these bindings solves (part of) the polymorphism, higher-order values and unrepresentable literals in an expression.

substitution notation p.64

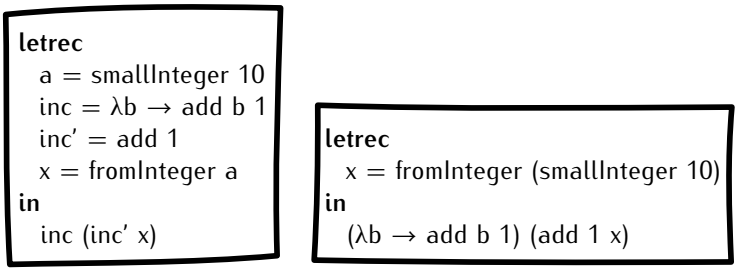
letrec
  $a_0 = E_0$
  $\vdots$
  $a_i = E_i$
  $\vdots$
  $a_n = E_n$
in
  M

E_i has a non-representable type.

---

letrec
  $a_0 = E_0\ [a_i \Rightarrow E_i] \vdots$
  $a_{i-1} = E_{i-1}\ [a_i \Rightarrow E_i]$
  $a_{i+1} = E_{i+1}\ [a_i \Rightarrow E_i]$
  $\vdots$
  $a_n = E_n\ [a_i \Rightarrow E_i]$
in
  $M[a_i \Rightarrow E_i]$

Example 4.25 Non-representable binding inlining



Original program Transformed program

4.3.6.5 Function specialization

This transform removes arguments to user-defined functions that are not representable at run-time. This is done by creating a *specialized* version of the function that only works for one particular value of that argument (in other words, the argument can be removed).

Specialization means to create a specialized version of the called function, with one argument already filled in. As a simple example, in the following program (this is not actual Core, since it directly uses a literal with the unrepresentable type GHC.Prim.Int#).

```
f = λa.λb.a + b
inc = λa.f a 1
```

We could specialize the function f against the literal argument 1, with the following result:

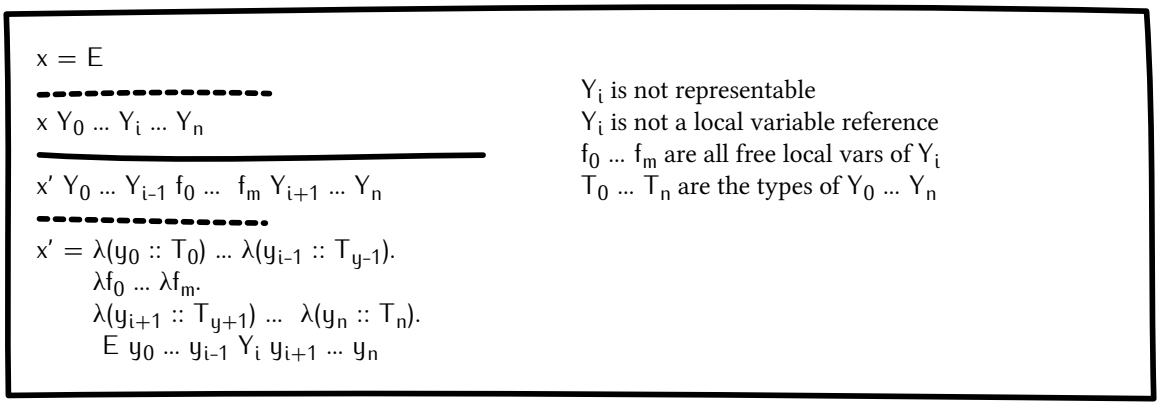
```
f' = λa.a + 1
inc = λa.f' a
```

In some way, this transformation is similar to β-reduction, but it operates across function boundaries. It is also similar to non-representable let binding inlining above, since it sort of 'inlines' an expression into a called function.

Special care must be taken when the argument has any free variables. If this is the case, the original argument should not be removed completely, but replaced by all the free variables of the expression. In this way, the original expression can still be evaluated inside the new function.

To prevent us from propagating the same argument over and over, a simple local variable reference is not propagated (since is has exactly one free variable, itself, we would only replace that argument with itself).

This shows that any free local variables that are not run-time representable cannot be brought into normal form by this transform. We rely on an inlining or β-reduction transformation to replace such a variable with an expression we can propagate again.



This is a bit of a complex transformation. It transforms an application of the function x, where one of the arguments (Y_i) is not representable. A new function x′ is created that wraps the body of the old function. The body of the new function becomes a number of nested lambda abstractions, one for each of the original arguments that are left unchanged.

The ith argument is replaced with the free variables of Y_i. Note that we reuse the same binders as those used in Y_i, since we can then just use Y_i inside the new function body and have all of the variables it uses be in scope.

The argument that we are specializing for, Y_i, is put inside the new function body. The old function body is applied to it. Since we use this new function only in place of an application with that particular argument Y_i, behavior should not change.

Note that the types of the arguments of our new function are taken from the types of the *actual* arguments (T_0 ... T_n). This means that any polymorphism in the arguments is removed, even when the corresponding explicit type lambda is not removed yet.

4.4 Unsolved problems

The above system of transformations has been implemented in the prototype and seems to work well to compile simple and more complex examples of hardware descriptions [1]. However, this normalization system has not seen enough review and work to be complete and work for every Core expression that is supplied to it. A number of problems have already been identified and are discussed in this section.

4.4.1 Work duplication

A possible problem of β-reduction is that it could duplicate work. When the expression applied is not a simple variable reference, but requires calculation and the binder the lambda abstraction binds to is used more than once, more hardware might be generated than strictly needed.

As an example, consider the expression:

```
(λx. x + x) (a * b)
```

When applying β-reduction to this expression, we get:

```
(a * b) + (a * b)
```

which of course calculates (a * b) twice.

A possible solution to this would be to use the following alternative transformation, which is of course no longer normal β-reduction. The following transformation has not been tested in the prototype, but is given here for future reference:

$$\frac{(\lambda x.E)\ M}{\textbf{letrec}\ x = M\ \textbf{in}\ E}$$

This does not seem like much of an improvement, but it does get rid of the lambda expression (and the associated higher-order value), while at the same time introducing a new let binding. Since the result of every application or case expression must be bound by a let expression in the intended normal form anyway, this is probably not a problem. If the argument happens to be a variable reference, then simple let binding removal (section 4.3.1.4) will remove it, making the result identical to that of the original β-reduction transformation.

When also applying argument simplification to the above example, we get the following expression:

```
let y = (a * b)
    z = (a * b)
in y + z
```

Looking at this, we could imagine an alternative approach: create a transformation that removes let bindings that bind identical values. In the above expression, the y and z variables could be merged together, resulting in the more efficient expression:

```
let y = (a * b) in y + y
```

4.4.2 Non-determinism

As an example, again consider the following expression:

```
(λx. x + x) (a * b)
```

We can apply both β-reduction (section 4.3.1.1) as well as argument simplification (section 4.3.3) to this expression.

When applying argument simplification first and then β-reduction, we get the following expression:

```
let y = (a * b) in y + y
```

When applying β-reduction first and then argument simplification, we get the following expression:

```
let y = (a * b)
    z = (a * b)
in y + z
```

As you can see, this is a different expression. This means that the order of expressions, does in fact change the resulting normal form, which is something that we would like to avoid. In this particular case one of the alternatives is even clearly more efficient, so we would of course like the more efficient form to be the normal form.

For this particular problem, the solutions for duplication of work seem from the previous section seem to fix the determinism of our transformation system as well. However, it is likely that there are other occurrences of this problem.

4.4.3 Casts

We do not fully understand the use of cast expressions in Core, so there are probably expressions involving cast expressions that cannot be brought into intended normal form by this transformation system.

The uses of casts in the Core system should be investigated more and transformations will probably need updating to handle them in all cases.

4.4.4 Normalization of stateful descriptions

Currently, the intended normal form definitionoffers enough freedom to describe all valid stateful descriptions, but is not limiting enough. It is possible to write descriptions which are in intended normal form, but cannot be translated into VHDL in a meaningful way (*e.g.*, a function that swaps two substates in its result, or a function that changes a sub-state itself instead of passing it to a sub-function).

intended normal form definition p.56

It is now up to the programmer to not do anything funny with these state values, whereas the normalization just tries not to mess up the flow of state values. In practice, there are situations where a Core program that *could* be a valid stateful description is not translatable by the prototype. This most often happens when statefulness is mixed with pattern matching, causing a state input to be unpacked multiple times or be unpacked and repacked only in some of the code paths.

Without going into detail about the exact problems (of which there are probably more than have shown up so far), it seems unlikely that these problems can be solved entirely by just improving the VHDL state generation in the final stage. The normalization stage seems the best place to apply the rewriting needed to support more complex stateful descriptions. This does of course mean that the intended normal form definition must be extended as well to be more specific about how state handling should look like in normal form. Section 3.7.1 already contains a tight description of the limitations on the use of state variables, which could be adapted into the intended normal form.

4.5 Provable properties

When looking at the system of transformations outlined above, there are a number of questions that we can ask ourselves. The main question is of course: 'Does our system work as intended?'. We can split this question into a number of sub-questions:

I. Does our system *terminate*? Since our system will keep running as long as transformations apply, there is an obvious risk that it will keep running indefinitely. This typically happens when one transformation produces a result that is transformed back to the original by another transformation, or when one or more transformations keep expanding some expression.

II. Is our system *sound*? Since our transformations continuously modify the expression, there is an obvious risk that the final normal form will not be equivalent to the original program: its meaning could have changed.

III. Is our system *complete*? Since we have a complex system of transformations, there is an obvious risk that some expressions will not end up in our intended normal form, because we forgot some transformation. In other words: does our transformation system result in our intended normal form for all possible inputs?

IV. Is our system *deterministic*? Since we have defined no particular order in which the transformation should be applied, there is an obvious risk that different transformation orderings will result in *different* normal forms. They might still both be intended normal forms (if our system is *complete*) and describe correct hardware (if our system is *sound*), so this property is less important than the previous three: the translator would still function properly without it.

Unfortunately, the final transformation system has only been developed in the final part of the research, leaving no more time for verifying these properties. In fact, it is likely that the current transformation system still violates some of these properties in some cases (see section 4.4.2 and section 4.4.4) and should be improved (or extra conditions on the input hardware descriptions should be formulated).

This is most likely the case with the completeness and determinism properties, perhaps also the termination property. The soundness property probably holds, since it is easier to manually verify (each transformation can be reviewed separately).

Even though no complete proofs have been made, some ideas for possible proof strategies are shown below.

4.5.1 Graph representation

Before looking into how to prove these properties, we will look at transformation systems from a graph perspective. We will first define the graph view and then illustrate it using a simple example from lambda calculus (which is a different system than the Cλash normalization system). The nodes of the graph are all possible Core expressions. The (directed) edges of the graph are transformations. When a transformation α applies to an expression A to produce an expression B, we add an edge from the node for A to the node for B, labeled α.

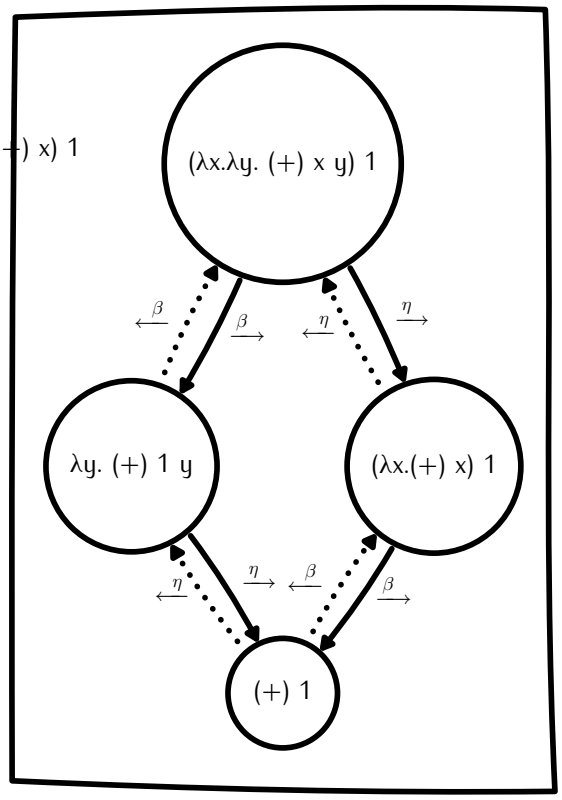
Of course the graph for Cλash is unbounded, since we can construct an infinite amount of Core expressions. Also, there might potentially be multiple edges between two given nodes (with different labels), though this seems unlikely to actually happen in our system.

See example 4.26 for the graph representation of a very simple lambda calculus that contains just the expressions (λx.λy. (+) x y) 1, λy. (+) 1 y, (λx.(+) x) 1 and (+) 1. The transformation system consists of β-reduction and η-reduction (solid edges) or β-expansion and η-expansion (dotted edges).

In such a graph a node (expression) is in normal form if it has no outgoing edges (meaning no transformation applies to it). The set of nodes without outgoing edges is called the *normal set*. Similarly, the set of nodes containing expressions in intended normal form is called the *intended normal set*.

From such a graph, we can derive some properties easily:

Example 4.26 Partial graph of a lambda calculus system with β and η reduction (solid lines) and expansion (dotted lines).



intended normal form definition p.56

I. A system will *terminate* if there is no walk (sequence of edges, or transformations) of infinite length in the graph (this includes cycles, but can also happen without cycles).

II. Soundness is not easily represented in the graph.

III. A system is *complete* if all of the nodes in the normal set have the intended normal form. The inverse (that all of the nodes outside of the normal set are *not* in the intended normal form) is not strictly required. In other words, our normal set must be a subset of the intended normal form, but they do not need to be the same set. form.

IV. A system is deterministic if all paths starting at a particular node, which end in a node in the normal set, end at the same node.

When looking at the example 4.26, we see that the system terminates for both the reduction and expansion systems (but note that, for expansion, this is only true because we have limited the possible expressions. In complete lambda calculus, there would be a path from (λx.λy. (+) x y) 1 to (λx.λy.(λz.(+) z) x y) 1 to (λx.λy.(λz.(λq.(+) q) z) x y) 1 etc.)

If we would consider the system with both expansion and reduction, there would no longer be termination either, since there would be cycles all over the place.

The reduction and expansion systems have a normal set of containing just (+) 1 or (λx.λy. (+) x y) 1 respectively. Since all paths in either system end up in these normal forms, both systems are *complete*. Also, since there is only one node in the normal set, it must obviously be *deterministic* as well.

4.5.2 Termination

In general, proving termination of an arbitrary program is a very hard problem. Fortunately, we only have to prove termination for our specific transformation system.

A common approach for these kinds of proofs is to associate a measure with each possible expression in our system. If we can show that each transformation strictly decreases this measure (*i.e.*, the expression transformed to has a lower measure than the expression transformed from).

A good measure for a system consisting of just β-reduction would be the number of lambda expressions in the expression. Since every application of β-reduction removes a lambda abstraction (and there is always a bounded number of lambda abstractions in every expression) we can easily see that a transformation system with just β-reduction will always terminate.

For our complete system, this measure would be fairly complex (probably the sum of a lot of things). Since the (conditions on) our transformations are pretty complex, we would need to include both simple things like the number of let expressions as well as more complex things like the number of case expressions that are not yet in normal form.

No real attempt has been made at finding a suitable measure for our system yet.

4.5.3 Soundness

Soundness is a property that can be proven for each transformation separately. Since our system only runs separate transformations sequentially, if each of our transformations leaves the *meaning* of the expression unchanged, then the entire system will of course leave the meaning unchanged and is thus *sound*.

The current prototype has only been verified in an ad hoc fashion by inspecting (the code for) each transformation. A more formal verification would be more appropriate.

To be able to formally show that each transformation properly preserves the meaning of every expression, we require an exact definition of the *meaning* of every expression, so we can compare them. A definition of the operational semantics of GHC's Core language is available [13], but this does not seem sufficient for our goals (but it is a good start).

It should be possible to have a single formal definition of meaning for Core for both normal Core compilation by GHC and for our compilation to VHDL. The main difference seems to be that in hardware every expression is always evaluated, while in software it is only evaluated if needed, but it should be possible to assign a meaning to Core expressions that assumes neither.

Since each of the transformations can be applied to any sub-expression as well, there is a constraint on our meaning definition: the meaning of an expression should depend only on the meaning of sub-expressions, not on the expressions themselves. For example, the meaning of the application in f (**let** x = 4 **in** x)

should be the same as the meaning of the application in f 4, since the argument sub-expression has the same meaning (though the actual expression is different).

4.5.4 Completeness

Proving completeness is probably not hard, but it could be a lot of work. We have seen above that to prove completeness, we must show that the normal set of our graph representation is a subset of the intended normal set.

However, it is hard to systematically generate or reason about the normal set, since it is defined as any nodes to which no transformation applies. To determine this set, each transformation must be considered and when a transformation is added, the entire set should be re-evaluated. This means it is hard to show that each node in the normal set is also in the intended normal set. Reasoning about our intended normal set is easier, since we know how to generate it from its definition.

intended normal form definition p.56

Fortunately, we can also prove the complement (which is equivalent, since $A \subseteq B \Leftrightarrow \overline{B} \subseteq \overline{A}$): show that the set of nodes not in intended normal form is a subset of the set of nodes not in normal form. In other words, show that for every expression that is not in intended normal form, that there is at least one transformation that applies to it (since that means it is not in normal form either and since $A \subseteq C \Leftrightarrow \forall x(x \in A \rightarrow x \in C)$).

By systematically reviewing the entire Core language definition along with the intended normal form definition (both of which have a similar structure), it should be possible to identify all possible (sets of) Core expressions that are not in intended normal form and identify a transformation that applies to it.

This approach is especially useful for proving completeness of our system, since if expressions exist to which none of the transformations apply (*i.e.* if the system is not yet complete), it is immediately clear which expressions these are and adding (or modifying) transformations to fix this should be relatively easy.

As observed above, applying this approach is a lot of work, since we need to check every (set of) transformation(s) separately.

4.5.5 Determinism

A well-known technique for proving determinism in lambda calculus and other reduction systems, is using the Church-Rosser property [14]. A reduction system has the CR property if and only if:

Definition 4.27 Church-Rosser theorem

$\forall$ A, B, C $\exists$ D (A $\twoheadrightarrow$ B $\wedge$ A $\twoheadrightarrow$ C $\Rightarrow$ B $\twoheadrightarrow$ D $\wedge$ C $\twoheadrightarrow$ D)

Here, A $\twoheadrightarrow$ B means A *reduces to* B. In other words, there is a set of transformations that can transform A to B. $\Rightarrow$ is used to mean *implies*.

For a transformation system holding the Church-Rosser property, it is easy to show that it is in fact deterministic. Showing that this property actually holds is a harder problem, but has been done for some reduction systems in the lambda calculus [15] [16]. Doing the same for our transformation system is probably more complicated, but not impossible.

5 Future work

5.1 New language

During the development of the prototype, it has become clear that Haskell is not the perfect language for this work. There are two main problems:

- Haskell is not expressive enough. Haskell is still quite new in the area of dependent typing, something we use extensively. Also, Haskell has some special syntax to write monadic composition and arrow composition, which is well suited to normal Haskell programs. However, for our hardware descriptions, we could use some similar, but other, syntactic sugar (as we have seen above).

- Haskell is too expressive. There are some things that Haskell can express, but we can never properly translate. There are certain types (both primitive types and certain type constructions like recursive types) we can never translate, as well as certain forms of recursion.

It might be good to reevaluate the choice of language for Cλash, perhaps there are other languages which are better suited to describe hardware, now that we have learned a lot about it.

Perhaps Haskell (or some other language) can be extended by using a preprocessor. There has been some (point of proof) work on a the Strathclyde Haskell Enhancement (SHE) preprocessor for type-level programming, perhaps we can extend that, or create something similar for hardware-specific notation.

It is not unlikely that the most flexible way forward would be to define a completely new language with exactly the needed features. This is of course an enormous effort, which should not be taken lightly.

5.2 Correctness proofs of the normalization system

As stated in section 4.5, there are a number of properties we would like to see verified about the normalization system. In particular, the *termination* and *completeness* of the system would be a good candidate for future research. Specifying formal semantics for the Core language in order to verify the *soundness* of the system would be an even more challenging task.

5.3 Improved notation for hierarchical state

The hierarchical state model requires quite some boilerplate code for unpacking and distributing the input state and collecting and repacking the output state.

Example 5.1 shows a simple composition of two stateful functions, `funca` and `funcb`. The state annotation using the `State` newtype has been left out, for clarity and because the proposed approach below cannot handle that (yet).

Example 5.1 Simple function composing two stateful functions.

```
type FooState = ( AState, BState )
foo :: Word -> FooState -> (FooState, Word)
foo in s = (s', outb)
  where
    (sa, sb) = s
    (sa', outa) = funca in sa
    (sb', outb) = funcb outa sb
    s' = (sa', sb')
```

Since state handling always follows strict rules, there is really no other way in which this state handling can be done (you can of course move some things around from the where clause to the pattern or vice versa, but it would still be effectively the same thing). This means it makes extra sense to hide this boilerplate away. This would incur no flexibility cost at all, since there are no other ways that would work.

One particular notation in Haskell that seems promising, is the `do` notation. This is meant to simplify Monad notation by hiding away some details. It allows one to write a list of expressions, which are composed using the monadic *bind* operator, written in Haskell as `>>`. For example, the snippet:

```
do
  somefunc a
  otherfunc b
```

will be desugared into:

```
(somefunc a) >> (otherfunc b)
```

The main reason to have the monadic notation, is to be able to wrap results of functions in a datatype (the *monad*) that can contain extra information, and hide extra behavior in the binding operators.

The `>>=` operator allows extracting the actual result of a function and passing it to another function. The following snippet should illustrate this:

```
do
  x <- somefunc a
  otherfunc x
```

will be desugared into:

```
(somefunc a) >>= (\x -> otherfunc x)
```

The `\x -> ...` notation creates a lambda abstraction in Haskell, that binds the `x` variable. Here, the `>>=` operator is supposed to extract whatever result somefunc has and pass it to the lambda expression created. This will probably not make the monadic notation completely clear to a reader without prior experience with Haskell, but it should serve to understand the following discussion.

The monadic notation could perhaps be used to compose a number of stateful functions into another stateful computation. Perhaps this could move all the boilerplate code into the `>>` and `>>=` operators. Because the boilerplate is still there (it has not magically disappeared, just moved into these functions), the compiler should still be able to compile these descriptions without any special magic (though perhaps it should always inline the binding operators to reveal the flow of values).

This highlights an important aspect of using a functional language for our descriptions: we can use the language itself to provide abstractions of common patterns, making our code smaller and easier to read.

5.3.1 Breaking out of the Monad

However, simply using the monad notation is not as easy as it sounds. The main problem is that the Monad type class poses a number of limitations on the bind operator `>>`. Most importantly, it has the following type signature:

```
(>>) :: (Monad m) => m a -> m b -> m b
```

This means that any expression in our composition must use the same Monad instance as its type, only the "return" value can be different between expressions.

Ideally, we would like the `>>` operator to have a type like the following

```
type Stateful s r = s -> (s, r)
(>>) :: Stateful s1 r1 -> Stateful s2 r2 -> Stateful (s1, s2) r2
```

What we see here, is that when we compose two stateful functions (that have already been applied to their inputs, leaving just the state argument to be applied to), the result is again a stateful function whose state is composed of the two *substates*. The return value is simply the return value of the second function, discarding the first (to preserve that result, the `>>=` operator can be used).

There is a trick we can apply to change the signature of the `>>` operator. GHC does not require the bind operators to be part of the `Monad` type class, as long as it can use them to translate the do notation. This means we can define our own `>>` and `>>=` operators, outside of the `Monad` type class. This does conflict with the existing methods of the `Monad` type class, so we should prevent GHC from loading those (and all of the Prelude) by passing `-XNoImplicitPrelude` to `ghc`. This is slightly inconvenient, but since we hardly using anything from the prelude, this is not a big problem. We might even be able to replace some of the Prelude with hardware-translatable versions by doing this.

The binding operators can now be defined exactly as they are needed. For completeness, the `return` function is also defined. It is not called by GHC implicitly, but can be called explicitly by a hardware description. Example 5.2 shows these definitions.

These definitions closely resemble the boilerplate of unpacking state, passing it to two functions and repacking the new state. With these definitions, we could have written example 5.1 a lot shorter, see example 5.3. In this example the type signature of foo is the same (though it is now written using the `Stateful` type synonym, it is still completely equivalent to the original: `foo :: Word -> FooState -> (FooState, Word)`.

Note that the `FooState` type has changed (so indirectly the type of `foo` as well). Since the state composition by the `>>` works on two stateful functions at a time, the final state consists of nested two-tuples. The final `()` in the state originates from the fact that the `return` function has no real state, but is part of the composition. We could have left out the return expression (and the `outb <-` part) to make `foo`'s return value equal to `funcb`'s, but this approach makes it clearer what is happening.

Example 5.2 Binding operators to compose stateful computations

```
(>>) :: Stateful s1 r1 -> Stateful s2 r2 -> Stateful (s1, s2) r2
f1 >> f2 = f1 >>= \_ -> f2

(>>=) :: Stateful s1 r1 -> (r1 -> Stateful s2 r2) -> Stateful (s1, s2) r2
f1 >>= f2 = \(s1, s2) -> let (s1', r1) = f1 s1
                             (s2', r2) = f2 r1 s2
                         in ((s1', s2'), r2)

return :: r -> Stateful () r
return r = \s -> (s, r)
```

Example 5.3 Simple function composing two stateful functions, using do notation.

```
type FooState = ( AState, (BState, ()) )
foo :: Word -> Stateful FooState Word
foo in = do
    outa <- funca in
    outb <- funcb outa
    return outb
```

An important implication of this approach is that the order of writing function applications affects the state type. Fortunately, this problem can be localized by consistently using type synonyms for state types (see section 3.6.3), which should prevent changes in other function's source when a function changes.

A less obvious implications of this approach is that the scope of variables produced by each of these expressions (using the `<-` syntax) is limited to the expressions that come after it. This prevents values from flowing between two functions (components) in two directions. For most Monad instances, this is a requirement, but here it could have been different.

5.3.2 Alternative syntax

Because of these typing issues, misusing Haskell's do notation is probably not the best solution here. However, it does show that using fairly simple abstractions, we could hide a lot of the boilerplate code. Extending GHC with some new syntax sugar similar to the do notation might be a feasible.

5.3.3 Arrows

Another abstraction mechanism offered by Haskell are arrows. Arrows are a generalization of monads [17], for which GHC also supports some syntax sugar [18]. Their use for hiding away state boilerplate is

not directly evident, but since arrows are a complex concept further investigation is appropriate.

5.4 Improved notation or abstraction for pipelining

Since pipelining is a very common optimization for hardware systems, it should be easy to specify a pipelined system. Since it introduces quite some registers into an otherwise regular combinational system, we might look for some way to abstract away some of the boilerplate for pipelining.

Something similar to the state boilerplate removal above might be appropriate: Abstract away some of the boilerplate code using combinators, then hide away the combinators in special syntax. The combinators will be slightly different, since there is a (typing) distinction between a pipeline stage and a pipeline consisting of multiple stages. Also, it seems necessary to treat either the first or the last pipeline stage differently, to prevent an off-by-one error in the amount of registers (which is similar to the extra `()` state type in example 5.3, which is harmless there, but would be a problem if it introduced an extra, useless, pipeline stage).

This problem is slightly more complex than the problem we have seen before. One significant difference is that each variable that crosses a stage boundary needs a register. However, when a variable crosses multiple stage boundaries, it must be stored for a longer period and should receive multiple registers. Since we cannot find out from the combinator code where the result of the combined values is used (at least not without using Template Haskell to inspect the AST), there seems to be no easy way to find how much registers are needed.

There seem to be two obvious ways of handling this problem:

- Limit the scoping of each variable produced by a stage to the next stage only. This means that any variables that are to be used in subsequent stages should be passed on explicitly, which should allocate the required number of registers.

  This produces cumbersome code, where there is still a lot of explicitness (though this could be hidden in syntax sugar).

- Scope each variable over every subsequent pipeline stage and allocate the maximum number of registers that *could* be needed. This means we will allocate registers that are never used, but those could be optimized away later. This does mean we need some way to introduce a variable number of variables (depending on the total number of stages), assign the output of a different register to each (*e.g.*., a different part of the state) and scope a different one of them over each the subsequent stages.

  This also means that when adding a stage to an existing pipeline will change the state type of each of the subsequent pipeline stages, and the state type of the added stage depends on the number of subsequent stages.

  Properly describing this will probably also require quite explicit syntax, meaning this is not feasible without some special syntax.

Some other interesting issues include pipeline stages which are already stateful, mixing pipelined with normal computation, etc.

5.5 Recursion

The main problems of recursion have been described in section 2.8. In the current implementation, recursion is therefore not possible, instead we rely on a number of implicitly list-recursive built-in functions.

Since recursion is a very important and central concept in functional programming, it would very much improve the flexibility and elegance of our hardware descriptions if we could support (full) recursion.

For this, there are two main problems to solve:

- For list recursion, how to make a description type check? It is quite possible that improvements in the GHC type-checker will make this possible, though it will still stay a challenge. Further advances in dependent typing support for Haskell will probably help here as well.

- For all recursion, there is the obvious challenge of deciding when recursion is finished. For list recursion, this might be easier (Since the base case of the recursion influences the type signatures). For general recursion, this requires a complete set of simplification and evaluation transformations to prevent infinite expansion. The main challenge here is how to make this set complete, or at least define the constraints on possible recursion that guarantee it will work.

5.6 Multiple clock domains and asynchronicity

Cλash currently only supports synchronous systems with a single clock domain. In a lot of real-world systems, both of these limitations pose problems.

There might be asynchronous events to which a system wants to respond. The most obvious asynchronous event is of course a reset signal. Though a reset signal can be synchronous, that is less flexible (and a hassle to describe in Cλash, currently). Since every function in Cλash describes the behavior on each cycle boundary, we really cannot fit in asynchronous behavior easily.

Due to the same reason, multiple clock domains cannot be easily supported. There is currently no way for the compiler to know in which clock domain a function should operate and since the clock signal is never explicit, there is also no way to express circuits that synchronize various clock domains.

A possible way to express more complex timing behavior would be to make functions more generic event handlers, where the system generates a stream of events (Like 'clock up', 'clock down', 'input A changed', 'reset', etc.). When working with multiple clock domains, each domain could get its own clock events.

Example 5.4 shows a simple example of this event-based approach. In this example we see that every function takes an input of type `Event`. The function `main` that takes the output of `func` and accumulates it on every clock cycle. On a reset signal, the accumulator is reset. The function `func` is just a combinational function, with no synchronous elements. We can see this because there is no state and the event input is completely ignored. If the compiler is smart enough, we could even leave the event input out for functions that do not use it, either because they are completely combinational (like in this example), or because they rely on the the caller to select the clock signal.

This structure is similar to the event handling structure used to perform I/O in languages like Amanda. There is a top level case expression that decides what to do depending on the current input event.

A slightly more complex example is show in example 5.5. It shows a system with two clock domains. There is no real integration between the clock domains in this example (there is one input and one output for each clock domain), but it does show how multiple clocks can be distinguished.

Note that in example 5.5 the `suba` and `subb` functions are *always* called, to get at their combinational outputs. The new state is calculated as well, but only saved when the right clock has an up transition.

As you can see, there is some code duplication in the case expression that selects the right clock. One of the advantages of an explicit approach like this, is that some of this duplication can be extracted away into helper functions. For example, we could imagine a `select_clock` function, which takes a stateful function that is not aware of events, and changes it into a function that only updates its state on a specific (clock) event. Such a function is shown in example 5.6.

Example 5.4 Hardware description using asynchronous events.

```
data Event = ClockUp | Reset | ...

type MainState = State Word

-- Initial state
initstate :: MainState
initstate = State 0

main :: Word -> Event -> MainState -> (MainState, Word)
main inp event (State acc) = (State acc', acc')
  where
    acc' = case event of
      -- On a reset signal, reset the accumulator and output
      Reset -> initstate
      -- On a normal clock cycle, accumulate the result of func
      ClockUp -> acc + (func inp event)
      -- On any other event, leave state and output unchanged
      _ -> acc

-- func is some combinational expression
func :: Word -> Event -> Word
func inp _ = inp * 2 + 3
```

As you can see, this can greatly reduce the length of the main function, while increasing the readability. As you might also have noted, the select_clock function takes any stateful function from the current Cλash prototype and turns it into an event-aware function!

Going along this route for more complex timing behavior seems promising, especially since it seems possible to express very advanced timing behaviors, while still allowing simple functions without any extra overhead when complex behavior is not needed.

The main cost of this approach will probably be extra complexity in the compiler: the paths (state) data can take become very non-trivial, and it is probably hard to properly analyze these paths and produce the intended VHDL description.

5.7 Multiple cycle descriptions

In the current Cλash prototype, every description is a single-cycle description. In other words, every function describes behavior for each separate cycle and any recursion (or folds, maps, etc.) is expanded in space.

Sometimes, you might want to have a complex description that could possibly take multiple cycles. Some examples include:

- Some kind of map or fold over a list that could be expanded in time instead of space. This would result in a function that describes n cycles instead of just one, where n is the length of the list.
- A large combinational expressions that would introduce a very long combinational path and thus limit clock frequency. Such an expression could be broken up into multiple stages, which effectively

Example 5.5 Hardware description with multiple clock domains through events.

```
data Event = ClockUpA | ClockUpB | ...

type MainState = State (SubAState, SubBState)

-- Initial state
initstate :: MainState
initstate = State (initsubastate, initsubbstate)

main :: Word -> Word -> Event -> MainState -> (MainState, Word, Word)
main inpa inpb event (State (sa, sb)) = (State (sa', sb'), outa, outb)
  where
    -- Only update the substates if the corresponding clock has an up
    -- transition.
    sa' = case event of
      ClockUpA -> sa''
      _ -> sa
    sb' = case event of
      ClockUpB -> sb''
      _ -> sb
    -- Always call suba and subb, so we can always have a value for our output
    -- ports.
    (sa'', outa) = suba inpa sa
    (sb'', outb) = subb inpb sb

type SubAState = State ...
suba :: Word -> SubAState -> (SubAState, Word)
suba = ...

type SubBState = State ...
subb :: Word -> SubAState -> (SubAState, Word)
subb = ...
```

results in a pipelined system (see also section 5.4) with a known delay. There should probably be some way for the developer to specify the cycle division of the expression, since automatically deciding on such a division is too complex and contains too many trade-offs, at least initially.

- Unbounded recursion. It is not possible to expand unbounded (or even recursion with a depth that is not known at compile time) in space, since there is no way to tell how much hardware to create (it might even be infinite).

  When expanding infinite recursion over time, each step of the recursion can simply become a single clock cycle. When expanding bounded but unknown recursion, we probably need to add an extra data valid output bit or something similar.

Apart from translating each of these multiple cycle descriptions into a per-cycle description, we also need to somehow match the type signature of the multiple cycle description to the type signature of the single cycle description that the rest of the system expects (assuming that the rest of the system is described in

Example 5.6 A function to filter clock events.

```
select_clock :: Event
                -> (input -> State s -> (State s, output))
                -> (input -> State s -> Event -> (State s, output))
select_clock clock func inp state event = (state', out)
  where
    state' = if clock == event then state'' else state
    (state'', out) = func inp state

main :: Word -> Word -> Event -> MainState -> (MainState, Word, Word)
main inpa inpb event (State (sa, sb)) = (State (sa', sb'), outa, outb)
  where
    (sa'', outa) = (select_clock ClockUpA suba) inpa sa event
    (sb'', outb) = (select_clock ClockUpB subb) inpb sb event
```

the 'normal' per-cycle manner). For example, an infinitely recursive expression typically has the return type [a], while the rest of the system would expect just a (since the recursive expression generates just a single element each cycle).

Naively, this matching could be done using a (built-in) function with a signature like $[a] \to a$, which also serves as an indicator to the compiler that some expanding over time is required. However, this poses a problem for simulation: how will our Haskell implementation of this magical built-in function know which element of the input list to return. This obviously depends on the current cycle number, but there is no way for this function to know the current cycle without breaking all kinds of safety and purity properties. Making this function have some state input and output could help, though this state is not present in the actual hardware (or perhaps there is some state, if there are value passed from one recursion step to the next, but how can the type of that state be determined by the type-checker?).

It seems that this is the most pressing problem for multi-cycle descriptions: How to interface with the rest of the system, without sacrificing safety and simulation capabilities?

5.8 Higher order values in state

Another interesting idea is putting a higher-order value inside a function's state value. Since we can use higher-order values anywhere, why not in the state?

Example 5.7 An accumulator using a higher-order state.

```
-- The accumulator function that takes a word and returns a new accumulator
-- and the result so far. This is the function we want to put inside the
-- state.
type Acc = Word -> (Acc, Word)
acc = \a -> (\b -> acc ( a + b ), a )

main :: Word -> State Acc -> (State Acc, Word)
main a s = (State s', out)
  where (s', out) = s a
```

As a (contrived) example, consider the accumulator in example 5.7. This code uses a function as its state, which implicitly contains the value accumulated so far. This is a fairly trivial example, that is more easy to write with a simple `Word` state value, but for more complex descriptions this style might pay off. Note that in a way we are using the *continuation passing style* of writing functions, where we pass a sort of *continuation* from one cycle to the next.

Our normalization process completely removes higher-order values inside a function by moving applications and higher-order values around so that every higher-order value will eventually be full applied. However, if we would put a higher-order value inside our state, the path from the higher-order value definition to its application runs through the state, which is external to the function. A higher-order value defined in one cycle is not applied until a later cycle. Our normalization process only works within the function, so it cannot remove this use of a higher-order value.

However, we cannot leave a higher-order value in our state value, since it is impossible to generate a register containing a higher-order value, we simply cannot translate a function type to a hardware type. To solve this, we must replace the higher-order value inside our state with some other value representing these higher-order values.

On obvious approach here is to use an algebraic datatype where each constructor represents one possible higher-order value that can end up in the state and each constructor has an argument for each free variable of the higher-order value replaced. This allows us to completely reconstruct the higher-order value at the spot where it is applied, and thus the higher-order value disappears.

This approach is commonly known as the 'Reynolds approach to defunctionalization', first described by J.C. Reynolds [19] and seems to apply well to this situation. One note here is that Reynolds' approach puts all the higher-order values in a single datatype. For a typed language, we will at least have to have a single datatype for each function type, since we cannot mix them. It would be even better to split these data-types a bit further, so that such a datatype will never hold a constructor that is never used for a particular state variable. This separation is probably a non-trivial problem, though.

5.9 Don't care values

A powerful value in VHDL is the *don't care* value, given as `'-'`. This value tells the compiler that you do not really care about which value is assigned to a signal, allowing the compiler to make some optimizations. Since choice in hardware is often implemented using a collection of logic gates instead of multiplexers only, synthesizers can often reduce the amount of hardware needed by smartly choosing values for these don't care cases.

There is not really anything comparable with don't care values in normal programming languages. The closest thing is an undefined or uninitialized value, though those are usually associated with error conditions.

It would be useful if CλaSh also has some way to specify don't care values. When looking at the Haskell typing system, there are really two ways to do this:

- Add a special don't care value to every datatype. This includes the obvious `Bit` type, but will also need to include every user defined type. An exception can be made for vectors and product types, since those can simply use the don't care values of the types they contain.

  This would also require some kind of 'Don't careable' type class that allows each type to specify what its don't care value is. The compiler must then recognize this constant and replace it with don't care values in the final VHDL code.

  This is of course a very intrusive solution. Every type must become member of this type class, and there is now some member in every type that is a special don't care value. Guaranteeing the obvious

don't care semantics also becomes harder, since every pattern match or case expressions must now also take care of the don't care value (this might actually be an advantage, since it forces designers to specify how to handle don't care for different operations).

- Use the special `undefined`, or *bottom* value present in Haskell. This is a type that is member of all types automatically, without any explicit declarations.

  Any expression that requires evaluation of an undefined value automatically becomes undefined itself (or rather, there is some exception mechanism). Since Haskell is lazy, this means that whenever it tries to evaluate undefined, it is apparently required for determining the output of the system. This property is useful, since typically a don't care output is used when some output is not valid and should not be read. If it is in fact not read, it should never be evaluated and simulation should be fine.

  In practice, this works less ideal. In particular, pattern matching is not always smart enough to deal with undefined. Consider the following definition of an `and` operator:

  ```
  and :: Bit -> Bit -> Bit
  and Low _ = Low
  and _ Low = Low
  and High High = High
  ```

  When using the `and` operation on an undefined (don't care) and a Low value should always return a Low value. Its value does not depend on the value chosen for the don't care value. However, though when applying the above and function to `Low` and `undefined` results in exactly that behavior, the result is `undefined` when the arguments are swapped. This is because the first pattern forces the first argument to be evaluated. If it is `undefined`, evaluation is halted and an exception is show, which is not what is intended.

These options should be explored further to see if they provide feasible methods for describing don't care conditions. Possibly there are completely other methods which work better.

6 Conclusions

The product of this research is a system called CλaSH, which allows hardware descriptions written in Haskell to be translated to VHDLand be programmed into an FPGA.

In this research, we have seen that a functional language is well suited for hardware descriptions. Function applications provide elegant notation for component instantiation and the various choice mechanisms (pattern matching, case expressions, if expressions) are well suited to describe conditional assignment in the hardware.

Useful features from the functional perspective, like polymorphism and higher-order functions and expressions also prove suitable to describe hardware and our implementation shows that they can be translated to VHDL as well.

A prototype compiler was created in this research. For this prototype the Haskell language was chosen as the input language, instead of creating a new language from scratch. This has enabled creating the prototype rapidly, allowing for experimenting with various functional language features and interpretations in CλaSH. Even supporting more complex features like polymorphism and higher-order values has been possible. If a new language and compiler would have been designed from scratch, that new language would not have been nearly as advanced as the current version of CλaSH.

However, Haskell might not have been the best choice for describing hardware. Some of the expressiveness it offers is not appropriate for hardware description (such as infinite recursion or recursive types), but also some extra syntax sugar could be useful (to reduce boilerplate).

The lack of real dependent typing support in Haskell has been a burden. Haskell provides the tools to create some type level programming constructs (and is improving fast), but another language might have support for more advanced dependent types (and even type level operations) as a fundamental part of the language. The need for dependent typing is particularly present in CλaSH to be able to fix some properties (list length, recursion depth, etc.) at compile time. Having better support for dependent typing might allow the use of type-safe recursion in CλaSH, though this is fundamentally still a hard problem.

The choice of describing state very explicitly as extra arguments and results is a mixed blessing. It provides very explicit specification of state, which allows for very clear descriptions. This also allows for easy modification of the description in our normalization program, since state can be handled just like other arguments and results.

On the other hand, the explicitness of the states and in particular substates, mean that more complex descriptions can become cumbersome very quickly. One finds that dealing with unpacking, passing, receiving and repacking becomes tedious and even error-prone. Removing some of this boilerplate would make the language even easier to use.

On the whole, the usefulness of CλaSH for describing hardware is not completely clear yet. Most elements of the language have proven suitable, and even a real world hardware circuit (a reducer circuit [1]) has been implemented. However, the language has not been used during a complete design process, where its rapid prototyping and reusability qualities could become real advantages, or perhaps the state boilerplate or synchronicity limitations could become real problems.

It is expected that CλaSH will be used as a tool in education at the University of Twente soon. Hopefully this will provide a better insight in how the system performs.

The prototype compiler has a clear design. Its front-end is taken from the GHC compiler and desugars Haskell into a small, but functional and typed language, called *Core*. CλaSH adds a transformation system that reduces this small language to a normal form and a simple back-end that performs a direct translation to VHDL. This approach has worked well and should probably be preserved. Especially the transformation based normalization system is suitable. It is easy to implement a transformation in the prototype, though it is not trivial to maintain enough overview to guarantee that the system is correct and complete. In fact, the current set of transformations is probably not complete yet, in particular when stateful descriptions are involved. However, the system can be (and has been) described in a mathematical sense, allowing us to reason about it and probably also prove various correctness properties in the future.

The scope of this research has been limited to structural descriptions that are synchronous in a single clock domain using cycle accurate designs. Even though this is a broad spectrum already, it may turn out that this scope is too narrow for practical use of Cλash. A common response from people that hear about using a functional language for hardware description is that they hope to be able to provide a concise mathematical description of their algorithm and have the hardware generated for them. Since this is obviously a different problem altogether, we could not have hoped to even start solving it. However, hopefully the current Cλash system provides a solid base on top of which further experimentation with functional descriptions can be built.

References

1 Baaij, C. (2009). Cλash: From haskell to hardware. Master's thesis, University of Twente.

2 van Deursen, A., Klint, P. and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not., 35*(6), 26–36.

3 Hudak, P..Building domain-specific embedded languages. *ACM Comput. Surv.*, pp. 196.

4 Sheeran, M. (1983). *μFP, an algebraic VLSI design language.* PhD thesis, Programming Research Group, Oxford University.

5 Jones, G. and Sheeran, M. (1990). Circuit design in ruby. In *Formal Methods for VLSI Design* . Lyngby, Denmark.

6 Claessen, K. (2000). *An Embedded Language Approach to Hardware Description and Verification.* PhD thesis, Dept. of Computer Science and Engineering, Chalmers University of Technology..

7 Sander, I. and Jantsch, A. (2004). System modeling and design refinement in forsyde. In , pages 17–32.

8 Gill, A. (2009). Type-safe observable sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium* .

9 Li, W. and Switzer, H. (1989). A unified data exchnage environment based on edif. In *DAC '89: Proceedings of the 26th ACM/IEEE Design Automation Conference* , pages 803–806. New York, NY, USA.

10 Jones, S. L. P. (1996). Compiling haskell by program transformation: A report from the trenches. In *Programming Languages and Systems — ESOP '96* , pages 18–44.

11 Peyton Jones, S. L. (1987). *The implementation of functional programming languages.* Prentice-Hall.

12 Mitchell, N. and Runciman, C. (2009). Losing functions without gaining data: another look at defunctionalisation. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell* , pages 13–24. New York, NY, USA.

13 Sulzmann, M., Chakravarty, M. M. T., Jones, S. P. and Donnelly, K. (2007). System f with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation* , pages 53–66. New York, NY, USA.

14 Church, A. and Rosser, J. B. (1936). Some properties of conversion. *Transactions of the American Mathematical Society, 39*(3), 472–482.

15 Klop, J. (1980). *Combinatory reduction systems.* Mathematisch Centrum.

16 Barendregt, H. P. (1984). *The lambda calculus.* revised edition Elsevier Science.

17 Hughes, J. (1998). Generalising monads to arrows. *Science of Computer Programming, 37*, 67–111.

18 Paterson, R. (2001). A new notation for arrows. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* , pages 229–240. New York, NY, USA.

19 Reynolds, J. C. (1998). Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput., 11*(4), 363–397.