CλasH:
From Haskell To Hardware

Master's Thesis

of

Christiaan Baaij

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

December 14, 2009

Abstract

Functional hardware description languages are a class of hardware description languages that emphasize on the ability to express higher level structural properties, such a parameterization and regularity. Due to such features as higher-order functions and polymorphism, parameterization in functional hardware description languages is more natural than the parameterization support found in the more traditional hardware description languages, like VHDL and Verilog. We develop a new functional hardware description language, CλasH, that borrows both the syntax and semantics from the general-purpose functional programming language Haskell.

In many existing functional hardware description languages, a circuit designer has to use language primitives that are encoded as data-types and combinators within Haskell. In CλasH on the other hand, circuit designers build their circuits using regular Haskell syntax. Where many existing languages encode state using a so-called *delay* element within the body of a function, CλasH specifications explicitly encode state in the *type*-signature of a function thereby avoiding the node-sharing problem most other functional hardware description languages face.

To cope with the direct physical restrictions of hardware, the familiar dynamically sized lists found in Haskell are replaced with fixed-size vectors. Being in essence a subset of Haskell, CλasH inherits the strong typing system of Haskell. CλasH exploits this typing system to specify the dependently-typed fixed-size vectors, be it that the dependent types are 'fake'. As the designers of Haskell never set out to create a dependently typed language, the fixed-size vector specification suffers slightly from limits imposed by the typing system. Still, the developed fixed-size vector library presents a myriad of functionality to an eventual circuit designer. Besides having support for fixed-size vectors, CλasH also incorporates two integer type primitives.

CλasH can be used to develop more than just trivial designs, exemplified by the reduction circuit designed with it. The CλasH design of this reduction circuit runs only 50% slower than a hand-coded optimized VHDL design, even though this first generation CλasH compiler does not have any optimizations whatsoever. With the used FPGA resources being in the same order as the resources used by the hand-coded VHDL we are confident that this first-generation compiler is indeed well behaved.

Much has been accomplished with this first attempt at developing a new functional hardware description language, as it already allows us to build more than just trivial designs. There are however many possibilities for future work, the most pressing being able to support recursive functions.

Acknowledgements

At the end of my *Informatica* bachelor I was convinced (as naïve as I was, and probably still am) that enjoyable intellectual challenges could only be found in the intersection of the fields of Electrical Engineering and Computer Science. That is, most Computer Science courses, given a few exception, in my curriculum were simply not that challenging, and hence, not fun. The exceptions were, perhaps unsurprisingly in retrospect, Functional Programming and Compiler Construction.

As I started my master's degree on Embedded Systems, I never thought that those two subjects would play such an important role in my master's thesis. When I was in the office of professor Smit, looking for a subject for my thesis, I was told that Jan Kuper had some ideas for a project. Jan's description of the project literally was (be it that he said it in Dutch): *"Do you remember those functional description I showed in ECA2? That's what I want. To make real hardware out of them".* Quite a vague description, and not the type of project I was expecting to find, but very interesting nonetheless; a chance to go back to those subjects of Computer Science I found interesting during bachelor and combine it with my (limited) acquired knowledge of hardware designs. It only took a few moments of deliberation to come to the conclusion that this project would indeed be a joy to work on.

Only a month into the project I was joined by Matthijs, of whom I am certain that he could have written the entire compiler himself, had he not been so busy organizing all those large events of his. I am glad I was able to work with him on this project, as he is certainly an enjoyable person to both work, and hang around with. I think we are both happy with the final result, having even been able to go to the official Haskell conference in Edinburgh to present our work. For this success, I most certainly want to thank Jan, for both initiating this great project, and always giving me much welcomed guidance when I was unsure what to work on next.

I also want to thank Bert for always being there to answer questions about VHDL (when I was perhaps too lazy to look for an answer myself), and of course I also want to thank Marco for helping me understand the design of his reduction circuit, and for aiding me in my work in general.

Last, but certainly not least, I would like to thank my mother, for her never ending love, and having always supported me during my studies.

Table of Contents

List of Acronyms

ADT	Algebraic Data Type
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
CAES	Computer Architecture for Embedded Systems
CλasH	CAES Language for Hardware
CPS	Continuation-Passing Style
DSL	Domain Specific Language
EDIF	Electronics Design Interchange Format
EDSL	Embedded DSL
FD	*Functional Dependencies*
FIFO	First In, First Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
GADT	Generalized ADT
GALS	Globally Asynchronous, Locally Synchronous
GHC	Glasgow Haskell Compiler
HDL	Hardware Description Language
IP	Intellectual Property
MoC	Model of Computation
MPTC	*Multi-Parameter Type Classes*
PCB	Printed Circuit Board
RAM	Random-Access Memory
SM$\times$V	Sparse Matrix Vector multiplication
VHDL	VHSIC HDL
VHSIC	Very High Speed Integrated Circuit

Chapter 1

Introduction

A Hardware Description Language (HDL) is any language from a class of computer languages and/or programming languages for formal descriptions of digital logic and electronic circuits. The most famous HDLs are VHSIC HDL (VHDL) [20] and Verilog [19]. These languages are very good a describing detailed hardware properties such as timing behavior, but are generally cumbersome in expressing higher level properties such as parameterization and abstraction. For example, polymorphism was only introduced in the 2008 standard of VHDL [20] and is unsupported by most, if not all, available VHDL simulation and synthesis tools at the time of this writing.

A class of HDLs that does prioritize abstraction and parameterization, are the so-called functional HDLs. Through such features as higher-order functions, polymorphism, partial application, etc. parameterization feels very natural for a developer, and as such, a developer will tend to make a highly parameterized design sooner in a functional HDL than he would in a more 'traditional' HDL such as VHDL. The ability to abstract away common patterns also allows functional descriptions to be more concise than the more traditional HDLs.

Another feature of (most) functional HDLs is that they have a denotational semantics, meaning that we can actually *proof* (with the help of an automated theorem prover) the equivalence of two designs. Though not further explored in this thesis, such equivalence proofs, could be used to *proof* that an highly optimized design has the same external behavior as the simple behavioral design the optimized design was derived from, eliminating the need for the exhaustive testing usually involved in the verification of optimized designs.

Even though the development of functional HDLs started earlier than the now well known HDLs such as VHDL and Verilog, these functional HDLs never achieved the same type of fame: at the time, the ability to make a highly parameterized designs in a natural way, and the ability to abstract common patterns, were not as important as the details you can specify with VHDL or Verilog. However, with the increasing complexity of todays hardware designs, and the amount of effort put into exhaustively testing these designs, the industry might soon recognize the merits of functional HDLs.

1.1 Our new functional HDL: CλasH

The CAES group came with the idea of investigating the use of functional hardware description as both a research platform for hardware design methodologies, but also as an educational tool

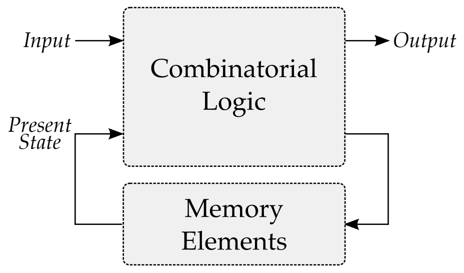

Figure 1.1: Basic Mealy Machine

to be used in practical assignments on hardware designs. The basic premise was that all hardware designs describe the combinatorial logic of a Mealy machine [30]. A graphical representation of a Mealy machine is shown in Figure 1.1. A Mealy machine is a finite state machine that generates an output, and updates its state, based on the current state and the input. We can simulate such a Mealy machine in a functional language, using the straightforward *run* function shown in Code Snippet 1.1. This simulation function basically maps the input over the combinatorial logic using the state as an accumulator.

Code Snippet 1.1 (*Simulation of a Mealy Machine).*

```
run _    _     []               = []
run func state (input : inputs) = output : outputs
  where
    (output, state') = func input state
    outputs          = run func state' inputs
```

So the *func* argument represents the combinatorial logic of the Mealy machine, and the *state* argument of course represents the memory element. So, the actual hardware we will have to describe is the function *func*, whose most basic design can be seen in Code Snippet 1.2.

Code Snippet 1.2 (*Mealy Machine Logic).*

```
func :: InputSignals → State a → (OutputSignals, State a)
func input state = (output, state')
  where
    ...
```

The state of the hardware design is modeled as just a regular argument of the function, and is as such made very explicit. Many existing functional HDLs hide the state within the body of the function; so in this aspect, the functional descriptions we propose really stands out from the rest.

When compared to a more traditional HDL such as VHDL, we can see how the abstraction of state in the proposed functional descriptions allows for a clear synchronous design. In Figure 1.2 we can see the description of a Multiply-Accumulate circuit in both a functional HDL and in the more familiar VHDL. Only by looking at the type of the functional description, it is already clear which part of the design will be part of the state of the circuit. However, if we examine the VHDL description, it is only due to our a-priori knowledge that the *clk* signal will not always have a rising edge, that we can infer that the circuit will have to 'remember' the value of the *acc* signal.

1.1.1 THE SOURCE

Now that the basic idea behind our new functional hardware descriptions is there, we have to think about what kind of syntax, semantics, etc. we want. In this, we have several options: We can

```
macc ::
  (Integer, Integer) →
  State Integer →
  (Integer, State Integer)
macc (x, y) acc = (u, u)
  where
    u = acc + x ∗ y
runMacc = run macc 0
```

```
entity macc is port (clk, resn : in     std_logic;
                     x, y      : in     integer  ;
                     u         : buffer integer  );
end entity macc;
architecture RTL of macc is
  signal acc : integer;
begin
  u   ⇐ acc + x ∗ y
  acc ⇐ 0 when resn = '0' else
        u when rising_edge (clk);
end architecture RTL;
```

Figure 1.2: Multiply-Accumulate: Functional HDL vs VHDL

either define the syntax and semantics ourselves, and write a parser for this language etc. We can also embed it as a Domain Specific Language (DSL) inside another language, where we encode the hardware in custom data-structures. Or, we can use an existing language as a source language and write our functional descriptions in this language.

The first option requires us to write our own parser, type-checker, etc. If we choose the second option we have to write a special interpretation function so that we may simulate the hardware description, and another function to translate it to hardware. For the third option, leveraging an existing language, we can take all the existing tooling (if available) and modify it so that we can translate the Abstract Syntax Tree (AST) of the compiled source to hardware.

Writing our own parser, type-checker, etc. seems too much of an effort when we are still in the exploration phase of our functional hardware designs, so for now, the first option will not be explored any further. The second option, embedding a DSL in an existing language is certainly appealing. We get all the parsing and type-checking for free, as it is provided by the host language, and we get to define our own syntax and semantics. This route has been taken by many other existing languages, some of which are shown in Chapter 2.

The third option, leveraging an existing language, gives us many of the same benefits as the an Embedded DSL (EDSL). The existing syntax and semantics are a double-edged sword of course: all the syntax and semantics is already defined, so we get all that for free. However, some of the existing language elements might have no meaning in hardware, so we have to teach a user of our language not to use those language elements. At the start of this master's assignment we did not yet fully appreciate/understand the merits of embedding a DSL in a host language, so the decision was made to leverage an existing functional language for our hardware descriptions. This has certainly not been a poor choice as we have successfully implemented many aspects of a functional hardware description language. Not only that, with this option explored, we can now also investigate how leveraging an existing functional language compares with the EDSL approach to designing functional HDLs.

There are many functional programming languages to use as the basis for our functional HDL: LISP [40], Haskell [34], ML [31], Erlang [3], etc. Of all these possible possible languages to leverage, we chose Haskell [34]. Even though we did not compare all the options to see which would suit functional hardware specifications the most, Haskell certainly has many features that make it a good choice. It has a strong type system that helps a designer to specify certain aspects of the hardware design upfront, before implementing the body of the function. It was developed to be *the* standard for functional languages. Also for us, as designers of a new functional HDL, there are many benefits: There is a whole set of existing open-source tools and compilers, including the highly optimized flagship Haskell compiler: the Glasgow Haskell Compiler (GHC).

1.1.2 THE TARGET

We do not only want to use our new language to specify and simulate hardware, we also want to generate the actual hardware from our descriptions. Solutions such as programming a Field-Programmable Gate Array (FPGA) directly are not really a sane exercise, so we have to translate our descriptions to a format that allows other tools to do this for us. The most basic (textual) hardware description that every FPGA programming software understands, is a netlist: A format that describes how the basic electronic components and gates are connected to each other. The most common netlist format is Electronics Design Interchange Format (EDIF); actually, its specification covers all aspects of electronic design: schematics, netlists, mask layout, PCB layout etc. The EDIF format is however too low-level for our current needs, which is just being able to synthesize functional specifications so that they can run on a FPGA[1].

As a netlist is too low-level, we will use an existing higher-level HDL that already has available tooling to translate to a netlist format as the target language of our compiler. At the start of this thesis, the higher-level HDL that met this requirement, and that we were most familiar with, was VHDL. VHDL can be synthesized to a netlist format as long as we restrict ourselves to a certain subset of the language. Having VHDL as our target language also gives us the advantage of having access to the optimizations in the existing VHDL synthesis tools. Maybe having an even higher-level HDL (such as BlueSpec [4]) as our target language, would have saved us some translations steps when turning Haskell into this target language. However, it would have taken us considerable time to familiarize ourselves with such a language and the corresponding tools.

As such, the initial goal for the project is set: to design the tools for our new language, which is a subset of Haskell, so that we may simulate our functional hardware descriptions and also translate them to synthesizable VHDL. We call this new language:

CAES Language for Hardware (CλasH)

1.2 Assignment

The original goal of the project soon proved to be too large for one master's assignment, so the work was divided over two assignments. The thesis of Kooijman [26] describes the general translation from Haskell programs to VHDL descriptions. The focus of Kooijman [26] lies on reducing higher-order, polymorphic functions to first-order, monomorphic function and then translating these normalized functions to VHDL. The focus of the work described in this thesis lies on the type aspects of CλasH and the simulation of the hardware descriptions. Also, where the work of Kooijman [26] describes the general translation to VHDL, the work in this thesis describes specific translations concerning certain types, data-structures and the functions on these data-structures.

The reason that this thesis so specifically focuses on types and simulation, is that, even though Haskell has established itself as a successful functional programming language, it still remains to be determined if its properties are equally useful for functional hardware descriptions. A highly regarded property of Haskell is its strong typing system, and this thesis will mostly focus on how we can use this type system to specify the types in our hardware descriptions. Besides being able to translate our descriptions to actual hardware, we of course also want to simulate our descriptions (in Haskell).

[1]As optimization is not a goal of our current language and tooling, it seems wise not to target Application-Specific Integrated Circuits (ASICs) for the time being.

1.3 Overview

This thesis continues in Chapter 2 with discussing existing functional HDLs, naming their merits, problems and solutions. Then Chapter 3 holds the bulk of this thesis, describing the design and implementation of the hardware specific types for our language CλasH. To show what is possible with this first incarnation of CλasH we examine a small case study in Chapter 4. As this is the first version of CλasH we ran into a few problems during its design, we discuss a few of these problems in Chapter 5, ultimately making our conclusions in Chapter 6. Many master's assignments are never complete, always finding new opportunities to improve the original work; this thesis is certainly no exception, so we describe possibilities for future work in Section 6.1.

As supporting material for this thesis, there are also a few appendices. Appendix A shows the VHDL translations for all the functions of our new Haskell vector library. We then have Appendix B, which goes into the details of our automated VHDL test bench generation, which was developed to verify the correctness of the generated VHDL. Appendix C discusses some solution for the node sharing problem encountered in many existing functional HDLs. Appendix D gives a short introduction to some of the GHC extensions to Haskell, that are relevant to this thesis. It is meant for a reader with some experience with functional languages, but not with Haskell and the GHC extensions to Haskell. Readers unfamiliar with Haskell are stressed to read this appendix before continuing with the rest of this thesis. The last appendix, Appendix E, finishes with a CλasH description of a 4-tap FIR Filter, and the corresponding, generated, VHDL code. It is included in this thesis to give the reader an idea of what the generated VHDL looks like.

1.3.1 typesetting

This thesis involves a lot of code snippets, and also references to those code snippets. For this reason, we try to distinguish between types, function, etc. by trying to use different typesetting for each of these elements:

- Function *names* are printed *italic*.
- Type **names** are printed in a **medium bold, sans** font.
- Both *function variables* and *type variables* are printed *italic*.
- Code that is *inlined* in the text is typeset in the same way as the *code snippets*.
- Library names are printed in SmallCaps.

Chapter 2

Domain

Hardware Description Languages (HDLs) have been around for some time, the popular ones: VHDL and Verilog both emerged around the mid 80's. But some functional HDLs, like DAISY [21] and μFP [38] were actually developed earlier. This chapter tries to give a short overview of those earlier functional HDLs, and also of the more recent ones like Lava [8] and ForSyDe [35]. We will touch on their merits and faults, and explain the so-called *node sharing* problem that many languages encountered in their design; solutions to this problem are however reserved for Appendix C. This is done because CλasH differs from many existing functional HDLs, and does not suffer from the *node sharing* problem. Most of the information in this chapter comes from an earlier assignment [5] on the exploration of existing functional HDLs.

2.1 Properties of Hardware Description Languages

The functional HDLs we will see in this chapter are all *structural* descriptions of *synchronous* hardware. Some languages, like CλasH, do however allow some *behavioral* aspects in the hardware descriptions. This section tries to informally introduce the meaning of the earlier mentioned terms, like *structural* and *synchronous*.

2.1.1 structural and behavioral descriptions

For a trivial circuit designs it might suffice draw a transistor layout, but a slight increase in complexity warrants the use of some kind of hierarchy in the design. In the general case a single hierarchy is not sufficient to properly describe the design process. There is a general consensus to distinguish *three* design domains, each with its own hierarchy. These domains are:

- *The behavioral domain*. In this domain, a part of the design (or the whole) is seen as a black box; the relations between outputs and inputs are given without a reference to implementation of these relations. The highest behavioral descriptions are algorithms that may not even refer to hardware that will realize the computation described.
- *The structural domain*. Here, a circuit is seen as the composition of sub-circuits. A description in this domain gives information on the sub-circuits used and the way they are interconnected. Each of the sub-circuits has a description in the behavioral domain or a

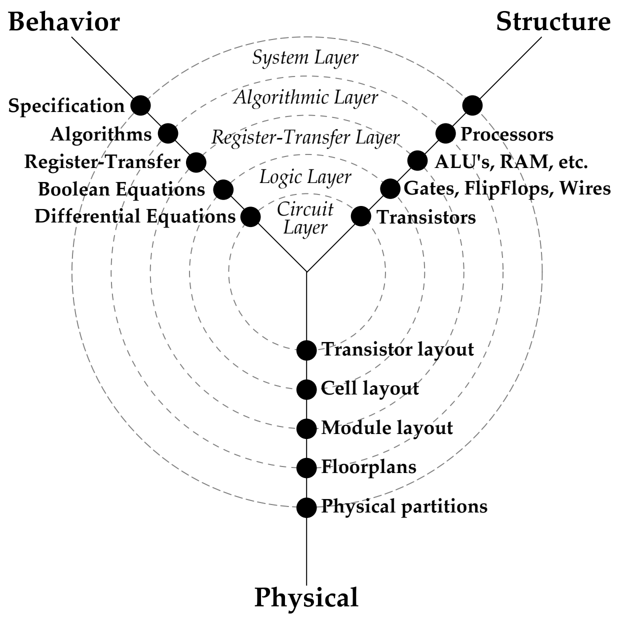

Figure 2.1: Gajski Y-Diagram

description in the structural domain itself (or both). A schematic showing how transistors should be interconnected to form a NAND gate is an example of a structural description, as is the schematic showing how this NAND gate can be combined with other logic gates to form some arithmetic circuit.

- *The physical (or layout) domain.* A circuit always has to be realized on a chip which is essentially two-dimensional. The physical domain gives information on how the subparts that can be seen in the structural domain, are located on the two-dimensional plane. Fore example, a *cell* that may represent the layout of a logic gate will consists of mask patterns that form the transistors of this gate and the interconnections within the gate.

The three domains and their hierarchies can be visualized on a so-called *Y-chart* [11] as depicted in Figure 2.1. Each axis represents a design domain and the level of abstraction decreases from the outside to the center. It was introduced by Gajski in 1983 and has been widely used since then.

Some of the design actions involved in circuit design can clearly be visualized as a transition in the Y-chart either within a single domain or from one domain to another. These are *synthesis* steps; they add detail to the current state of the design. Synthesis steps may be performed fully automatically by some synthesis tools or manually by the designer. Tools that translate designs that are in structural domain to designs in the physical domain are usually called *Place & Route* tools. Large circuit are usually designed in a HDL. These HDLs usually allow for both descriptions in the structural domain and the behavioral domain, sometimes allowing even annotations related to the layout (physical domain). Automated synthesis for the behavioral set of those languages is usually limited or sub-optimal, due to the complexity of the involved problems, such as automated scheduling, and the analysis of complex memory access patterns.

Designs approaches in this thesis

The functional HDL we will see in this chapter all belong to the class of *structural* HDLs, in that they (only) support designs in the *structural* domain. CλasH is currently also a *structural* HDL, though it also has aspects that belong the *behavioral* design domain. For example, it has support for choice elements and integer arithmetic.

2.1.2 SYNCHRONOUS AND ASYNCHRONOUS HARDWARE

The kind of hardware we will deal with in this report is *synchronous* hardware. In synchronous hardware, every component obeys the same omnipresent clock. The semantics of synchronous circuits are quite simple, and can be modeled as functions from input streams, and some state, to output streams. Some *synchronous* circuits have components that run at different, but related speeds, mostly through the use of a clock divider circuit. It is still a *synchronous* circuit as there is still a single clock that dictates all the other clocks. When modeling this kind of synchronous hardware, extra measures have to be taken to properly update the different memory elements.

A more general approach is *asynchronous* hardware, where different components listen to different, unrelated clocks (usually called Globally Asynchronous, Locally Synchronous (GALS)). Some *asynchronous* hardware design have no clock at all. Asynchronous hardware is usually very difficult to reason about, as such, it is hard to find a semantic model which is powerful enough to predict what is going on in the circuit at the electronic level, and simple enough to reason with from the point of view of a designer.

The functional HDLs described in this chapter, like CλasH, can only model *synchronous* circuit where *all* components run at the same clock frequency.

2.2 Existing Functional HDLs

There have been many functional hardware description languages over the years, usually made obsolete by their successor. This section starts with the two predecessors, μFP [38] and Ruby [22], of the still actively researched language: Lava. Lava, the third language we touch on, is one of the most extensively documented functional HDLs, and was the main focus of the individual assignment [5] leading up to this thesis. The fourth and final language is ForSyDe, of which part of its compiler (the VHDL AST) is even used in CλasH. Readers interested in other existing functional hardware description languages are referred to the individual assignment of Baaij [5].

2.2.1 μFP

μFP [38] extends the functional language, FP [6], designed for describing and reasoning about regular circuits, with synchronous streams. μFP advocates descriptions using only built-in connection patterns, also called combinators. An example of such a combinator is a *row* combinator, which, when applied to a list of circuits, for each circuit in the list connects the output to the input of the consecutive circuit. The result of applying this combinator to this list of circuits, is a single larger circuit, which has only one input and one output.

A result of only being allowed to use these combinators is that one is not allowed to give names to intermediate values or wires, which might lead to awkward circuit descriptions. However, according to Claessen [8], a big advantage of this connection pattern style that μFP advocates, is the ease of algebraic reasoning about circuit descriptions: Every built-in connection pattern comes with a set of algebraic laws.

2.2.2 RUBY

The idea of connection patterns was taken further in the relational hardware description language Ruby, which can be seen as the successor of μFP. In Ruby, circuits and circuit specification are seen as relations on streams. Ruby also supports built-in connection patterns that have an interpretation in terms of layout. Like μFP, Ruby descriptions might get awkward because one is forced to use the connection style pattern.

2.2.3 LAVA

Chalmers-Lava

Lava is a hardware description language embedded in the functional language Haskell. There are two versions of Lava in use. The one described in this section, Chalmers-Lava [8] is developed at Chalmers University of Technology in Sweden and is mainly aimed at interfacing to automatic formal hardware verification tools.

Lava facilitates the description of *connection patterns* so that they are easily reusable. Lava also provides many different ways of analyzing circuit descriptions. It can simulate circuits, just as with most standard HDLs, but can also use symbolic methods to generate input to analysis tools such as automatic theorem proves and model checkers. The same methods are used to generate VHDL from the Lava circuit description. To give a better understanding of how those symbolic methods work, an example of a symbolic Signal API is shown in Code Snippet 2.1.

Code Snippet 2.1 (*Symbolic Signals).*

```
data Signal = Var String | Component (String, [Signal])
invert b     = Component ("invert" [b])
flipflop b   = Component ("flipflop" [b])
and a b      = Component ("and" [a, b])
 ...
```

So a signal is either a variable name (a wire), or the result of a component which has been supplied with its input signals. When we build a description out of the above primitive components we are actually building a data-structure that represents a signal graph. As we now have an actual graph, we can use standard graph traversal methods which makes the analysis methods for these descriptions a lot easier to design and implement.

Xilinx-Lava

The Xilinx version of Lava [39] is almost similar to the Chalmers version but focusses more on the Xilinx FPGA products and has been used to develop filters and Bezier curve drawing circuits. It also has support for specifying the actual layout of the different components on the FPGA slices.

2.2.4 FORSYDE

ForSyDe [35] is implemented as an EDSL on top of the Haskell programming language. Its implementation relies on many Haskell extensions, some of which are exclusive to GHC, such as Template Haskell[1]. Two different sets of features are offered to the designer, depending on the signal API used to design the hardware:

Deep-embedded

The deep-embedded signal Application Programming Interface (API), is based on the same concepts as the symbolic methods in Lava: By encoding the signals as data-structures, a traversal of a hardware description will expose the structure of the system. Based on that structural information, ForSyDe's embedded compiler can perform different types of analysis and transformations.

[1]More information about Template Haskell can be found in Section D.5.

It has a back-end for translation to synthesizable VHDL, and also a back-end for simulation. Even though it would be possible to simulate the generated VHDL, instead of simulating the original circuit description, debugging a design using the simulation back-end is most likely faster than generating and simulating the VHDL for each debug iteration. The *deep embedded* signal API only supports synchronous descriptions (or synchronous Model of Computation (MoC), in the ForSyDe terminology).

Shallow-embedded

Shallow-embedded signals are modeled as streams of data isomorphic to lists. Systems built with them are unfortunately restricted to simulation (the traversal algorithms work only on symbolic signals), however, shallow-embedded signals provide a rapid-prototyping framework with which to experiment with different types of MoCs. The models of computation that are supported are the Synchronous MoC, the Untimed MoC, and the Continuous Time MoC. Also, ForSyDe has so-called Domain Interfaces which allow for connecting various subsystems, regardless of their MoC.

2.3 Signals and State

All (functional) hardware description languages have to deal with how to model the electronic signals that will eventually flow through the actual hardware. A complete physical model is often overly complicated, so usually an abstraction of a signal is used. In CλasH, a synchronous HDL, a signal is modeled to have a single steady value for a particular tick of the clock.

Many other functional HDLs are often more data-flow like, in that a signal is modeled as a stream (an infinite list) of values, one for each clock cycle. Instead of thinking of a signal as something that changes over time, it is a representation of the entire history of values on a wire. This approach is efficient for many functional HDL when simulating the hardware, because lazy evaluation and garbage collection combined keep only the necessary information in memory at any time. To give a better feel for this stream-based approach, we see an And-gate as it would be modeled in a stream-like language in Code Snippet 2.2

Code Snippet 2.2 (*AND-Gate in a stream-based approach).*

```
andGate :: [Bool] → [Bool] → [Bool]
andGate a b = zipWith (∧) a b
```

Simulating this And-gate for three clock cycles with signal *a* having the values [*True, True, False*] and signal *b* having the values [*False, True, True*], will give the expected output: [*False, True, False*]. Note that the above lists are finite only for the purposes of presentation.

Almost all functional descriptions of hardware require that each circuit acts like a pure mathematical function, yet real circuits often contain a state as well. To model this state in the stream-based approach we delay a stream by one or more clock cycles. Looking at the external behavior, it now seems as if the circuit description can recollect the state of the signal of one or more clock cycles ago. To give an idea of how this works, we show the description of one of the most primitive stateful components you typically find in hardware, a delay flip-flop, in Code Snippet 2.3[2].

Code Snippet 2.3 (*Stream-based Delay flip-flop).*

```
flipflop :: [Bool] → [Bool]
flipflop a = False : a
```

[2]In many papers on functional HDLs that use *streams* to model signals, this code example is often referred to as a *latch*, which is incorrect, as the stream definition clearly states that it represents signal values for entire clock cycles; latches can change value during a cycle.

This description hard-codes the initial state of this delay flip-flop to *False*, but we could of course make the description parameterized in this aspect.

2.4 Generating Netlists: Problems & Solutions

When a circuit contains a feedback loop, its corresponding graph will be cyclic. For example, take a trivial circuit with no inputs and one output, defined as follows:

Code Snippet 2.4 (*Oscillate circuit).*

$$\begin{aligned}
&\mathit{inv} :: [\mathit{Bool}] \to [\mathit{Bool}] \\
&\mathit{inv}\ a = \mathit{map}\ (\neg)\ a \\
&\mathit{oscillate} :: [\mathit{Bool}] \\
&\mathit{oscillate} = \mathit{flipflop}\ (\mathit{inv}\ \mathit{oscillate})
\end{aligned}$$

Assuming that the flip flop is initialized to *False* when power is turned on, the circuit will oscillate between *False* and *True* forever.

Perhaps the deepest property of a pure functional language is referential transparency, which means that we can always replace either side of an equation by the other side. Now, in the equation:

$$\mathit{oscillate} = \mathit{flipflop}\ (\mathit{inv}\ \mathit{oscillate}) \tag{2.1}$$

We can replace the *oscillate* in the right hand side by any value α, as long as the following holds:

$$\alpha = \mathit{oscillate} \tag{2.2}$$

And we do: the entire right hand side is equal to *oscillate*. The same reasoning can now be repeated indefinitely:

$$\begin{aligned}
\mathit{oscillate} &= \mathit{flipflop}\ (\mathit{inv}\ \mathit{oscillate}) \\
&= \mathit{flipflop}\ (\mathit{inv}\ (\mathit{flipflop}\ (\mathit{inv}\ \mathit{oscillate}))) \\
&= \mathit{flipflop}\ (\mathit{inv}\ (\mathit{flipflop}\ (\mathit{inv}\ (\mathit{flipflop}\ (\mathit{inv}\ \mathit{oscillate}))))) \\
&\ldots
\end{aligned}$$

All of these circuits have exactly the same behavior. But it is less clear whether they have the same structure. Figure 2.2 shows the circuits corresponding to the above equations. So, depending on how many times you want to evaluate the description in Code Snippet 2.4, the corresponding structure might be any of the circuits in Figure 2.2; or even all three circuits in parallel, depending if hardware is generated for each iteration.

It is absolutely essential for a hardware description language to be able to generate netlists. We must find a way to determine if we already visited a node as we traverse the circuit graph, so that we can describe the desired feedback loop. The problem thus becomes that we need to be able to uniquely determine each node in the graph. The problem can be circumvented by only evaluating a function once, at the cost of losing the ability to evaluate recursive functions.

2.4.1 node sharing in cλash

The problem of not knowing the exact structure of a circuit description with a feedback loop does not apply to CλasH. That is because such a feedback is not made as explicit as they are in other existing functional HDLs, which are more data-flow like. Take the CλasH description of the oscillation circuit for example:

$$\begin{aligned}
&\mathit{oscillate} :: \mathit{State}\ \mathit{Bool} \to (\mathit{Bool}, \mathit{State}\ \mathit{Bool}) \\
&\mathit{oscillate}\ (\mathit{State}\ s) = (s, \mathit{State}\ (\neg\ s))
\end{aligned}$$

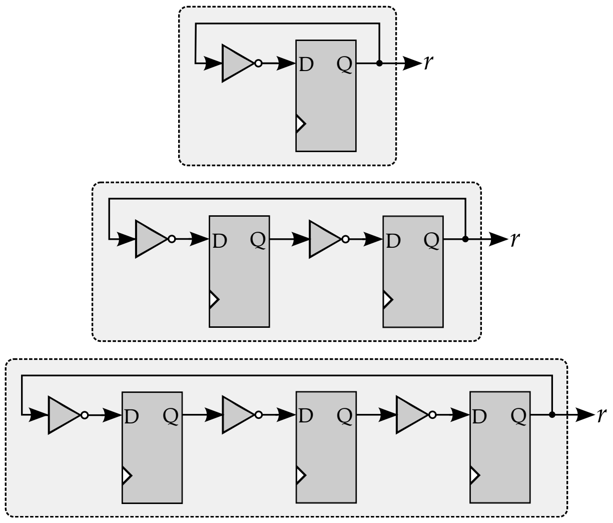

Figure 2.2: Several oscillation circuits

Compared to the general circuit description found in Code Snippet 1.2, we see that this circuit has no input signals. The **State** type indicates that the only input of the function is the state of the function, which is of type **Bool**. The function signature also indicates that the output of the function is of type **Bool**.

To get back to the issue of node sharing in CλasH: The connecting element between the updated state and the present state is not visible on evaluation of the circuit description, so there can be no endless evaluation of a circuit description akin to what we witnessed in the earlier oscillation circuit description. There is no doubt that the CλasH description precisely corresponds to the structure of the top instance of the oscillation circuits portrayed in Figure 2.2. We will only witness endless evaluation of this circuit if the we apply the circuit description to the *run* function found in Code Snippet 1.1.

What this means is that there is no need for CλasH to uniquely determine nodes in the circuit graph to support feedback loops. Actually, the *only* type of feedback loops you can *explicitly* describe in a CλasH function are purely *combinatorial* feedback loops. And since we do not want to make purely combinatorial loops in hardware, there is no reason for a CλasH compiler to support descriptions that have those feedback loops. Many existing functional HDLs do suffer from the node sharing problem however, so compilers of these languages had to find ways to uniquely determine graph nodes. The interested reader is referred to Appendix C to see a number of approaches to uniquely determine these graph nodes.

Chapter 3

Hardware Types

As CλasH is basically just a subset of Haskell which is translatable to VHDL, CλasH gets all the benefits (and burdens) of Haskell's strong type system. Throughout this chapter we discuss whether the type specification needs for a HDL can be met by Haskell's type system. A very important property of types in HDLs is that they are able to specify the size of an object; something we also see in many of the VHDL types, such as **array** and **unsigned**. The reason why size is so important in hardware specifications is that without knowing this property there is no way we can determine the ultimate structure of the hardware. These size-dependent hardware types are part of a larger class of types called *dependent types*.

This chapter begins with an informal introduction of such dependent types in Section 3.1. As Haskell does not have real dependent types we will also see how they can be *faked*. The limits imposed by this *fakery* play an important role in the ultimate design of the fixed-size vector type, which we discus in Section 3.2. We add support for vectors in CλasH as they are an ubiquitous concept to conveniently group elements. So the second section of this chapter discusses the design process behind the current implementation of fixed-size vectors in CλasH, how they work under simulation, and how they are translated to VHDL. As we do not want CλasH to be a purely structural language, we have added support for a very important behavioral concept: integers and their corresponding arithmetic operators. Section 3.3, which is the final section of this chapter, discusses the design of the integer primitives in CλasH, how they function under simulation, and finally how they are translated to VHDL.

3.1 Dependent Types

Concepts, such as *programs*, *programming languages*, *computations* and *types*, are probably familiar to most readers of this thesis. So to make a potentially long story short: Programming languages are used to express computations. Computations manipulate values. Typed programming languages distinguish between *types* and *values*. Types are related to values by a *typing relation* that says what values *belong* to what types, so one usually thinks of *types* as a *set of values*. Expressions, and other program parts, can be *assigned* types, to indicate what kind of values to produce or manipulate. Types can thus be used to document programs (to clarify what kind of values are involved in a certain part of the program) and to help detect programmer mistakes.

In statically typed languages, the types are not seen as something that take part in computations, but rather something that allows a compiler to check that a program is *type correct* without actually running the program. Seeing types as a way to organize values, one can ask the question whether it would be meaningful to have a similar way to organize types? Or to even have values aiding in the organization of types? The answer is yes, and this is where dependent types come in.

Dependent types reflect the fact that validity of data is often a *relative* notion by allowing prior data to affect the types of subsequent data [29]. Types are first class objects in dependent types systems: they may be passed as arguments and computed by functions from other types or from ordinary data. Type-level programming is just ordinary programming which happens to involve types, and the systematic construction of types for generic operations is correspondingly straightforward, as we will see later on in this chapter. This thesis is not the place for a full explanation of dependent types, so a reader in search of (a lot) more detail is referred to works such as those by Barendregt [7] and Luo [27]. However, a more pragmatic approach to understanding dependent types might to experiment with them in a dependently typed language like Agda [32], which has a Haskell-like syntax. The standard example of a dependent type is the type of lists of a given length[1]:

$$\begin{array}{ll} \textbf{data} & \mathit{Vector} :: \mathit{Nat} \rightarrow \mathit{Type} \rightarrow \mathit{Type} \\ \textbf{where} & [\,] \;:: \forall\ (A :: \mathit{Type}).\ \ \mathit{Vector}\ \mathit{Zero}\ A \\ & (:) :: \forall\ (A :: \mathit{Type}).\forall\ (n :: \mathit{Nat}).\ \ A \rightarrow \mathit{Vector}\ n\ A \rightarrow \mathit{Vector}\ (\mathit{Suc}\ n)\ A \end{array}$$

The above states that a **Vector** type is constructed out of a natural number (**Nat**) and another arbitrary **Type**. The *datatype* has two constructors, the empty vector, [ ], which results in a **Vector** with length *Zero*. The second constructor, (:), concatenates an element to a vector of length *n*, resulting in a vector whose length is the **Suc**cessor of *n*. The presence of explicit length information allows us to enforce stricter static control on the usage of vector operations. For example, we ensure that the *tail* operations is applied only to *nonempty* vectors:

$$\begin{array}{l} \mathit{vTail} :: \forall\ (A :: \mathit{Type}).\forall\ (n :: \mathit{Nat}).\ \ \mathit{Vector}\ (\mathit{Suc}\ n)\ A \rightarrow \mathit{Vector}\ n\ A \\ \mathit{vTail}\ (x : \mathit{xs}) = \mathit{xs} \end{array}$$

Programming with dependent types is much less convoluted in practice than it might seem at first glance because the compiler can fill in the details which are forced by the type, such as the **A** and *n* arguments for *vTail*. In addition, the need for 'exception handling' code is greatly reduced: *vTail* has no [ ] case, because [ ] is not in its domain.

3.1.1 DEPENDENT TYPES IN HASKELL

Haskell's developers did not set out to create a type-level programming facility, but non-standard extensions with *Multi-Parameter Type Classes* (MPTC) and *Functional Dependencies* (FD) (and more recently also *Type Families*) nonetheless provide the rudiments of one, albeit serendipitously. The concepts behind *Multi-Parameter Type Classes*, *Functional Dependencies* and *Type Families* are explained in greater details in Appendix D. These extensions to the Haskell *type class* mechanism give us strong tools to relativize types to other types. We may simulate some aspects of dependent typing by making *counterfeit* type-level copies of data, with type constructors simulating data constructors and type classes simulating datatypes.

Unless a reader is already quite familiar with the mentioned Haskell constructs, all of the above will probably sound quite alien. For this purpose we will give a short introduction to type-level programming in Haskell. We do this by first defining a 'familiar' term-level representation and afterwards showing the type-level equivalent. The example will consist of some very basic arithmetic relations, be it that we might use some unfamiliar encoding of natural numbers: Peano numerals.

[1]The reader should note that the example is not Haskell code.

Peano numerals encode natural numbers using the two basic constructs which we saw earlier when we encoded the length property in the dependently typed vector type: The *Zero* construct represents the natural number 0, and the *Successor* construct (of course) represents the successor of a Peano encoded natural number.

To start, we will define these Peano encoded natural numbers (and an abbreviation for a sample number) in Haskell at the *term*-level:

```
data Nat = Zero | Succ Nat
three = Succ (Succ (Succ Zero))
```

The 'counterfeit' *type*-level copy of the above datatype could then be constructed as follows:

```
data Zero
data Succ n
type Three = Succ (Succ (Succ Zero))
```

So where *Zero* and *Succ* were constructors for the **Nat** type in the *term*-level example, they are now *types* in their own right. And the sample number is now also a *type* on its own right (be it that it is 'just' a type *alias*).

Now that we have these natural numbers we want to define a function that tells us if a number is even or odd, at the *term*-level we do that as follows:

```
even Zero     = True
even (Succ n) = odd n
odd Zero      = False
odd (Succ n)  = even n
```

We will now define these *even* and *odd* functions at the type level using Haskell's type-class mechanism. Details of the type class mechanism can be found in Appendix D and will not be elaborated any further in this section. For now, the type class specific syntax should just be seen as the required syntactic sugar for type-level programming. The *type*-level functions are defined as follows:

```
class Even n where isEven :: n
instance           Even Zero
instance Odd n ⇒   Even (Succ n)
class Odd n where isOdd :: n
instance Even n ⇒ Odd (Succ n)
```

The *isEven* and *isOdd* functions specified in their respective classes are defined as a matter of convenience, and could be discarded. We defined these functions so it is easier to ask a Haskell interpreter to check if a number is even (or odd). So using the class functions we can ask a Haskell interpreter to check if the earlier defined *type*-level number **Three** is even or odd:

```
GHCi> :type (isEven :: Three)
*** Error:
  No instance for (Odd Zero)
    arising from a use of 'isEven' at <interactive>:1:0-5
  Possible fix: add an instance declaration for (Odd Zero)
```

We get a type error because three is not an even number. An interpretation of the last line is that if zero were odd, then three would be even.

```
GHCi> :type (isOdd :: Three)
(isOdd :: Three) :: Three
```

The absence of a type error means that three is an odd number.

The given example certainly does not touch on all of the *type*-level programming facilities found in Haskell, nor the simulation issues of dependent types in Haskell; this thesis is not the place for such work. However, there is a lot of excellent material available on these subjects. A good introductory tutorial on type-level programming in Haskell is *Fun with Functional Dependencies* by Hallgren [14]. Readers who are keen on knowing more about the simulation of dependent types in Haskell in general will certainly enjoy reading Conor McBride's article: *Faking It: Simulating Dependent Types in Haskell* [29].

3.2 Fixed-Size Vectors

In general-purpose programming languages, and also HDLs, lists/vectors are used to conveniently group elements, such as bits. In many programming languages we can deal with dynamically sized vectors (e.g. linked lists), or even infinitely large vectors when we apply a lazy evaluation strategy. In HDLs however, both concepts are problematic in their physical realization on hardware. As we do not have an infinite amount of resources, such as floor space, infinite lists that expand in space are out of the question. Infinite lists that expand in time are beyond the scope of this thesis, as CλasH designs can only describe the structural properties of hardware.

To have dynamically sized lists, we would have to reconfigure the layout of the hardware at run-time. With ASICs this is impossible; and even though some FPGAs do allow for runtime reconfiguration, it is virtually impossible to use this feature for scaling purposes such as dynamically sized lists. The reason being of course that it is hard, or even impossible, to determine beforehand the upper bound of the required floor space for the dynamically sized list. Also, we would need to design dedicated hardware on the FPGA that will do all this runtime reconfiguration while the chip is running.

In the end this means that it is paramount for a HDL to support fixed/statically sized lists, from here on called fixed-size vectors. Recognizing this almost obvious need for fixed size vectors, and having Haskell's type system at our disposal, we would of course like to specify the size at the type level. There are two very important reason why we would like to specify the exact length of a fixed-size vector at the *type* level (information available at compile-time), and not at the *term* level (information usually only available at run-time):

- When the exact length of a vector is specified at the *type* level, it is statically available at compile-time. This makes the VHDL translation of the vector type and the operations on vectors very straightforward, as VHDL also needs the length of its arrays to be specified as part of their type. If the size of the vector where to be specified at the *term* level, the compiler will need to do a lot of partial evaluation and bookkeeping to know the exact length of the vector at any time in the compilation process. This becomes even harder when vectors can change length by a variable amount. Also, when we have a purely combinatorial circuit with a vector as one of its inputs there is the problem of the inability to specify the size of this input vector at the *term* level. We would have to take special matters, such as specifying a compiler pragma, to let the compiler know how big the input vector is.
- The type checker will help the engineer in designing correct hardware, as he can specify, in the signature of the function, what the length of the input vectors should be, and what the length of the output vectors will be. This way you do not have to do an exhaustive simulation to find conflicting vector sizes, as they will be caught at compile time.

In Haskell we can easily specify function signatures for functions that work on unconstrained vectors (implementation details omitted for purposes of presentation):

$$\mathit{head} :: \mathit{Nat}\ n \Rightarrow \mathit{Vector}\ n\ \mathit{Int} \rightarrow \mathit{Int}$$
$$\mathit{head} = \ldots$$

Where the type variable n indicates the length of the vector; note that by specifying the context $Nat\ n$ we try to indicate that the variable n is some sort of natural number. Ideally we would then like to have something that allows us to specify a vector with a specific length along the lines of:

$$v :: Vector\ 23\ Int$$

Where we say that v is of a **Vector** type containing 23 values of type **Int**. Alas, it is not possible to directly specify the above in Haskell: the *term*, 23, is not allowed on the right side of the double colon (::); only *types* may venture there. As we saw in the previous section however, we can *simulate* dependent types in Haskell by making a *type*-level copies of data/terms. So in a way, we can have something quite similar to the defined **Vector** type. We will just use a *type-level numeral*, akin to natural numbers from the previous section, to parameterize the vector.

Beside the need to specify the size of the vectors, we also want to specify transformations on the size of the vectors, or how the size of two vectors relate to each other. Because the size of the vectors is a type, the operations will of course also have to work directly on types. For example, we want to be able to specify that the size of a vector resulting from the concatenation of two vectors is the combined size of the two input vectors:

$$(+\!\!+) :: Vector\ s1\ a \rightarrow Vector\ s2\ a \rightarrow Vector\ (s1 + s2)\ a$$

We start our investigation with an existing library that already has all of the above features. This fixed-size vector library, called FSVec [1], has been developed as part of an existing functional HDL mentioned in Chapter 2: ForSyDe [35].

3.2.1 FSVEC LIBRARY

The FSVec library uses the type-level numerals from the TYPE-LEVEL library [2] to indicate the size of the vector at the type level. This type-level numerals library uses *Multi-Parameter Type Classes* and *Functional Dependencies*[2] to specify the relations between the numerals, and the operations on them, like the summation (+) operator in the type signature of the above (++) function. In this subsection we will deal with the particulars of the FSVec library and the TYPE-LEVEL library at the same time as we explore the functionality of FSVec. We start with the datatype that represents the fixed-size vector:

$$\textbf{newtype}\ Nat\ s \Rightarrow FSVec\ s\ a = FSVec\ \{unFSVec :: [a]\}$$

The **newtype** keyword indicates that the datatype definition is actually a datatype renaming from the list type, **[a]**, to **FSVec s a**. In this sense it is not a 'true' datatype; the exact details of **newtype** declarations can be found in Appendix D. The context of the datatype, the part before the ⇒ symbol, means the following: *Nat s* implies that the type variable *s*, the size of the vector, is a numeral that belongs to the set of natural numbers. The natural number, *s*, is thus not a *term literal*, but a *type*. For example, the number two is represented by the *type* **D2**, where the **D** indicates that is is a decimal representation[3]. If you want to use these *type*-level numerals in the body of a function, you have to use some sort of *term*-level representations, as types are only allowed in a signature. The TYPE-LEVEL library therefor has *term*-level aliases for all its *types*; for example, the term-level representation of the type **D2** is *d2*, whose actual *value* is $\perp$.

The symbol $\perp$ is pronounced 'bottom', and refers to a computation which never terminates. It is a 'value' with no information: as such, it can be of any type. All the other theoretical aspects of the $\perp$ element are beyond the scope of this thesis. We use it here, and in other examples, when we need to specify a term of which we do not know the value, but also where the value is of no use

[2]Both *Multi-Parameter Type Classes* and *Functional Dependencies* are explained in Appendix D.

[3]As apposed to a Binary (**B**), Hexadecimal (**H**) or Octal (**O**) representation that are also present in TYPE-LEVEL.

to us. In this case, we only care about the type information, so there is no need for knowing the actual value.

Having covered the context of the datatype declaration, let us go on with explaining the type declaration, the part between the $\Rightarrow$ symbol and the $=$ symbol: The name of the type is **FSVec** and it has two type variables, s and a, which represent the size and the element type respectively.

Then follows the constructor part of the datatype declaration, the part after the $=$ symbol. Actually, we declare both the constructor and de-constructor at the same time. We call them constructors and de-constructors here, but they are actually renaming constructs: As said earlier, a **newtype** declaration is datatype renaming. So what this means is that the 'constructor', *FSVec*, retypes its argument, which had type **[a]**, to be of type **FSVec s a**. That is why the size argument, s, is not part of the 'constructor' (even though you might think it should be): The *FSVec* 'constructor' is *just* a *retyping* construct. The curly braces following the *FSVec* constructor, $\{\ldots\}$, are not meant to be read as the Haskell record syntax, but as the syntactic sugar to introduce the de-constructor, *unFSVec*. This 'de-constructor' *retypes* from type **FSVec s a** to type **[a]**. Again, a more detailed explanations about *newtype* declarations in general can be found in Appendix D.

Safe and Unsafe Constructors

We call the constructor of **FSVec** introduced by the *newtype* declaration *unsafe*, as it can not give any static guarantees about the list that is given as the argument for constructing the **FSVec** vector. This means that the type checker can not guarantee that the actual number of elements inside the vector, is the same as is indicated by its type. For example, the following code is allowed by any Haskell compiler:

$$\begin{aligned}&v :: \mathit{FSVec}\ \mathit{D2}\ \mathit{Int}\\ &v = \mathit{FSVec}\ [1,2,3]\end{aligned}$$

So there is now a mismatch between the number of elements in the vector (what we call the *dynamic* length), and the length indicated by the type (what we call the *static* length). For this reason, the constructor exposed by the *newtype* declaration is explicitly hidden by the designers of the FSVec library, so that users can not create invalid vectors by accident. With the original constructor hidden, the FSVec library exposes several other ways to construct a fixed size vector. This way, a user has to explicitly choose either a *safe* (we call it *safe* when the user of a library can *only* create vectors that have a matching *dynamic* and *static* length) or *unsafe* constructor.

We can, for example, unsafely build a vector from a list using *unsafeVector*, which has two arguments: A type-level numeral indicating the static size of the list, and a list containing the elements. It is unsafe because, like the original constructor of *FSVec*, the static size of the vector and dynamic size of the list can differ. However, the function *unsafeVector* does, at runtime, check for mismatches between the *dynamic* and the *static* length of the vector and reports an error if this happens to be the case (something the original *FSVec* constructor could not do):

```
GHCi> unsafeVector d2 [1,2,3]
*** Exception: Data.Param.FSVec.unsafeVector: dynamic/static length mismatch
```

As we mentioned earlier, the designers of the FSVec library also implemented *safe* constructors. An example of such a *safe* vector constructor is the Template Haskell function *vectorTH*, which has the following signature:

$$\mathit{vectorTH} :: \mathit{Lift}\ a \Rightarrow [a] \rightarrow \mathit{ExpQ}$$

It takes a *list* as its argument, and turns it into an AST that represents the Haskell code for the fixed-size vector. Some explanation about Template Haskell is certainly in order here, as it should give some insight as to how the *vectorTH* function works, and why it is considered *safe*.

Template Haskell provides the ability for a Haskell program to perform computations at compile time, which generates new code that can then be spliced into the program. Splicing is the act of inserting a generated AST in the AST of the original program. Template Haskell defines a standard algebraic data type for representing the abstract syntax of Haskell programs, and a set of monadic operations for constructing programs. These are all expressible in pure Haskell. Two additional syntactic constructs are introduced:

- A quotation construct, ⟦...⟧, that gives the AST representation of the fragment of code within the brackets.
- A splicing construct, $(...), that takes a code representation tree (AST) and effectively inserts it into a program.

In Template Haskell, all aspects of Haskell which the ordinary programmer can use are also available to process the AST at program generation time. Thus a function that works on these ASTs, e.g. *vectorTH*, is just an ordinary Haskell function definition.

When we look at the signature of the *vectorTH* function we see that the result is of type **ExpQ** (an alias for **Q Exp**), which indicates that it is a (monadic) function which returns the AST for a piece of Haskell code. The elements of the list argument have to be of the **Lift** class, so that the *vectorTH* function can get the AST representation of the elements.

The reason that this *vectorTH* function can safely construct a vector from a list, is that we can just ask for the length of the list at compile-time and generate the corresponding vector type. We could not do this with the *FSVec* constructor, as we needed to specify the exact vector type (and thus its length) up front. The resulting AST of the *vectorTH* function is of no use to us, we want the actual vector. So, using the $(...) syntax, we can splice the generated AST of the vector into a program, and resume compilation. When we ask for the type of the spliced *vectorTH* function, we see that the vector has a *generated* static length that corresponds with the number of elements in the list, meaning that the vector was constructed *safely*:

```
GHCi> :type $(vectorTH [1::Int,2,3])
$(vectorTH [1::Int,2,3]) :: FSVec D3 Int
```

How type-level vector sizes help a hardware designer

Using one of the *safe* constructors, we will examine an example function and show how the type checker can help us ensure that the vector length we want is the vector length we get. Code Snippet 3.1 shows a function *foo* that, according to its type signature, only accepts vectors of size two, and always outputs a vector of size four. The body of the function however does not do what the signature promises[4].

Code Snippet 3.1 (*Vector concatenation - Incorrect).*

```
foo :: FSVec D2 Int → FSVec D4 Int
foo x = out
  where
    y   = copy d3 0
    out = x ++ y
```

Looking at the body of the function we see a new function, *copy n a*, which creates a vector containing *n* copies of element *a* (the **Int**eger 0 in the example), where *n* is a type-level numeral. The function ++ is the vector concatenation function equivalent to the one for lists. The above code does not compile (type-check), because the body of the function does something different than expected. The compiler reports the error shown on the next page:

[4]Or, as seen from the other side of the fence: The signature lies about what the body does.

```
Couldn't match expected type 'D4' against inferred type 'D3'
  ...
  arising from a use of '++' at foo.hs:16:10-15
```

The reported error is slightly confusing, especially given that it rises from the use of $+\!\!+$. It requires an in-depth knowledge of the type-level numerals of the TYPE-LEVEL library to understand why the error was reported as it was above, and is beyond the scope of this thesis. Nonetheless, the compiler informs us that it can not match the length of two vectors. The reason being of course that concatenating the vector x (size 2) to the vector y (size 3) results in a vector of size 5. However, the type signature of *foo* promised that the output would be a vector of size 4. Depending on what we want, we can either change the body, or, change the type signature of the function, to fix the error.

Problems with FSVec

Having a potential fixed-size vector library for CλasH in the form of the FSVEC library, one would think we could start focussing on the translation of the vectors and vector functions to VHDL. Instead, due to certain problems with the TYPE-LEVEL library that FSVEC uses, we are forced to search for a new type-level numerals library first. Below we see one of many examples where the TYPE-LEVEL library makes it impossible for us to describe an (intuitive) type signature for a function.

Code Snippet 3.2 shows a simple rotation function, with its intuitively correct (but sadly incomplete!) type signature.

CODE SNIPPET 3.2 (*Rotate in FSVec).*

```
rotate :: Pos s ⇒ FSVec s a → FSVec s a
rotate vect = (last vect)+>(init vect)
```

Some explanation for the code snippet is in order: The context of the signature, *Pos s*, tells us that the size, *s*, of the vector should be non-zero; it has to be provided because the *last* and *init* function are only defined for non-empty vectors. The function signature, $FSVec\ s\ a \rightarrow FSVec\ s\ a$, then tells us that the vector length should stay the same. The operator +>, is equivalent to the *cons (:)* operator for lists. When we try to compile the *rotate* function we get the following error:

```
Could not deduce (Data.TypeLevel.Num.Ops.IsZero s yz, DivMod10 s yi yl)
  from the context (Pos s)
  arising from a use of 'init' at rotate.hs:25:30-38
Possible fix:
  add (Data.TypeLevel.Num.Ops.IsZero s yz, DivMod10 s yi yl)
    to the context of the type signature for 'rotate'
  ...
```

The above error apparently rises from the use of *init*:

$$init :: (Pos\ s, Succ\ s'\ s) \Rightarrow FSVec\ s\ a \rightarrow FSVec\ s'\ a$$

The $Pos\ s$ part of the context should be familiar to us now, the $Succ\ s'\ s$ specifies that s is the successor of s': Meaning that the *init* function returns a vector that is one size smaller than the vector it was given as its arguments. Still this gives us no clue as to why the dependencies on the **IsZero** and **DivMod10** class are suddenly exposed in the error message. Most likely they are part of some of the induction rules related to **Succ**.

As **IsZero** and **DivMod10** are part of the internal induction rules they are hidden by the TYPE-LEVEL library (meaning that they can only be used by functions inside the TYPE-LEVEL library), making it impossible to implement the suggestion given by the error message. The problem lies with the TYPE-LEVEL library, not the FSVEC library itself: Meaning that we only need to find a replacement for the type-level numerals and keep most of the functionality of the FSVEC library to be used as the vector library for CλasH.

As the TYPE-LEVEL library is not the only existing type-level numerals library in existence, we do not have to write our own numerals library just yet. Another type-level numerals library, called TFP [12], has similar signatures as the TYPE-LEVEL library, but uses *Type Families* instead of *Multi-Parameter Type Classes* and *Functional Dependencies* to describe the relations between numerals and the operations on them. In the next subsection, we will see that a fixed-size vector library using the numerals from the TFP library allow us to write a larger number of functions when compared to the FSVEC library.

3.2.2 VECTOR LIBRARY WITH TYPE FAMILY BASED NUMERALS: TFVEC

Due to the problems with the numerals of the TYPE-LEVEL library described in the previous subsection we have completely re-written the original FSVEC library to make use of the type-level numerals and operations from the TFP library. We call this new library TFVEC: *Type Family Vector* library, as it uses type-level numerals based on *Type Families*.

In Code Snippet 3.3 we can see a new description of the *rotate* function that we tried to describe earlier using the FSVEC library. Although the type context is certainly larger than that of the *rotate* function found Code Snippet 3.2, it compiles, and also behaves correctly.

CODE SNIPPET 3.3 (*Rotate with TFVec).*

```
rotate :: (PositiveT pT, NaturalT nT, nT~Pred pT, pT~Succ nT) ⇒
  TFVec pT a → TFVec pT a
rotate vect = (last vect)+>(init vect)
```

Let us examine the new syntax first: The $\sim$ operator in the context of the type signature. This operator, $\sim$, is the *type equality coercion* operator, and was introduced together with the *type families* extension to Haskell [41]. The $\sim$ operator asks the type-checker to enforce that the type on the left-hand side of the operator is 'equal' to the type on the right-hand side. We put 'equal' between quotes to indicate that the types are not intentionally *equal*, rather that, if the types were to be erased the program would not 'go wrong'.

The context of the above type signature is larger than expected, let us examine it in greater detail:

- *PositiveT pT*: The variable *pT*, the size of the vector, is a positive number; this context must hold because the functions *last* and *init* are only defined for vectors whose size is larger than zero.
- *NaturalT nT*, *nT*$\sim$*Pred pT*, *pT*$\sim$*Succ nT*: Specifies that the successor of the predecessor of the variable *pT* is again *pT*. Currently, there seems to be no way in GHC to specify this relation at a higher level[5]. This rule is needed because GHC infers that the type of the body of the function is: *TFVec pT a* → *TFVec* (*Succ* (*Pred pT*)) *a*. Without the context, the type checker can not reduce (*Succ* (*Pred pT*)) to *pT*, and as such not determine that the type of the body is equal to the function signature.

As we will see later on, many more arithmetic relations, some of which most people take for granted (such as the commutativity of addition (+)), have to be *explicitly* specified by a developer! They can not be *implicitly* deduced by a compiler (like GHC)! The *'technical'* reason why GHC will not

[5]There is however research on specifying invariants at the *type*-level [36]

reduce the above arithmetic relation will be discussed below. The *'theoretical'* reason why GHC can not automatically deduce arithmetic relations will be discussed in a later subsection.

Now, the technical reason why GHC would not automatically reduce *Succ* (*Pred pT*) to *pT* is that the type-checker has not been given the 'rules' to execute this reduction. In the *Type Family* system, *Succ a* is defined as a relation for *all* types *a*. We need to define so-called *Type Instances* for each *specific* type so that the type-checker knows what to do with the relation *Succ a* when applied to that type. For example, the *Type Instance* for *Succ a* on the **D1** type could be specified as follows:

type instance *Succ D1* = *D2*

This means, that the type checker now knows that it can reduce the instance of *Succ D1* to *D2*. In the TFP library, **D1**, **D2**, etc. are actually convenient type *aliases* for more 'complicated' underlying types. So in the TFP library, the *Type Instances* for those relations, such as **Succ**, are actually specified on the 'complicated' underlying types. However, the type instances for those relations work on the type *aliases* as well, as type *aliases* are transparent for the type-checker. In effect, the type-checker can reduce *every* instance of the *Succ* relation for *every defined*[6] decimal *a* to the corresponding successive decimal *b*. This also applies to all the other type-level relations and operations: All *specific* decimal instances can be reduced to a normalized form. But, we can not reduce the *general* case of the relations and operations, as we can not specify any rule (*Type Instance*) for this. That is the reason why we had to supply the type equality coercion in the context of the new *rotate* function, so that the type-checker now knows that the 'rule': the successor of the predecessor of *a* is again *a*, applies to the *general* case in that specific function.

When we look at the context of the *rotate* function it is certainly not as concise as we hoped for. But, it is at least *possible* to specify a context that is satisfactory for the type-checker. Hopefully we can lift the context (*Succ* (*Pred pT*))$\sim$*pT* to a higher level, like maybe a type class or type-level invariant, in a future release of GHC[7].

Another bonus of using the TFP type-level numerals, and their operators, are the clearer error messages. If we refactor the function *foo* function from Code Snippet 3.1 to use the new TFVec vector library and try to compile it we get the following error:

```
Couldn't match expected type 'Dec4' against inferred type 'Dec5'
  Expected type: TFVec D4 Int
  Inferred type: TFVec (D2 :+: D3) Int
In the expression: out
    ...
```

The new error message is much clearer about how the expected and inferred lengths vector actually differ, when you compare it to the error given by the compilation attempt of the original function *foo* of Code Snippet 3.1.

TFVec and pattern matching

Recursive functions on the default lists in Haskell often exploit pattern matching, such as matching on either the empty list, or a non-empty list. A very familiar function on lists, *map*, could for example be specified as shown in Code Snippet 3.4.

Code Snippet 3.4 (*Map over lists).*

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

[6]The TFP library has generated aliases, using Template Haskell, for type-level numerals between -10000 and +10000.
[7]The stable release of GHC at the time of this writing is 6.10.4.

The *map* function from Code Snippet 3.4 can not be specified in the same recursive manner with vectors from either the FSVec library or TFVec library. The root of the problem lies in its non-recursive constructor, shown below:

$$\mathbf{newtype}\ (NaturalT\ s) \Rightarrow TFVec\ s\ a = TFVec\ \{unTFVec :: [a]\}$$

The implication of the above constructor is that there is only *one* constructor, and so it is used to construct both empty vectors and non-empty vectors. This is very unlike the familiar Haskell **list** type, which has *two* constructors, [] and (:). This single constructor of the **TFVec** vector makes it impossible to implement a simple *map* function *without* deconstructing the vector to a list. For example, the definition of *vmap* found in the TFVec library, internally uses the *map* function defined for lists:

$$\begin{aligned}&vmap :: (a \to b) \to TFVec\ s\ a \to TFVec\ s\ b\\&vmap\ f\ (TFVec\ xs) = TFVec\ (map\ f\ xs)\end{aligned}$$

However, as we explained earlier, the original *TFVec* constructor is hidden from the eventual user (it is only available within the library): We, the developers of TFVec, did not want a user to be able to access the internal list representation of the vector as he could then accidentally mismatch the *dynamic* and *static* length of the vector. Also, allowing functions defined on lists would mean that the CλasH compiler suddenly needs to support dynamic lists, something which does not fit in the timeframe of this thesis.

Even if we have access to the internal list (remember that the **TFVec** constructor is actually hidden from users), but disallow any operations on lists, we run in to problems if we try to define a *map* function similar to the one in Code Snippet 3.4:

Code Snippet 3.5 (*Map over Vectors - Failed attempt).*

$$\begin{aligned}&vmap :: (a \to b) \to TFVec\ s\ a \to TFVec\ s\ b\\&vmap\ f\ (TFVec\ [\,])\qquad\ \ = TFVec\ [\,]\\&vmap\ f\ (TFVec\ (x : xs)) = (f\ x) {+}{>} vmap\ f\ (TFVec\ xs)\end{aligned}$$

The problem we run into with the *vmap* function from Code Snippet 3.5 is that we do not know the *static* length of the vector returned by the recursive *vmap* call, as we do not know the *static* length of the expression $(TFVec\ xs)$. So, given that the signature of +> is:

$$({+}{>}) :: a \to TFVec\ s\ a \to TFVec\ (Succ\ s)\ a$$

Then the only way for the type signature of *vmap* to hold is when the following is true:

$$Succ\ s \sim s$$

This needs to be true because the *vmap* function, given a vector of length *s* returns a vector of length *s*, but the +> function returns a vector with a length of *Succ s* given a vector of length *s*. Obviously our arithmetic relations would become inconsistent if, $Succ\ s \sim s$, was true. We should therefor *not* specify it as part of the context of *vmap*. The result is that the definition of *vmap* in Code Snippet 3.5 is invalid and can not be corrected.

As we still want a developer to be able to exploit pattern matching for his recursive functions we will explore a new approach to fixed-sized vectors. This new approach will use a recursive datatype (like the familiar list type) where the constructor for an empty vector is distinct form the constructor of a non-empty vector, while still be encoding the size of the vector inside the type. A first attempt at this new vector description is made in the next subsection.

3.2.3 NEW FIXED-SIZE VECTOR WITH PATTERN MATCHING

One approach to constructing fixed-size vectors (instead of the single constructor *newtype* approach) is a Generalized ADT (GADT), where we use Peano numerals to indicate the size. The GADT shown in Code Snippet 3.6 is the Haskell equivalent of the dependently typed vector we saw in Section 3.1. As Haskell does not have real dependent types, we use the Peano encoded type-level natural numbers from Section 3.1 to encode the size of the vector.

Code Snippet 3.6 (*Fixed Size vector as a GADT).*

```
data Zero
data Succ n

data Vector n e where
  Nil  :: Vector Zero a
  Cons :: a → Vector n a → Vector (Succ n) a
```

The first two lines of Code Snippet 3.6 repeat the definition for the Peano-encoded numerals. We can't make these type-level numerals *safe*: The argument *n* of the *Succ n* datatype can not be restricted to either *Succ n* or *Zero* by Haskell's type system. We, as developers, have to keep ourselves in check when we use these Peano numerals and not generate inconsistent numerals by accident.

Then we see the **Vector** GADT, which is parametric in its size and element type. The GADT has two constructors:

- *Nil*, which creates an empty vector (its size is **Zero**).
- *Cons*, which has two arguments, the first argument being the new element that we want to add, and the second argument being the existing vector we want to add this new element to. It results in a vector whose size is the successor (**Succ n**) of the size of the vector that was passed as an argument.

Using this new vector implementation we can define the *vmap* function similar to the *map* function for lists we saw in Code Snippet 3.4. We can see this new *vmap* function in Code Snippet 3.7.

Code Snippet 3.7 (*Map over fixed-size vectors).*

```
vmap :: (a → a) → Vector s a → Vector s a
vmap f Nil          = Nil
vmap f (Cons x xs) = Cons (f x) (vmap f xs)
```

Another function we can describe in a recursive manner is the *append* function (++), which takes two vectors and appends them. The code is shown in Code Snippet 3.9. But before we do that we define the type-level addition operation on Peano-encoded numerals in Code Snippet 3.8.

Code Snippet 3.8 (*Addition of two Peano-encoded numerals).*

```
type family x + y
type instance Zero + y     = y
type instance (Succ x) + y = Succ (x + y)
```

So the first line, **type family** $x+y$, defines the existence of the *type*-level + operator. The second and third line specify the instances of the + operator for the Peano-encoded numerals. The instances themselves should be straightforward in how they encode addition (+).

We need this + operator to be able to specify that the size of the resulting vector of the *append* function is the size of the two input vectors combined.

Code Snippet 3.9 (*Appending two fixed-size vectors*).

```
append :: Vector s1 a → Vector s2 a → Vector (s1 + s2) a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Having this new GADT-based vector, it seems as though we can at least have the same functionality that we had with the vectors from the TFVec library. However, we now have the additional ability to define pattern-match based recursive functions on fixed-size vectors.

Problems on the horizon

Even though the use of GADTs and type families seems to solve all our recursive function definition problems at first, there are now other problems that arise. A simple function, *merge*, makes us reconsider if type-level programming in Haskell is truly ready to be combined with generally recursive functions. The function *merge* interleaves the elements of two vectors, and the description is very similar to that of the *append* function. It is shown in Code Snippet 3.10. We purposely made the description of *merge* very similar to *append*, to highlight the problem at hand.

Code Snippet 3.10 (*Merging two fixed-size vectors*).

```
merge :: Vector s1 a → Vector s2 a → Vector (s1 + s2) a
merge Nil         ys = ys
merge (Cons x xs) ys = Cons x (merge ys xs)
```

If you scan over the code too quickly you would not even notice the difference between the *append* and *merge* function, except for the name of course. The difference lies in the reversal of the arguments in the recursive call of *merge*.

Unlike the *append* function, *merge* will not compile. In the second clause of *merge*, the GADT pattern match exposes the equality $s1 \sim Succ\ s1'$, where *s1'* is the length of *xs* variable. Now, the type checker expects the resulting vector of the expression *merge ys xs* to have the length $s1' + s2$, based on the definition of the $(+)$ operator in Code Snippet 3.8. However, it infers, based on the signature of *merge*, the length $s2 + s1'$. Hence, for this code to type check, commutativity of addition $(+)$ must hold:

$$\forall s1.\forall s2.(s1\ +\ s2)\ \sim\ (s2\ +\ s1) \tag{3.1}$$

While we may think that this equation readily holds for any *s1* and *s2*, this is not so. Due to the openness of the type family **Add** type, one may add at any time an additional type instance, e.g.,

```
data K
type instance K + Zero = Zero
```

such that $K + Zero \not\equiv Zero + K$ and Equation (3.1) no *longer* holds. This is the theoretical reason why GHC can not just automatically infer 'reduction rules' for type families, something we saw earlier when we had to define the type context, $Succ\ (Pred\ s) \sim s$, for the *rotate* function (Code Snippet 3.3).

Invariants as term-level functions

Haskell has no support for invariants, such as the commutativity of addition, at the *type* level. However, it is still possible to implement invariants at the *term* level, so as to make the *merge* function type-check. This solution comes from a paper by Schrijvers et al. [36], which actually describes a language extension to Haskell to support type invariants. We first need to reify coercions at the *term* level using a GADT which witnesses the equivalence of two types, as well as reify the types (such as length annotations) using singleton types (Code Snippet 3.11).

Code Snippet 3.11 (*Equality witness of two types).*

```
data Eqv s t where
  Eqv :: (s∼t) ⇒ Eqv s t
data Nat n where
  Nz :: Nat Zero
  Ns :: Nat n → Nat (Succ n)
```

An invariant such as commutativity of addition is then implemented as a function that analyzes the term level representatives and constructs the proof (Code Snippet 3.12).

Code Snippet 3.12 (*Proof construction for commutativity of addition).*

```
comm :: Nat x → Nat y → Eqv (x + y) (y + x)
comm Nz Nz          = Eqv
comm Nz (Ns y)      = case addZ y of Eqv → Eqv
comm (Ns x) Nz      = case addZ x of Eqv → Eqv
comm (Ns x) (Ns y) =
  case comm x y of
    Eqv → case (comm (Ns x) y, comm x (Ns y)) of
      (Eqv, Eqv) → Eqv
```

where *addZ* function similarly implements an auxiliary invariant stating that:

$$\forall n.\ n\ +\ Zero\ \sim\ n. \tag{3.2}$$

To apply the invariant and have the *merge* function function type-checked, the length of the vectors must be computed separately. The new *merge* function function that takes the commutativity of the addition of the vector lengths into account is shown in Code Snippet 3.13.

Code Snippet 3.13 (*Merge refactored - taking commutativity of addition into account).*

```
merge :: Vector s1 a → Vector s2 a → Vector (s1 + s2) a
merge Nil ys = ys
merge (Cons x xs) ys =
  case comm (length xs) (length ys) of
    Eqv → Cons x (merge ys xs)
length :: Vector s1 a → Nat s1
length Nil          = Nz
length (Cons x xs) = Ns (length xs)
```

Of course, implementing invariants in this way has a cost at runtime, and since Haskell does not enforce that such a 'proof' covers all the cases and will not loop indefinitely, we would need to rely on an external verifier to check the totality of that function. Also, we could have written the *comm* function as shown below, which does not *prove* commutativity at all. The *unsafeCoerce* function[8] will just *lie* about the commutativity (even though it sometimes holds).

```
comm :: Nat x → Nat y → Eqv (x + y) (y + x)
comm _ _ = unsafeCoerce Eqv
```

[8]The highly unsafe primitive *unsafeCoerce* converts a value from any type to any other type.

3.2.4 CHOSING A VECTOR LIBRARY

This section set out with the goal to make a fixed-size vector type, where the length is a parameter of the vector type. This led to the creation of the TFVec library, which has a myriad of functionality. However, realizing we could no longer pattern match on either empty or non-empty vectors, and thus lacking the means to specify constructor-based recursive functions, we explored the use of representing a fixed-size vector using two constructors. Sadly, the current state of type-level programming in Haskell prevented us from achieving general recursion that was easy to use: Needing to prove such basic things as the commutativity of addition would probably be a first of many hurdles to take.

We chose not to explore a GADT based vector library for CλasH any further for two reasons: The first one being that even though describing invariants such a commutativity, distributivity, etc. for the type-level operations would be a useful exercise indeed, it does not fit into the timeframe of this master's thesis. The second reason is that, for the time being, CλasH is mostly meant as a rapid prototyping language. Therefore we do not want to confront users with cumbersome ways to incorporate the proof builders, like the example shown in Code Snippet 3.13. Preferably we do not want to confront them with proofs at all.

The reason why we do not run into the kind of problems like proving commutativity, with the TFVec library lies in its implementation. Most functions of the TFVec library unpack the vector to a list, apply the equivalent list-transformation, and pack the list into a vector of the correct size again. The safety of the library can thus not be guaranteed by the compiler. The user will have to put his faith in us, the designers of the TFVec library, that all the functions behave correctly.

It would be an option to expose these unsafe constructors and deconstructors to the eventual user of CλasH and the TFVec library. We chose not to however, because we want CλasH to catch as many design errors at compile-time as possible. Exposing the unsafe constructors and deconstructors would mean that errors could then only be caught at run-time. So, with those unsafe constructors hidden, users of CλasH will only be able use the vector transformation functions defined in the TFVec vector library, and not be able to define any of their own.

In the end, we decide to use the TFVec library as *the* vector library for CλasH. Even though it has the earlier mentioned shortcomings, it still provides the eventual hardware designer with a plethora of functionality. Perhaps the further exploration of the GADT-based vector can be performed in future work.

3.2.5 TRANSLATION TO VHDL

Having made the choice for the vector library, all that remains is to translate the types and functions to equivalently behaving VHDL constructs. Translating the Haskell type to the VHDL type is the easiest part, as VHDL already has the **array** type. Take the following Haskell type for example:

type *Example* = *Vector D8 Bit*

This type can be translated to the following, corresponding VHDL type:

subtype *tfvec_index* **is** *integer* **range** − 1 **to** *integer'high*;
type *vector_std_logic* **is** **array** (*tfvec_index* **range**<>) **of** *std_logic*;
subtype *vector_std_logic_0_to_7* **is** *vector_std_logic* (0 **to** 7);

The first subtype declaration, the index range, is needed because some vector functions might return empty vectors. To define such arrays in VHDL, we usually give them an index range from 0 *to* -1, to indicate that they are a null slice. That is why the default *natural* range (used for example by **std_logic_vector**) will not suffice. This range index has only to be defined once, as all unconstrained vectors will use this range.

The second type declaration is an unconstrained array declaration, it only defines that **vector_std_logic** is an array, and has **std_logic** as an element type. The CλasH built-in **Bit** type is translated to the VHDL **std_logic** type.

The third, and final declaration, defines the size of the array. In this case it has 8 elements, and thus the index runs from 0 to 7. Seasoned hardware designers might wonder why the index is ascending (0 **to** 7) and not descending (7 **downto** 0), as a descending index is a common practice. We have chosen to use an ascending index because Haskell lists also intuitively also have an ascending index. For example, when when define the list: $[a, b, c, d]$, the index of the left-most element is 0, and the right-most element has index 3, and the list has as such an intuitively ascending index for those of us used to a left-to-right writing system. Nonetheless, if deemed preferable, the order of the indexes could be reversed with relative ease in a future version of the CλasH compiler.

The subtype with the specified range will be used for signal declaration, and the unconstrained vector type is mostly used in vector transformation functions. If there is a another vector type that has the same element type but a different size, we will only have to add another *subtype* declaration.

Translating vector functions

When translating the vector functions to VHDL two things have to be taken into account. The body of the vector functions of the TFVec library actually deconstruct the vector to a list, and applies the corresponding transformation on this list. So, if we chose to generate hardware as we traverse the recursive list functions, we need not only suddenly support Haskell lists, but also calculate the length of the list for each recursive step. Secondly, because of the choices made earlier, there is only a predefined set of vector transformation functions.

For these two reasons, we chose to make fixed translations, templates, for all the functions defined in the TFVec library, instead of a general way to handle all vector functions. The downside of this solution is of course, that most of this work could be deleted from the compiler codebase once CλasH supports general recursion. However, the framework for specifying fixed translations could be left intact for optimization purposes. For the time being though, having fixed translations seems to be a valid choice, as it will take some time and work (as in, at least the timeframe and amount of work of an entire master's assignment) before general recursion will be supported in CλasH.

It turns out that the fixed translations for all the vector functions are quite straightforward. All fixed translations are listed in Appendix A, and only one will be discussed here. We chose one of the higher-order functions, *foldr*, as it is one of the more elaborate translations. Let us start with the function signature as we know it for lists:

$$\begin{aligned}
&\mathit{foldr} :: (b \rightarrow a \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
&\mathit{foldr}\ f\ z\ [\,] \qquad\ = z \\
&\mathit{foldr}\ f\ z\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}$$

What *foldr* does is replace the structural components of the vector data structure with functions and values in a regular way. The empty vector is replaced by a starting value, and all the cons operators ((:) for lists) are replaced by a function. In Figure 3.1 we can see a graphical representation of this transformation applied to a list, where the starting value is called *z*. So, if an empty vector is passed to this function, all it actually does is to return the starting value. So in VHDL we will translate this to a simple signal assignment:

$$a \Leftarrow z;$$

The most interesting part is of course the transformations for vectors that have one element or more. The VHDL template for the *foldr* function on non-empty vectors is shown in Code Snippet 3.14.

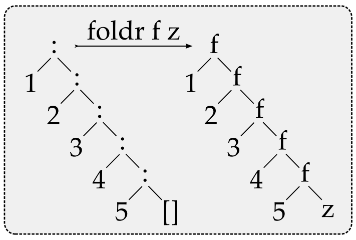

Figure 3.1: Right fold transformation

It is best to make the template a VHDL *block* statement, so that we can define a temporary vector signal that only exists within the local scope of the *block* statement. This temporary vector signal, *tmp*, is used as a 'storage' vector for the results generated by the instantiated components. Also, you can witness that the calls to the *function f* are translated to a component instantiation of the *entity f*. As you can determine by examining the first *generate* statement, we work from the last vector element towards the first element. So for the first iteration of this *generate* statement, the last element of the vector and the starting value are passed to the function *f*, and the result is stored in the last element of the temporary vector, *tmp*. For the other iterations the *generate* statement works its way up the input vector, applying the function *f* in the same fashion as witnessed in Figure 3.1. The result of the last function application is finally assigned to the output value.

Code Snippet 3.14 (*VHDL template for foldr).*

```
foldrVectorBlock : block
  signal tmp : array_of_a_with_length_of_veci;
begin
  foldrVector : for n in (veci'length − 1) downto 0 generate
  begin
    firstcell : if n = (veci'length − 1) generate
    begin
      comp_ins : entity f
        port map (output ⇒ tmp (n),
                  input1 ⇒ z       ,
                  input2 ⇒ veci (n),
                  clock  ⇒ clock   );
    end generate firstcell;
    othercells : if n /= (veci'length − 1) generate
    begin
      comp_ins : entity f
        port map (output ⇒ tmp (n)     ,
                  input1 ⇒ tmp (n + 1),
                  input2 ⇒ veci (n)    ,
                  clock  ⇒ clock       );
    end generate othercells;
  end generate foldrVector;
  a ⇐ tmp (0);
end block foldrVectorBlock;
```

In the eventual generated VHDL the *veci'length* − 1 statements are replaced with the actual corresponding values. Also, the component instantiations might be replaced by an arbitrary concurrent statement (usually a function call) when certain built-in functions (a function for which a VHDL template exists) are used.

3.3 Integers

In software, and certainly in hardware, integers are usually represented by a fixed number of bits: Meaning that integers usually have a maximum and minimum value that is allowed by its representation. It also means that certain arithmetic operators behave in a special way when the result of an arithmetic expression exceeds those extremes. The point where an operator causes the result to exceed an extreme is called an *overflow*.

The most common behavior performed by an operator when an *overflow* occurs is to cause a so-called *wrap around*, where the value of the result flips to the other extreme and continues the operation from there. A second, less common behavior also found in hardware, is *saturation*, where the resulting value will remain at the extreme of its representation.

For a HDL that wants to have (some) behavioral aspects (instead of being purely structural) it is important to have support for integers. An important part of this is that you should be able to specify the range of values an Integer can represent. This range specification is important as it also determines the ultimate size/structure the hardware working on these integers will have. For the above reasons CλasH has support for integers with a certain range specification. Again we want to specify these range details at the *type*-level, and for the same reasons as we wanted size specification at the *type*-level for fixed-size vectors.

Besides being able to specify these ranges, it is also important to see the implications (such as the behavior that occurs at an overflow) of these representation ranges during simulation. There are two possible options for potentially specifying the range of an integer:

- You can specify them directly: e.g. *Range -128 To 127*
- Define the number of bits by which the Integer is represented: e.g. 8 bits for the range of values between -128 and 127

In CλasH we will support both ways of specification, each for different purposes. As the *overflow* behavior is caused by the number of bits, we have chosen to use the second specification for integers that will be used in arithmetic operations. Section 3.3.1 goes into detail as to how we describe these *sized* integers, how their simulation works and how they, and the operations on them are translated to VHDL.

Sometimes you want to make sure that an integer does not exceed a certain subrange, even if it is within their representable range. An example of such an integer is a *safe*[9] index into an array, when the size of that array is not of a power of 2. For these cases we want the simulator to throw an error when the integer exceeds its specified subrange. Of the two range specification approaches mentioned earlier, the first approach is the only approach that matches this goal. So, in Section 3.3.2 we will go into the detail of these *ranged* integers. We will see how they are implemented and how they are translated to VHDL.

3.3.1 sized integers

The familiar TFP library, whose *type*-level numerals are used in the TFVec library of the last section also defines the types **SizedInt** and **SizedWord**. They represent *signed* and *unsigned* integers of a certain bit size respectively. **SizedNat** or **SizedNatural** might have been a better name for the type that represents unsigned (hence natural) integers. The name **SizedWord** was however chosen because the behavior of the operators defined for **SizedWord** match the behavior of the operators defined for

[9]An index is considered *safe* when it can not exceed the number of elements in an array.

the default Haskell **Word** types: **Word8**, **Word16**, etc. The constructors for **SizedInt** and **SizedWord** are shown below:

```
newtype (NaturalT nT) ⇒ SizedInt nT   = SizedInt Integer
newtype (NaturalT nT) ⇒ SizedWord nT = SizedWord Integer
```

Their constructors are of course nothing special, however, their run-time behavior and their dependency on the *type*-level numerals to specify their size are quite interesting. The *Num* type class[10] is most interesting for these sized integers, as it specifies addition, subtraction and multiplication operators. Not only is the *Num* type class important because of these operators, the *fromInteger* function is also the general way to construct a **SizedInt**. This is because the normal constructors for **SizedInt** and **SizedWord** are hidden, as those are *unsafe* (they would allow us specify integers that do not respect the bounds set by the size variable). Constructing them with the *fromInteger* function allows us to make sure they are constructed correctly, respecting the bounds set by the size variable. The *fromInteger* function can be called implicitly, like it happens in the code below, meaning that our descriptions can remain concise:

```
let x = (3 :: SizedInt D4)
```

So what the Haskell compiler actually does 'under the hood' is translate the above code to the code shown on the next page:

```
let x = (fromInteger 3) :: SizedInt D4
```

In Code Snippet 3.15 we can see some of the most important functions of the *Num* instance for the **SizedInt** type.

Code Snippet 3.15 (*Num instance for SizedInt).*

```
instance NaturalT nT ⇒ Num (SizedInt nT) where
  (SizedInt a) + (SizedInt b) = fromInteger (a + b)
  negate (SizedInt n)          = fromInteger ((n `xor` mask (⊥ :: nT)) + 1)
  fromInteger n
    | n > 0                    = SizedInt (n .&. mask (⊥ :: nT))
    | n < 0                    = negate (fromInteger (negate n))
    | otherwise                = SizedInt 0
```

On the first line we specify that **SizedInt nT** is an instance of the **Num** class, the class that defines the basic numeric operations such as *addition* (+), and *negate*. Even though *NaturalT nT* was already part of the context of the **SizedInt** type, we need to specify this context again, as the *mask* function needs to know that it is working with the required *type*-level integers.

The most important function in the **Num** instance is the *fromInteger* function, as that is the only constructor of **SizedInt** available to a developer. Let us examine the case for positive integers, the case guarded by $n > 0$:

- The (.&.) operator is the bitwise-*and* operation.
- The *mask* function takes the *type*-level integer, nT (which represents the size of the **SizedInt**), as its arguments and returns a *term*-level integer. This *term*-level integer is the maximum representable *natural value* for the specified size nT. For example, if *mask* is passed (⊥ :: *D4*), it returns the integer *term* 15.

[10]The type class system is explained in Appendix D.

The effect of bitwise *and*-ing the variable n with the maximum representable *natural value*, is that all the bits in n that exceed the maximum representable *natural value* will be set to zero, thereby respecting the bounds set by the size of **SizedInt**.

The observant reader will note that **SizedInt** is supposed to be a *signed* integer: When we exceed half of the maximum representable *natural* range, an integer should flip to a negative number. And the above *fromInteger* function certainly does not make that happen. This is certainly true, however, we are not really interested in the sign of the integer until we actually print the value. That is why the *show* function (which turns a **SizedInt** into a printable string) makes a call to the *toInteger* function, which is defined as follows:

```
toInteger s@(SizedInt x) =
  if isNegative s
    then let SizedInt x' = negate s in negate x'
    else x
```

The *isNegative* function checks if the *sign* bit of the sized integer is set, and then calls the *negate* function of **SizedInt** first, followed by a call to *negate* for the integer x'. This will certainly result in the correct integer, as the *negate* function for **SizedInt** basically does a 2's complement conversion of the bit pattern: The *negate* function first *xor*s with the mask, which is basically the inversion part of the 2's complement conversion, and then adds one bit. Proof 3.1 should convince us: we will add two 4-bit integers, both having the value 7, which should result in a value of -2.

Having covered the most important functionality of **SizedInt**, we will not elaborate on the functionality of the **SizedWord** type, as the implementation of the *Num* instance of **SizedWord** is very similar to **SizedInt**. The *negate* function does of course not do a 2's complement conversion as **SizedWord** is an *unsigned* integer. Because **SizedWord** can not represent negative numbers, the implementation of the *toInteger* function also differs slightly from that of **SizedInt**.

Resizing Integers

When we look at the numeric operators of the *Num* class, we soon realize that the results will *never* exceed the bounds of their representation. This is usually what we want, but for some cases, like multiplication, we might want the result to be of a larger representation. For example if we try to do:

```
let
  x = (7 :: SizedInt D4)
in
  (z :: SizedInt D8) = fromInteger $ toInteger $ (x * x)
```

The variable z will not have the value 49, but the value 1, because the original multiplication, $x * x$, had an overflow before it was converted to the new **SizedInt**. Of course, what we had to do was first turn the variable x to an Integer before multiplying it. But then we get such intricate code as:

```
(z :: SizedInt D8) = fromInteger $ (toInteger x) * (toInteger x)
```

Precisely for these situations, a *resize* function function was added. The *resize* function does exactly what the name implies, it correctly resizes the argument to the type of the result. The behavior of the *resize* function for **SizedWord** is to zero-extend, whilst for **SizedInt**, the *resize* function will sign-extend. The resize function works quite intuitively, for example, we would rewrite the above multiplication as such:

```
(z :: SizedInt D8) = (resize x) * (resize x)
```

Proof 3.1 (*toInteger ((7 :: SizedInt D4) + (7 :: SizedInt D4)) ≡ -2).*

$(7 :: \mathit{SizedInt\ D4}) + (7 :: \mathit{SizedInt\ D4})$
$\equiv$ definition of (+)
$\mathit{fromInteger}\ (7 + 7)$
$\equiv$ definition of (+)
$\mathit{fromInteger}\ 14$
$\equiv$ definition of fromInteger
$\mathit{SizedInt}\ (14\ .\&.\ \mathit{mask}\ (\bot :: \mathit{D4}))$
$\equiv$ definition of mask
$\mathit{SizedInt}\ (14\ .\&.\ 15)$
$\equiv$ definition of .&.
$\mathit{SizedInt}\ (14)$

$\mathit{negate}\ (14 :: \mathit{SizedInt\ D4})$
$\equiv$ definition of negate
$\mathit{fromInteger}\ ((14\ `\mathit{xor}`\ \mathit{mask}\ (\bot :: \mathit{D4})) + 1)$
$\equiv$ definition of mask
$\mathit{fromInteger}\ ((14\ `\mathit{xor}`\ 15) + 1)$
$\equiv$ definition of xor
$\mathit{fromInteger}\ ((1) + 1)$
$\equiv$ definition of (+)
$\mathit{fromInteger}\ 2$
$\equiv$ definition of fromInteger
$\mathit{SizedInt}\ (2\ .\&.\ \mathit{mask}\ (\bot :: \mathit{D4}))$
$\equiv$ definition of mask
$\mathit{SizedInt}\ (2\ .\&.\ 15)$
$\equiv$ definition of .&.
$\mathit{SizedInt}\ (2)$

$\mathit{negate}\ 2$
$\equiv$ definition of negate
-2

Q.E.D.

Translating to VHDL

The VHDL translation for the **SizedInt** and **SizedWord** type, and the corresponding numeric operations: $(+)$, $(-)$, *negate* and $(*)$, are very simple. That is because VHDL already has the corresponding **signed** and **unsigned** type. And the corresponding numeric operations are also defined for these types, except for the *negate* function on **unsigned**. For this reason, the *negate* function of **SizedWord** throws an error when called. A preferred solution would of course be to not define the *negate* function at all for **SizedWord**. Sadly so, it is not possible to hide class functions in Haskell, meaning that we can't both let **SizedWord** be an instance of *Num* and hide the *negate* function. So throwing an error is the small sacrifice we have to make for getting access to the other numeric operators.

As far as the *resize* function goes, we call the respective VHDL *resize* function for **unsigned** and **signed**, which behave the same as their Haskell counterparts: they zero and sign extend respectively.

3.3.2 RANGED INTEGERS

Until now we have mostly talked about sized integers. These sized integers can of course also be used to represent ranged integer, having of course only ranges that are a power of two. So the question is: Why would we want integers with an arbitrary range specification? Especially when we consider that the actual hardware, which is binary, can only physically manifest integers with ranges that are a power of two. we will see a convincing use case as to why CλasH should support type-level ranged integers.

Use Case: 10 x 8-bit Memory

Suppose we want a 10 x 8-bit Random-Access Memory (RAM), in which we want to store 8-bit integers. The read and write address will have to be at least 4 bits wide, so already we can see that we can give no guarantees that the address value will stay within the limits of the memory. Also, the element access function, (!), which we would use to specify the read functionality of the RAM, had[11] the following type:

CODE SNIPPET 3.16 (*The original index operator (!)).*

```
(!) :: (PositiveT s
       , NaturalT i
       , (s > i)~True) ⇒ TFVec s a → i → a
```

This poses a big problem, the function expects a type-level integer, because it needs to know that the indexing argument does not exceed the size of the vector. A first attempt to specify the read logic of the RAM is seen on the next page:

$$data\_out = reifyNaturalD\ (toInteger\ read\_address)\ (\lambda i \rightarrow ram\ !\ i)$$

A piece of code we would think should be simple, the read functionality of a RAM, turns out to be much more intricate than hoped for. What is even worse, the code will not even compile. Let us start with the unfamiliar *reifyNaturalD* function. This function is implemented in a Continuation-Passing Style (CPS) fashion, meaning that it passes the result of its calculation to the second argument. What *reifyNaturalD* does is convert an integer *term* to the equivalent *type*-level integer. So what happens in the above piece of code is that the *reifyNaturalD* function translates the *read_address* variable to the type-level equivalent, which is then passed to the (!) function to access that position in the vector.

You can see why we would assume the code works, it gives the (!) function the type-level integer it wanted. However, the only guarantee *reifyNaturalD* can give at compile-time is that it gives back a type-level natural number[12], but it can (of course) not say which instance exactly. As such, the type-checker will complain that it can not verify that the index is indeed smaller than the size of the vector.

The problem is that the restriction imposed by the indexing operator, (!), is too strict. It demands to know the exact instance of the type-level integer. However, the only static guarantee that we should care about is: "Is the static upper bound of this integer larger than the highest index position of the vector?" If not, than we do not care which exact instance it is. Now we could of course, using only the types we currently have at our disposal, change the type signature of the indexing operator *(!)* function into something like the code shown on the next page.

[11] This use case actually resulted in the change of the (!) and *replace* functions.
[12] Or gives an error when the integer is smaller than zero

Code Snippet 3.17 (*Index operator (!), using SizedWord as index parameter).*

```
(!) :: (Positive s
       , NaturalT i
       , (s ≡ (Pow2 i))
       ) ⇒ TFVec s a → SizedWord i → a → TFVec s a
```

In which we use a **SizedWord** type as the index parameter, and **Pow2 i** operator is the type-level equivalent of 2^i. But then, we would never be able to use this indexing operator on a 10-bit vector: We need 4 bits to represent the number 9, but $2^4 = 16$ is larger than 10.

The introduction of RangedWord

The solution to this problem are of course arbitrary ranged integers. For example, a simple solution for the 10 x 8-bit RAM use case are indexes that are natural numbers with only an upper bound, specified as such:

```
newtype (NaturalT upper) ⇒ RangedWord upper = RangedWord Integer
```

We would then change the indexing operator *(!)* to have the following type signature:

Code Snippet 3.18 (*The current implementation of the Index (!) operator).*

```
(!) :: (PositiveT s
       , NaturalT u
       , (s > u)~True) ⇒ TFVec s a → RangedWord u → a
```

The implementation of the *Num* class for the **RangedWord** type is not as interesting as that of **SizedInt** and **SizedWord** type. Again, the unsafe constructor exposed by the *newtype* declaration is hidden, and the *fromInteger* function is used to safely construct the **RangedWord**. What the *fromInteger* function does for the **RangedWord** type is throw an error when the integer passed to it as its argument exceeds the implicit minimum range of 0 or the maximum bound specified by the type.

What is interesting are the conversion functions between **RangedWord** and **SizedWord**. Using these conversion functions, a developer can still exploit the *overflow* behaviour of **SizedWord**, but guarantee the range safety offered by **RangedWord**. Of course, there are some limitations: The largest being that when converting from **SizedWord** to **RangedWord**, then the upper bound of a resulting **RangedWord** has to be of a power of two, minus one, as that is the only static guarantee the conversion can give. The function that converts a **RangedWord** to a **SizedWord** is seen in Code Snippet 3.19, and the function that converts a **SizedWord** to a **RangedWord** is seen in Code Snippet 3.20.

Code Snippet 3.19 (*Convert RangedWord to SizedWord).*

```
fromRangedWord ::
   (NaturalT nT
   , NaturalT nT'
   , ((Pow2 nT') > nT)~True
   , Integral (RangedWord nT)
   ) ⇒ RangedWord nT → SizedWord nT'
fromRangedWord rangedWord = SizedWord (toInteger rangedWord)
```

Most of the type context should be familiar; the line, $((Pow2\ nT') > nT) \sim True$, asks the type-checker that the upper bound (nT) of the **RangedWord** type fits in the maximum representable value

(*Pow2 nT'*) of the **SizedWord** type. The context, *Integral* (*RangedWord nT*), has to be specified so that type-checker knows that the **RangedWord** type has a *toInteger* function.

Code Snippet 3.20 (*Convert SizedWord to RangedWord).*

```
fromSizedWord ::
  (NaturalT nT
  , Integral (SizedWord nT)
  ) ⇒ SizedWord nT → RangedWord ((Pow2 nT) − D1)
fromSizedWord sizedWord = RangedWord (toInteger sizedWord)
```

The functionality, and the type-signature, of the above conversion should be straightforward for the reader. Also here, the context, *Integral* (*SizedWord nT*), has to be specified to let the type-checker know that the **SizedWord** type has a *toInteger* function.

Translation to VHDL

VHDL has support for natural numbers (the **natural** type) that have a certain range. However, we do not translate **RangedWord** to this VHDL type **natural**. The underlying reason for this is explained in Chapter 5. So what we do, is translate the **RangedWord** type and all its numeric operators in the same way as the **SizedWord** type: We translate to the VHDL **unsigned** type. This is a sound translation, meaning that we do not sacrifice any type safety, as we can already check for any range errors when we simulate a design in Haskell.

Chapter 4

Case Study: Reduction circuit

The development and implementation of the CλasH compiler was an incremental process, in which new functionality was added piece by piece. Usually, a small *toy* code example was created to determine if a newly added piece of functionality was behaving correctly. However, we never tested the compiler with a relatively large hardware design.

We therefore did this case study, to examine if CλasH can do more than *toy* examples, and see if it was ready for *real* hardware designs. The piece of hardware designed and implemented is a reduction circuit. It was originally designed in the context of Sparse Matrix Vector multiplication (SM$\times$V), but the circuit can be used for other purposes as well. A reduction circuit reduces all the values in a row to a single value through a binary operation, and for this particular exercise we use summation.

When we deal with integer values, this circuit can be quite straightforward. We use a single integer adder with an accumulator register, and we will just sum the values of each row by streaming the rows through this circuit one by one. When the values of the rows to be reduced are floating point values there are several hardware specific problems we need to solve. Most floating point adders are pipelined circuits (to achieve a reasonable clock speed), meaning that the result of the operation is delayed by multiple cycles. We can *not* take the same approach as we did with integer values: If we would try to stream the values through such an accumulation circuit we risk adding the values of multiple rows together, as values of a new row would enter the floating point adder before the values of the previous row exited the pipeline.

We can not stall the reduction of a new row until the last value of the previous row is flushed from the pipeline as we risk needing an infinitely large input buffer. The speed by which we can reduce all the rows in a matrix would also significantly degrade. For the above reasons special algorithms are used to keep the floating point adder pipeline as filled up as possible, while still keeping track to which row the values belong.

The specific reduction circuit design implemented in this chapter is from the work of Gerards [13]. Figure 4.1 shows the basic components of the design: The input buffer is a First In, First Out (FIFO) buffer which is special in the sense that it can release between 0 and 2 values.

When a partially reduced value is at the end of the pipeline, while other partial results are still in the pipeline, then that value needs to be stored in special partial results buffer so that they can be reduced further later on. We have to store these partial results until the complete row has been reduced; when the row has been completely reduced it can be flushed towards the output.

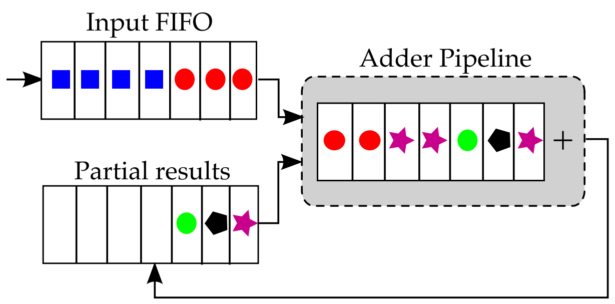

Figure 4.1: Reduction Circuit

The part of the design that is not shown in Figure 4.1 is the control logic that is needed to have the different parts work together. This control logic needs to:

- Keep track to which rows the values in the pipeline belong to.
- Determine from which source values should enter the adder pipeline next: from the input buffer, the partial results buffer, and/or the top of the adder pipeline.
- Determine if a row is completely reduced and needs to be purged from the partial results buffer.

The problem is complicated even more by the fact that the rows can have different lengths, and that some rows have more elements than that there are pipeline stages in the floating point adder. A reader in search of more detail about about both the problem and the architecture used to solve this problem is referred to the master's thesis of Gerards [13].

The focus of the reduction circuit design lies mainly on the control logic, not the floating point adder itself. We therefore replace the floating point adder in the design by a much simpler datapath: an integer adder connected to a shift register. This is a valid replacement, as far as the control logic is concerned, as the external timing behavior of shift register and the pipeline of the floating point adder are the same: The output of the computation is delayed by several clock periods.

The ultimate design of the reduction circuit does not demonstrate the merits of functional HDLs very clearly. This is largely due to the fact that the design was already completely described in the thesis of Gerards [13]. The merit of HDLs like CλasH is that the designs described in these languages allow for a high degree of abstraction and parameterization, due to such features as polymorphism and higher-order functions. These features aid a designer in dealing with the complexities when a circuit design is far from finished and there are still many uncertainties. As the architecture of the reduction circuit was already finalized, the implementation in CλasH resembles the implementation a seasoned designer would write using VHDL or Verilog; be it that CλasH might be less verbose than VHDL. That being said, a novice designer might have an easier time implementing the circuit in CλasH due to this language being less verbose than VHDL or Verilog, and that a design that can be compiled is always synthesizable.

Speaking of verbosity and conciseness, the fixed-size vectors found as they are currently implemented in CλasH put a strain on both concepts when dealing with certain parts of the architecture, especially the input buffer of the reduction circuit. That is why we will discuss the design of the input buffer in further detail. The reader should note that knowledge of the **State** *newtype* is assumed; readers unknown with the use of this *newtype* are referred to the thesis of Kooijman [26].

4.1 The input buffer

The input buffer of the reduction circuit is a special type of FIFO buffer, that can shift either 2, 1 or 0 values out of the buffer. This specialized FIFO was needed so that the floating point adder could read 2 values out of the input buffer at once, without delay. Using dynamically sized lists (which are unsupported in CλasH), you might describe such a buffer along the lines of the code below, where *NotValid* indicates that a memory *Cell* is invalid:

Code Snippet 4.1 (*Purely functional design of a FIFO buffer).*

```
fifoBuffer (State mem) (input, shift) = (State mem', out1, out2)
  where
    out1 | length mem ≡ 0 = NotValid
         | otherwise       = head mem
    out2 | length mem ≡ 1 = NotValid
         | otherwise       = head (tail mem)
    mem' = drop shift mem ++ [input]
```

The above can currently not easily be translated to the fixed-size vectors like we have them in CλasH: vectors that have a *static size* indicated at the *type*-level. In the above code the *drop* function trims a *variable* amount from the head of the list *mem*. The input is concatenated to his trimmed list. This type of functionality can not be specified with the fixed-size vectors in CλasH, as the result vector *mem'* needs to be assigned a vector with a *static* length. Maybe future versions of CλasH will support *bounded* vectors (vectors with a static upper bound) or even *dynamically* sized lists. These types of functionality is what we would call *behavioral* aspects (as a *structure* will have to be inferred) of the language, and would most likely complicate the design of the CλasH compiler significantly. So, for the time being, *bounded* vectors and *dynamic* lists are left as possibilities for future work.

At the moment however, we as designers need to find our own solutions to mimic the behavior of dynamically sized lists. What seems to a cheap solution to our problem is the use of a write pointer. This write pointer points to the first open spot in the FIFO buffer. The result design, which is in the old design of the input buffer, is shown in Code Snippet 4.2.

Code Snippet 4.2 (*CλasH design of a Shift buffer (FIFO)).*

```
fifoBuffer (State (Fifo {..})) (inp, shift) = (State (Fifo { mem   = mem'
                                                            , wrptr = wrptr'
                                                            })
                                                , out1, out2
                                                )
  where
    -- Increase or decrease write pointer according to value of 'shift'
    wrptr' = wrptr − shift + 1
    -- Write input value to the free spot
    mem''          = replace mem wrptr (Valid inp)
    -- Flush values at the head of fifo according to value of 'shift'
    mem' | shift ≡ 0 = mem''
         | shift ≡ 1 = (tail mem'')<+NotValid
         | otherwise = ((tail (tail mem''))<+NotValid)<+NotValid
    -- Output the last two values of the buffer
    out1           = head mem
    out2           = head (tail mem)
```

Functions like *head* and *tail* should be familiar. So let us examine the unfamiliar ones. The *replace* function does exactly what it name suggests, it replaces a value in a vector with a new value; in the case of the input buffer it places the new input value at the spot indicated by the write pointer. The *snoc* (<+) operator, attaches an element to an existing vector on the *right* side; in this sense it is the opposite of the more familiar *cons* (+>) operator which attaches an element to the *left* of an existing vector. The first line of the **where**-clause, $wrptr' = wrptr - shift + 1$, shifts the write pointer either one position to the left, one position to the right, or does not change its value at all, depending how many values are *shifted* out of the input buffer. We can also see that invalid elements (*NotValid*) are shifted into the buffer when values are shifted out of the input buffer; this way, the size of the storage vector stays the same.

After some tests we noted that the shift buffer design of Code Snippet 4.2 is faulty: The buffer can never be completely filled without raising an error. When the buffer is completely filled, the write pointer *wrptr* will have a value that is equal to the maximum number of elements in the buffer. And if the *shift* variable is 0 at such a time, the computation $wrptr - shift + 1$ will give an error. That is because the *replace* function demands that the write pointer is of type **RangedWord** with a static upper bound that does not exceed the highest index of the vector. So when the *shift* variable has the value 0, the above computation will exceed the upper bound, and an exception will be raised.

A possible solution is to turn the write pointer to a **SizedWord**, and converting it to a **RangedWord** again when we use it as the indexing parameter for the *replace* function. However, this will limit the design of the FIFO buffer to have a length that is a power of two. We had a different solution in the old design: As we know that the minimum required length of the input buffer for this reduction circuit design is $\alpha + 1$, where α is the pipeline depth, we will just make the size of the input buffer $\alpha + 2$. This way the write pointer will never go out of bounds. Even though one memory location will always go unused, it is a better solution than increasing the size of the design to be a power of two.

Code Snippet 4.3 (*CλasH design of a Circular buffer (FIFO)).*

```
fifoBuffer (State (Fifo {..})) (inp, shift) = (State (Fifo  { mem   = mem', rdptr = rdptr'
                                                             , wrptr = wrptr', count = count'
                                                             })
                                              , out1, out2
                                              )
  where
    -- Update the read pointers and element counter according to value of 'shift'
    (rdptr', count') | shift ≡ 0  =                                (rdptr    , count + 1)
                     | shift ≡ 1  = if rdptr ≡ max          then (0        , count    ) else
                                                                 (rdptr + 1, count    )
                     | otherwise  = if rdptr ≡ (max − 1) then (0        , count − 1) else
                                    if rdptr ≡ max          then (1        , count − 1) else
                                                                 (rdptr + 2, count − 1)
    rdptr2                        = if rdptr ≡ max then 0 else rdptr + 1
    -- Write input to first free spot, update free spot pointer
    mem'                          = replace mem wrptr inp
    wrptr'                        = if wrptr ≡ max then 0 else wrptr + 1
    -- Validity of the output is based on the number of elements
    out1             | count ≡ 0 = NotValid
                     | otherwise  = (Valid (mem ! rdptr))
    out2             | count ⩽ 1 = NotValid
                     | otherwise  = (Valid (mem ! rdptr2))
```

The current solution uses a circular buffer (Code Snippet 4.3), employing two read and one write pointer. This way we can add explicit overflow to the pointers, meaning that the type of the pointers can still be of **RangedWord**, without running into the unwanted overflow in the old design. In Code Snippet 4.3, the *max* variable is the maximum buffer index, and is defined outside of the circuit description of the FIFO buffer. We can also see that the design uses a counter, the variable *count*, that counts the number of valid elements in the buffer. This *count* variable is used to indicate if the values at the output of the FIFO buffer are valid values.

Using a circular buffer we have a design which probably uses fewer resources than the previous buffer design, as the old design, having a large shift register, probably used more multiplexers. However, the logic generated for the circular buffer might have a lower clock speed than the logic generated for the shift buffer.

4.2 Results

As the current incarnation of the CλasH compiler is certainly not optimized for either speed or resource usage, there is no point in discussing these aspects of the reduction circuit design in great lengths. We will however mention them briefly at the end of this section. What is worth elaborating is the design process of the reduction circuit, and how it helped finding many bugs in the CλasH compiler.

So first of all, the original architecture of the reduction circuit described by [13] was already there before work on CλasH started. Not only that, the functional description of the circuit was also already designed before real work on CλasH was started. So not only was it a nice use case to see what of kind descriptions CλasH *can* currently handle, it has also been sort of a guiding example of what the compiler *should* handle. During the implementation of CλasH we had to update the design of the reduction circuit to deal with certain language specifics, such as the *State* newtype and the hardware specific types such as the **RangedWord** type. The biggest update being of course moving from dynamically sized lists to fixed size vectors. Actually, the restrictions of physical hardware were recognized before the fixed size vectors were implemented in CλasH: So the actual description of the reduction circuit was updated to reflect these restrictions even before the TFVec library was implemented.

The reduction circuit proved to be a good use case as far as compiler stability goes, allowing us to catch many bugs in the *State* handling parts of CλasH that did not show up in our *toy* examples. The final design was tested in the Haskell simulation environment using randomly generated test input that constructed rows of random lengths. The output of the reduction circuit was machine-checked against the output of a simple Haskell equation that also reduced the rows: The output of the circuit was equivalent to that of the Haskell equation indicating that our design behaved correctly under simulation.

After the confirmation that the design behaved correctly in the Haskell simulation, the CλasH compiler translated the design to VHDL and also automatically generated a testbench that uses the same input values as those in the Haskell simulation. The automated testbench generation was another fruit that grew out of the reduction circuit use case: Determining the correctness of the generated VHDL code by manual code inspection was infeasible for such a large design. So to acquire confidence in the correctness of the generated VHDL, we needed to perform a VHDL simulation run of the generated design to determine that the generated design, given the same input, had the same external behavior as the CλasH design. The reader can find more information about the automated testbench generation in Appendix B.

The VHDL that was generated by the CλasH compiled without errors or warnings, and machine-checking the output of the VHDL testbench against the output of the Haskell simulation showed that the external behavior of the CλasH design and the generated VHDL were equivalent (at least for the specified test input). Also, the VHDL synthesis tool compiled the design without

	CλasH	VHDL	FP Adder
CLB Slices	1816	3556	n/a
MHz	106	200	253
Block RAMs	2	9	0
DSP48s	0	3	3
Function Generators (LUTs)	3631	2927	1220
Dffs or Latches	2192	3437	1139

Table 4.1: Design characteristics Reduction circuit

errors or significant warnings, indicating that the design of the reduction circuit can be turned into actual hardware.

4.2.1 SYNTHESIS RESULTS

For the sake of completeness, this subsection goes into the details of the synthesis results of the reduction circuit. It has to be noted that the CλasH compiler is in no way optimized for speed or resource usage. The design was synthesized for a Xilinx Virtex-4 4VLX160FF1513-10, the same FPGA that was used by Gerards [13]. The pipeline depth of the adder is 12, and the circular buffer was used as the input buffer for the design. In Table 4.1 we can see the resource usage of the CλasH design and the optimized VHDL design of Gerards [13]. The numbers of the optimized VHDL design can not really be used for comparison with the CλasH design for two reasons: The VHDL design uses a real pipelined floating point adder, whose resource usage is much higher than the integer adder used in the CλasH design. Table 4.1 therefore includes the resources used by a stand-alone pipelined floating point adder, to give some indication as to how much the floating point adder affects the resource usage of the reduction circuit by Gerards [13].

Secondly, because the CλasH design uses an integer adder, the delays between input buffer and adder and between adder and output buffer are lower than if a pipelined floating point adder was used. This means that the maximum clock speed of the CλasH design would probably be lower if a floating point adder was used.

Still, Table 4.1 shows us that the number of FPGA resources used by the CλasH design are in the same order as the number of resources used by the hand-coded VHDL[1], as such, we are confident that this first-generation CλasH compiler is indeed well behaved.

[1]Even though the the number of used CLB slices for the FP Adder is unknown, and as such can not be deducted from the resources used by the VHDL design, the other resource usages do suggest that the CλasH design and the VHDL have a resource usage that is in the same order.

Chapter 5

Discussion

The use case in the previous chapter certainly shows that CλasH can be used for more than just *toy* examples. It might not have the same performance as hand-coded VHDL, but the results are certainly reasonable. However, that does not mean that we, as the designers of CλasH, are entirely satisfied with the current implementation. This chapter discusses some of the design decisions made, and what the implications are for the current state of CλasH.

5.1 Combining higher-order functions with fixed translations

CλasH supports higher-order functions in many cases, but not in the *general* case. The cases where CλasH certainly does *not* support general high-order functions, are those where fixed-size vectors are involved. The reason behind this, is that templates (fixed translations) are used when we translate vector functions. As an example we will examine the *map* function, which has the following signature:

$$map :: (a \rightarrow b) \rightarrow Vector\ s\ a \rightarrow Vector\ s\ b$$

The type variable b can be of any type, which is what we want, so it can even be $(a \rightarrow a)$, which it will have in the example below:

$$\begin{aligned}&\textbf{let}\ (v :: Vector\ D4\ Bit) = copy\ Low\\&\textbf{in}\ \ andv = map\ (.\&.)\ v\end{aligned}$$

The (.&.) function is the bitwise *and* operation in Haskell, meaning that it takes two arguments. So, what this means is that is that *andv* has the type **Vector D4 (Bit → Bit)**, as the *and* operations still expects an argument. This code example is not translatable in CλasH, as we always translate the *map* function through a VHDL template. However, all VHDL function templates expect that both the input and output signals are of a primitive types, to which **(a → a)** does *not* belong.

What seems to be a general solution for this problem is for the normalization transformations, which are described by Kooijman [26], to have some kind of knowledge about the built in functions, such as *map*. This way, if the normalization process notices that a vector operation gives a result that is a function type, it can unwrap the vector to separate functions. It also has to do some bookkeeping in case an other function tries to use one of these unwrapped functions. All this is far from trivial, and the details should be investigated in future work.

5.2 Separation of Logic and State

As we have seen in the introduction and the use case, hardware descriptions in CλasH are basically descriptions of Mealy machines, where state and logic are separated into two different entities. And even though we use functions that operate on the state variables, when we take the Mealy machine perspective, we are actually performing those operations on the output signals of the memory elements. When we translate our description to VHDL, we keep this separation very much intact. All the operation are translated to parallel signal assignments, and the state update is a single sequential signal assignment along the lines of Code Snippet 5.1.

Code Snippet 5.1 (*VHDL State Update).*

```
state : process (clock)
begin
  if rising_edge (clock) then
    statesignal <= newstate;
  end if ;
end process state;
```

This approach works well for most functions, except for the *replace* function that we find in the vector library. The signature of this *replace* function is:

```
replace ::
  (PositiveT s
  , NaturalT u
  , (s > u)~True) => TFVec s a -> RangedWord u -> a -> TFVec s a
```

So, this function takes an entire vector as an input, and also has an entire vector as its output. We can think of many ways to translate this function to VHDL, but in CλasH we have chosen to do it like this:

```
function replace (veci : array_of_a;
                  ix   : unsigned   ;
                  el   : a          )
  return array_of_a is
  variable res : array_of_a (0 to veci'length - 1);
begin
  res := veci;
  res (to_integer (ix)) := a;
  return res;
end;
```

Why we use unsigned instead of natural as the indexing parameter is something that will be discussed in Section 5.3. Now, this function does exactly what you want it to do for combinatorial logic: from a vector of signals it replaces the signal at position *ix* with the signal *a*. However, when you want to use this function as the write logic for a memory element such a RAM, then almost any VHDL synthesis tool will have troubles recognizing the above as the write logic for a RAM. And indeed, when we apply the *replace* function to a *state* variable, most synthesis tools will lie down a whole set of multiplexers connected to a set of memory elements, e.g. flip-flops. Instead of the *blockRAM* you might be hoping for.

If we were to write a special instance of the *replace* function for stateful variables, we would probably design something along the lines of Code Snippet 5.2.

Code Snippet 5.2 (*Replace for state variables)*.

```
replace : process (clock)
begin
  if rising_edge (clock) then
    veci (to_integer (ix)) <= a;
  end if ;
end process state;
```

Which, in most cases, will be turned into some type of RAM by the VHDL synthesis tools; exactly what we want. Sadly so, it is not possible to generate a function like the above in the current architecture of the CλasH compiler. All operations are translated to concurrent signal assignments, and as such, all operations are translated in the same way, irrespective whether they work on *stateful* variables or on *normal* variables.

5.2.1 introduction of blockrams

The translation of the *replace* function for state variables does not lead to any problems with the functionality of the generated VHDL; its behaviour is equivalent to that of the Haskell function. However, when we look at efficient use of hardware, especially on FPGAs, then the hardware inferred by the synthesis tools for the VHDL in Code Snippet 5.2 is certainly more desirable. The reason being that, if combined with the indexing function, *(!)*, it can be recognized by VHDL synthesis tools as a piece of RAM.

So, to give the designer at least the option to use any of the potential RAM resources on an FPGA, we have to make an extra primitive that has a direct translation to VHDL. And of course we need to define a Haskell function as well, so that we may simulate the RAM primitive. We can see the Haskell function for this *blockRAM* primitive in Code Snippet 5.3. It is parametric in both size and element type.

Code Snippet 5.3 (*BlockRAM - Haskell Specification)*.

```
type RAM s a       = Vector s a
type MemState s a = State (RAM s a)
blockRAM ::
  (PositiveT s
  , NaturalT (s − D1)
  , (s > (s − D1))~True
  ) ⇒ (MemState s a) → a → RangedWord (s − D1) → RangedWord (s − D1) → Bool →
      (MemState s a, a)
blockRAM (State mem) data_in rdaddr wraddr wrenable = (State mem', data_out)
  where
    data_out = mem ! rdaddr
    mem'     = if wrenable then
                 replace mem wraddr data_in
               else
                 mem
```

Again, we have to specify some extra context, because, as explained earlier in Chapter 3, arithmetic relations can not be deduced in the current type system. We have to specify *NaturalT* $(s{-}D1)$ because the index parameter needs to have a natural upper bound, and it cannot deduce that subtracting *1* from the *positive* number *s* results in *at least* a *natural* number. Also it can not deduce that *s - 1* is smaller than *s*, thus we need to supply the context $(s > (s - D1))\sim True$. The *blockRAM* function itself is quite straightforward: the output, *data_out*, is the value of the RAM at the read address (*rdaddr*). The RAM is updated at the write address (*wraddr*), with the input value (*data_in*), if the write enable signal (*wrenable*) is *true*.

Many VHDL synthesis tools describe VHDL templates for RAM memories, that is, code snippets of VHDL that are recognized by the synthesis tools as descriptions of a RAM. In CλasH we translate to a very common template; the relation with the Haskell code of Code Snippet 5.3 should be obvious. The VHDL template is seen in Code Snippet 5.4.

Code Snippet 5.4 (*BlockRAM - VHDL Template).*

```
blockRAM : block
  signal ram : array_of_a_with_length_s;
begin
  data_out ⇐ ram (to_integer (rdaddr));
  updateRAM : process (clock)
  begin
    if rising_edge (clock) and wrenable then
      ram (to_integer (wraddr)) ⇐ data_in;
    end if ;
  end process updateRAM;
end block blockRAM;
```

Both the Haskell simulation code and the VHDL template are really quite intuitive, and we hope that future versions of CλasH will no longer need a primitive and could just turn the Haskell code (Code Snippet 5.3) into VHDL code (Code Snippet 5.4) that can be synthesized to a RAM. Also note that, if required, the addition of a dual-port blockRAM primitive to a future version of the CλasH compiler is trivial.

5.3 Haskell is Lazy, Hardware is Strict

As we saw in the previous section, when we discussed the VHDL template for the *replace* function, the CλasH **RangedWord** type is translated to a VHDL **unsigned** type. Both seasoned and beginning VHDL programmers might wonder why we do not use a **natural**, as it allows us to specify a representable range, which of course corresponds more closely with the **RangedWord** type. The reason is that the Haskell simulation results will not correspond with the VHDL simulation results if we use the **natural** type. It even goes so far, that the VHDL simulation will give us a fatal error, whilst the Haskell simulation runs smoothly. The cause of this discrepancy between simulation results are the different evaluation strategies applied by Haskell and VHDL: Haskell is lazy, and VHDL (Hardware) is strict. The best way to show this difference is with an example. Code Snippet 5.5 shows a write pointer calculator for a FIFO buffer with five spaces. The write pointer points to the first free space in the buffer, when it is zero, the buffer is full.

Code Snippet 5.5 (*Write pointer circuit).*

```
writePointer ::
  State (RangedWord D4) →
  SizedWord D2 →
  (State (RangedWord D4), RangedWord D4)
writePointer (State wrptr) enable = (State wrptr′, wrptr)
  where
    wrptr′ | enable ≡ 0 = wrptr − 1
           | enable ≡ 1 = wrptr
           | otherwise  = wrptr + 1
```

The circuit works as follows: when the *enable* signal is 0, an element is added to the FIFO, so the write pointer is decreased. When the *enable* signal is 1, nothing is done to the FIFO, so the write pointer stays the same. And, when the *enable* signal is 2, an element is released from the FIFO, so the write pointer is increased. Now, the context for this example is that the FIFO is full, which means the write pointer is set to zero. Also, no value is about to enter the FIFO, so the *enable* signal is set to one. Now, because Haskell is lazy, only the case $enable \equiv 1 = wrptr$ is evaluated. The write pointer stays within the boundaries of zero and four, so no exceptions are thrown. Hardware however, is completely strict, and so is VHDL. So, when we translate the code from Code Snippet 5.5 to VHDL, each case is turned into a separate signal assignment, which are then fed into a multiplexer. This multiplexer will, according to the value of *enable*, route the correct assignment to the $wrptr'$ signal.

Where the signal assignments in the Haskell simulation did not go out of their bounds in simulation, some of the signal assignments in VHDL will certainly go out of their bounds. And, in the case that they are specified to be of type **natural**, will throw a fatal exception when that happens. So in the earlier explained case, where *wrptr* has the value: 0 and *enable* has the value: 1, the $wrptr - 1$ expression will be evaluated in VHDL. The result of this assignment is: -1, which is below the lower bound of a **natural**. Even though the invalid output of this assignment will never be routed to the $wrptr'$ signal, the VHDL simulator will throw a fatal error.

To avoid this discrepancy between the Haskell and VHDL simulation, the most sound solution is to use **unsigned** for the VHDL representation of **RangedWord**, because the external behavior of the design can be correctly simulated in Haskell. So we know, in the above case, that $wrptr'$ will never get an out of bounds value in real hardware, because we didn't see it in the simulation either. The only downside of the current solution is of course that we lose the safety of the bounds in the VHDL description. But then again, this is only a problem if we want to incorporate the generated VHDL in a larger, hand-coded, design.

Chapter 6

Conclusions

This thesis shows that the expressivity of functional languages, Haskell specifically, can be used for hardware descriptions. The CλasH compiler can now translate a certain subset of Haskell, with much room to grow. At some point we will have to draw a line and say which parts of Haskell we can translate to hardware, and which parts we can not. For the time being though, the focus should be more on being able to translate parts of Haskell of which we know that they can be used for hardware descriptions.

This thesis explored the use of Haskell's type system for hardware description. And even though it did not have all the type-level programming facilities for general recursion over our fixed-size vectors, it is certainly powerful enough to meet most of our requirements.

The major contribution of this thesis is the TFVec fixed-size vector library that acts as the back-end of the current CλasH **Vector** type. Besides being used for CλasH hardware descriptions, the library can of course be used in other projects as well. All the functions defined in the TFVec library have a corresponding translation to VHDL, which are described in Appendix A. This thesis has also added functionality to the existing TFP library, adding the **RangedWord** type, and adding functionality to the existing **SizedInt** type and **SizedWord** type.

The use case of Chapter 4, the reduction circuit, shows that CλasH can be used for more than just trivial/toy designs. The simulation results from the generated testbench (automated testbench generation is discussed in Appendix B) are equivalent to the simulation results in Haskell, indicating that the generated VHDL behaves as expected. Not only that, VHDL synthesis tools compile the designs without errors or warnings, indicating the design will most likely function correctly when placed on an FPGA.

A minor contribution of this thesis is to make the CλasH compiler more than just an obscure research project. Some effort has gone into making the compiler easy to install and easy to use. For example: there is now support for the GHC *Annotation* system, a system that is currently only implemented in the development version of GHC, but will be available in the stable versions starting at version 6.12.1. With support added for this annotation system, a designer can annotate functions in his design, indicating that it is either the top-level entity, the initial state or the test input. Because of the support for the annotation system, the user no longer has to specify the top-level entity, etc., for every run of the compiler[1]: Having annotated certain functions, the user

[1]The top-level entity name can also be passed as a string, in which case the top-level entity annotation will be ignored.

now only has to tell CλasH compiler which file to translate, and the annotations will guide the compiler to the top entity.

Also, CλasH can be compiled and packaged as a separate library, ready to be installed in any existing Haskell system. Not only that, it can be packaged as part of the Haskell flagship compiler GHC, so that new users can start with CλasH straight out the box. Added to this package is GHC compiler which is customized with a *–vhdl* flag to quickly translate a file to VHDL. Also the GHC interpreter (GHCi) is expanded with the the *:vhdl* command, that translates currently loaded modules to VHDL. So, once a user has finalized his circuit design, remaining in the environment in which he simulated and tested his design, he can now translate the design to VHDL in that same environment.

6.1 Future Work

As with most master theses, the work done in this project is far from finished and leaves much room for future work. The CλasH language as a whole should be able to serve as a good basis for much research and many assignments to come.

6.1.1 general recursion

At the moment, only a limited subset of Haskell is translatable, and a crucial aspects of functional programming, recursion, is missing. However, as the reduction circuit shows, lacking recursion does not mean we can not not describe real world circuits. Not having recursion does however mean that we can not, for example, describe parametric tree structures.

The reason we lack recursion is two-fold: The first reason is that the normalization process can not do compile-time evaluation, and as such can not determine when the recursion step should stop. The thesis of Kooijman [26] explains this in greater detail. The second reason is that, unless we resort to cumbersome proof builders, general recursion and type-level programming can not be properly combined in the Haskell's current type system. Something we witnessed during the exploration of the GADT-based fixed-size vectors in Chapter 3.

New source language

A possible solution would be to switch from the current source language Haskell to a source language that is also a functional language but has *real* dependent types. This work would fit well in a master's thesis where a student would investigate dependent type systems, and languages that have them. He or she should then try to implement non-trivial recursive vector function in those dependently typed languages, and determine which of those functional languages would be a good candidate as the new source language for CλasH.

Type level invariants

The work of Schrijvers et al. [36] describes how type-level invariants can be implemented and used within Haskells type system, they also mention that they have implemented these type level invariants in a custom version of GHC. These invariants allow a developer to specify proofs for such things as the commutativity of addition at the type-level, instead of using a GADT as we saw in Chapter 3. However, a user still has to specify when these proofs have to be used. A possible assignment would be to investigate the use of these type-level invariants in greater details, and determine how the current VHDL translator should handle (ignore) them.

6.1.2 LISTS: VECTORS WITH A DYNAMIC LENGTH

In the case study of Chapter 4 we saw that the description of the input buffer using lists was concise, straightforward and easy to understand. The problem was of course that the *drop* function changed the length of the list by a variable amount: either by zero, by one or by two elements. It is impossible to write such a function with fixed-size vectors, so what we did was introduce a shift register combined with some write functionality. The other option that was presented, was a circular buffer. Both of these options are fairly standard in hardware, not only that, there are probably not a lot of other options to mimic the behavior of dynamic lists.

As it seems there are not many other options to mimic the behavior of dynamic lists, it might be possible to automate the process of converting a set of operations that work on a list to a set of operations that update the read and write pointers of either a circular or shift buffer. A possible approach is to use Template Haskell to analyze the AST of the functions and determine the read and write patterns of the dynamic lists and create corresponding read and write pointers. Such an analysis would probably not work for general the general case, but still a (small) subset of functions working with lists could be translated to a fixed-size vector equivalent.

6.1.3 STATIC LOOP UNROLLING

As explained in an earlier subsection it is currently not possible to make a recursive description in CλasH. However, there are techniques to eliminate the recursive calls while still preserving the meaning of the recursive description, thus allowing certain recursive hardware descriptions in CλasH. The technique discussed and explored here is *loop unrolling*. Specifically: static loop unrolling of number-guided recursive functions. What we mean by *static* is that the loop unrolling happens at compile-time, which is important for CλasH as the compiler needs to know the unrolled structure of the description to be able to generate hardware for it. The kind of recursive functions we are concerned with are the number-guided recursive functions, meaning that the descriptions bases its decision to make an additional recursive call on one of its numeric input parameters. Another common way to write recursive description are those based on pattern matching on a recursive data structure (such as lists); these descriptions are however not discussed in this subsection.

As an example of a number-guided recursive function in CλasH, we examine an adder tree, whose recursive description we see in Code Snippet 6.1.

CODE SNIPPET 6.1 (*Adder Tree*).

```
treeSum i xs | i < 1     = head xs
             | otherwise = let (a, b) = split xs
                           in (treeSum (i − 1) a) + (treeSum (i − 1) b)
```

In the description of *treeSum*, the variable *i* is used as the number guiding the recursion. It also dictates that the size the input vector should have: the size of the input vector has to be a power of two. The variable *xs* is this input vector, containing the elements we want to sum. The *split* function splits an input vector into two equally sized halves; it is an unsafe operation in the sense that it does not check that the input vector has an even length. The function *treeSum* recursively adds the two halves of its input vector as long as the recursion guiding variable *i* (the depth of the adder tree) is larger than zero. Once *i* drops to zero only a single element is returned. The intuitive structure for a tree adder (with a depth of 3) based on this description is shown in Figure 6.1 on page 55.

The basic idea is now to unroll the recursive description of Code Snippet 6.1 given that the depth of the tree, the variable *i*, is set to 3. Not only should we simply unroll the recursive description, but also simplify the description, as the choice elements guiding the recursion should not be seen in the structure of the hardware. In the following code snippets we will walk through the process of simplifying and unrolling until we have a description of a tree adder with a depth of 3 and no longer contains any recursive calls:

Code Snippet 6.2 (*Step 1: Simplify*).

```
treeSum 3 xs | 3 < 1       = head xs
             | otherwise = let (a, b) = split xs
                           in (treeSum 2 a) + (treeSum 2 b)
```

In the first step we replace all occurrences of the variable i with the value 3, and also reduce the arithmetic operations applied on i, such as $i - 1$, to single values.

Code Snippet 6.3 (*Step 2: Unroll and Simplify*).

```
treeSum xs      = let (a, b) = split xs
                  in (treeSum2 2 a) + (treeSum2 2 b)
treeSum2 2 xs  |  2 < 1       = head xs
               |  otherwise = let (a, b) = split xs
                              in (treeSum2 1 a) + (treesum2 1 b)
```

We now unroll the recursive call inside *treeSum* and call it *treeSum2*, once again replacing all occurrences of i by its value and reducing all the arithmetic operations applied on i. We also simplify *treeSum* even further by removing the guarded commands whose guard, $3 < 1$, evaluates to *false*. We also remove the depth parameter of the function as it no longer servers any purpose.

Code Snippet 6.4 (*Step 3: Unroll and Simplify*).

```
treeSum xs      = let (a, b) = split xs
                  in (treeSum2 a) + (treeSum2 b)
treeSum2 xs     = let (a, b) = split xs
                  in (treeSum1 1 a) + (treesum1 1 b)
treeSum1 1 xs  |  1 < 1       = head xs
               |  otherwise = let (a, b) = split xs
                              in (treeSum1 0 a) + (treeSum1 0 b)
```

We now unroll the recursive call of *treeSum2* in the same way as we did for *treeSum* and also apply the same simplifications. Also notice that because *treeSum2* no longer has a depth parameter, that the call to *treeSum2* made by *treeSum* no longer supplies this depth argument either.

Code Snippet 6.5 (*Step 4: Unroll and Simplify*).

```
treeSum xs      = let (a, b) = split xs
                  in (treeSum2 a) + (treeSum2 b)
treeSum2 xs     = let (a, b) = split xs
                  in (treeSum1 a) + (treesum1 b)
treeSum1 xs     = let (a, b) = split xs
                  in (treeSum0 0 a) + (treeSum0 0 b)
treeSum0 0 xs | 0 < 1       = head xs
              | otherwise = let (a, b) = split xs
                            in (treeSum0 (−1) a) + (treeSum0 (−1) b)
```

The last unroll step unrolls the recursive call of *treeSum1* and applies the aforementioned simplifications on *treeSum2*, *treeSum1* and of course the new function *treeSum0*.

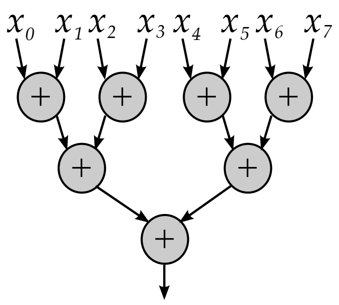

Figure 6.1: Tree Adder (Depth = 3)

Code Snippet 6.6 (*Step 5 (final): Simplify*).

```
treeSum xs  = let (a, b) = split xs
              in (treeSum2 a) + (treeSum2 b)
treeSum2 xs = let (a, b) = split xs
              in (treeSum1 a) + (treesum1 b)
treeSum1 xs = let (a, b) = split xs
              in (treeSum0 a) + (treeSum0 b)
treeSum0 xs = head xs
```

The guards in the *treeSum0* function are now both *true*, the arithmetic relation $0 < 1$, and the default case (*otherwise*). We follow the regular semantics of guarded commands, meaning that the command guarded by the first guard that yields *true* is evaluated, resulting in the new definition of *treeSum0*. This new simplified version of *treeSum0* has no recursion, and so unrolling stops. We now have a definition of *treeSum* (Code Snippet 6.6) that is specialized for a depth of 3, and no longer contains any recursive calls and can thus be compiled by CλasH.

The above unrolling and simplifying steps are specialized for the *treeSum* example and are based on the general unrolling and simplification flow described by Lynagh [28]. His work describes a set of Template Haskell functions that together unroll and simplify recursive functions discussed in this subsection: number-guided recursive functions. These unroll and simplification functions were originally meant for optimization purposes. The exact unroll and simplify flow as performed by that Template Haskell functions in the work of Lynagh [28] does not match the example of the tree adder we saw earlier. The Template Haskell functions of Lynagh [28] inline recursive calls as anonymous functions, instead of generating new functions as we saw in the earlier example.

We generated new functions for each unroll and simplification step in our example for the purposes of presentation, as nested inlined anonymous functions are much harder to read. The behavior of the set of generated functions is of course the same as the single nested function.

The removal of unused function parameters is also not performed by the original Template Haskell functions described by Lynagh [28]. From an optimization standpoint this makes sense, as you might not want to unroll the entire loop, and therefor want to keep the recursion guiding parameter intact. For CλasH we always want to completely unroll a function, so that is why we also want to remove the unused function parameters.

The function signature of the Template Haskell *unroll* function is the following:

```
type Depth = Maybe Int
type Argument = Int
unroll :: Depth → Argument → Lit → Dec → Dec
```

The first argument of the *unroll* function, the depth, indicates how many recursive steps should be unrolled. The depth parameter is of type **Maybe**, and is a safety parameter indicating how many recursive steps should be unrolled. If the depth parameter is set to *Nothing*, the *unroll* function can potentially unroll forever. This is why the original simplification process called by the *unroll* function does not remove the recursion guiding parameter: If, for example, the *unroll* function is told to unroll only 4 recursive calls, then the original recursive function should still exists for the calls that were potentially not unrolled: this recursive call still needs the recursion guiding parameter. For CλasH this does not make sense as we always want to completely unroll a recursive function. The parameter is however left in for CλasH, so a developer can use it to check his expectations of the recursive depth of his description.

The second parameter of the *unroll* function indicates which argument of the recursive function is the recursion guiding parameter. The third parameter is the literal that will be substituted for the recursion guiding parameter. The fourth and final argument is the AST representation of the function we want to unroll. This AST can be acquired by quoting the recursive function inside the special $[d|...\rrbracket$ brackets.

If we want to make a type-safe tree adder with a depth of 3, we have to specify that the input vector has to be of size 8. Due to the fact that the GHC compiler typechecks Haskell code within quotation brackets we can not simply ask for the AST of the following function definition:

```
treeSum :: Vector D8 (SizedWord D8) → (SizedWord D8)
treeSum i xs = ...
```

The typechecker will complain that number of arguments given by the signature does not match the number of arguments of the function body. We will thus have to apply some Template Haskell trickery to get a matching function signature and body for the *treeSum* function. The final declaration for a tree adder with a depth of 3 is shown in Code Snippet 6.7.

Code Snippet 6.7 (*Adder Tree (Depth = 3)).*

```
$(do
   [typ, _] ← [d|{
treeSum :: Vector D8 (SizedWord D8) → (SizedWord D8);
treeSum xs = ⊥
   }⟧
   [func] ← [d|{
treeSum i xs | i < 1       = head xs
             | otherwise = let (a, b) = split xs
                           in (treeSum (i − 1) a) + (treeSum (i − 1) b)
   }⟧
   let func′ = unroll Nothing 0 (IntegerL 3) func
   return [typ, func′])
```

The description starts with a splice statement, because we want the unrolled *treeSum* function to be spliced back into the code. The next thing we see is the Template Haskell trickery mentioned earlier, a quoted declaration of the type signature we want, accompanied by a function body that has the correct number of arguments, but whose output is $\perp$. As you can see, the variable *typ* now holds the AST of the function signature, and the AST belonging to the body is simply thrown away. The next quotation statement returns the AST of the recursive *treeSum* function. The third statement uses the *unroll* function to give that AST that represents the unrolled version of the *treeSum* function. As you can see, the depth argument of *unroll* is set to *Nothing* as we want a fully unrolled function, no matter how deep the recursion goes. The fourth and final statement then returns both the AST of the type signature we originally wanted, combined with the AST of the unrolled and simplified version of the tree adder. These two ASTs are then spliced back into the resulting code which will now pass compilation in CλasH. Once compiled, the structure of the described hardware will match what we see in Figure 6.1.

We slightly refactored the original work of Lynagh [28] for CλasH so that it would work with the latest version of Template Haskell. We also added the simplification step that removes the recursion guiding parameter. We explored and extended the work of Lynagh [28] not only to allow some form of recursive functions in CλasH, but also to have a proof of concept to indicate that Template Haskell can greatly increase the expressiveness of CλasH. Future works thus not only lies in expanding the *unroll* function to work for pattern-match based recursive functions, but also to explore further uses of Template Haskell in CλasH. On a shorter timescale we should also explore the limits of the recursive functions we can now describe using the current implementation of the *unroll* function.

6.1.4 graphical circuit designer

In a structural design language like CλasH you often describe a circuit with a certain hierarchy in mind. In many cases it would prove useful if the designer has a graphical overview of this hierarchy. This gives rise to the idea of a graphical designer for CλasH, with a Simulink like interface. Like Simulink, you would connect blocks with a certain functionality to each other using wires. To support the idea of hierarchy these blocks should be able to be built from other blocks.

We can have predefined blocks that contain certain default operations like the basic arithmetic and bit level functions. And similar to Simulink, a designer should also be able make custom blocks that contain function description in CλasH. It should always be possible to save the graphical design as a set of CλasH source file, where a project file would then contain all the block placement information, etc. It should also be possible to load textual designs, and either use automatic or user-guided block placement in the graphical environment.

The designer should also be able to communicate with external VHDL synthesis and modeling tools, so that you can easily run a post-synthesis simulation of your design and verify that your design behaves correctly when delays are introduced.

The graphical circuit designer should also have extended support for simulation, in which a user can step through a simulation and view both the output and state for that specific iteration. A user should be able to have several simulation input sources: manual input, a set of default input generators, or a custom input generator. Related to this, is that blocks containing state should have an interface to set their initial state.

We can think of a lot more functionality, but the ideas set out in this subsection should give a general idea of what a graphical designer for CλasH could look like.

6.1.5 edif support

At the moment, some CλasH designs introduce so-called null slices: empty arrays. These null slices are not accepted by every VHDL synthesis tool. Translating CλasH designs to EDIF would hopefully make it more tool independent. Such an endeavor should not be taken lightly however, besides having to redesign certain optimizations steps that the VHDL synthesis tools do for us now, we will also run into technology dependencies. Because even though there are certain standard components in the EDIF specification, most hardware vendors specify their own components. And not using those components will certainly result in suboptimal hardware designs. Just documenting the possible implications of switching to EDIF as a target language for CλasH would be a large task indeed.

6.1.6 VHDL BLACK BOX FUNCTIONS

In some situations you might have an optimized Intellectual Property (IP) block that you want to incorporate in your CλasH design. The problem is of course, that the IP block is most likely supplied as a binary file, and in rare cases described in a language like VHDL or Verilog. An example of such a predefined block is for example floating point adder akin to the one that would be used for the design of the reduction circuit of Chapter 4. We will use this example as we describe a solution to use existing IP blocks.

If we wanted to replace the simplified adder by the real IP block, we have to define a placeholder for it. We have two options for this placeholder as far as functionality goes:

- The placeholder design ignores the input, and assigns $\perp$ to the output, making simulation of the new design impossible; the design will however compile. In the reduction circuit design this is a valid decision, as the behavior of the control logic was independent of the calculated values. This approach is of course not valid for designs in general.
- We implement something that is functionally equivalent to the IP block, so that we can at least simulate it. However, to speed up the replacement design we can choose a implementation that is not translatable to VHDL, meaning that we could use Haskell lists for example.

Another problem is that the IP block might have defined its own datatypes that it uses as input and output, and which might not belong to the primitives CλasH can currently translate.

For both the placeholder and type problem we can use the new annotation system in GHC. We can annotate the placeholder as a black box, so that CλasH knows not to translate this function, but only define an internal VHDL entity (and no architecture) for it (so that it can be port mapped in other entities).

For the type problem we can define a simple Haskell datatype, and annotate it so that CλasH knows that this is a black box type. CλasH will then create an invalid VHDL type declaration, which the user will have to fill in with the actual type data after translation. This way the user will *not* have to define VHDL functions that translate between the built-in CλasH types and VHDL types introduced by the IP block.

APPENDIX A

Vector Function Templates

With the decision made that CλasH, for the time being, will only support the pre-defined functions of the TFVEC library library, we have to make VHDL templates for these functions. All of the templates expect vectors that only contain primitives such as Bits, Integers or other vectors containing only primitives. When that condition is met we can observe that all translations from CλasH to VHDL are very straightforward.

Select Head Element

The type signature:

$$head :: PositiveT\ s \Rightarrow Vector\ s\ a \rightarrow a$$

Corresponding VHDL:

```
function head (veci : array_of_a)
  return a is
begin
  return veci (0);
end;
```

Select Last Element

The type signature:

$$last :: PositiveT\ s \Rightarrow Vector\ s\ a \rightarrow a$$

Corresponding VHDL:

```
function last (veci : array_of_a)
  return a is
begin
  return veci (veci'length − 1);
end;
```

Select Tail Elements

The type signature:

$$tail :: PositiveT\ s \Rightarrow Vector\ s\ a \rightarrow Vector\ (Pred\ s)\ a$$

Corresponding VHDL:

```
function tail (veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci'length − 2);
begin
  res := veci (1 to veci'length − 1);
  return res;
end;
```

Select Initial Elements

The type signature:

$$init :: PositiveT\ s \Rightarrow Vector\ s\ a \rightarrow Vector\ (Pred\ s)\ a$$

Corresponding VHDL:

```
function init (veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci'length − 2);
begin
  res := veci (0 to veci'length − 2);
  return res;
end;
```

Element Selection

The type signature:

```
(!) :: (PositiveT s
       , NaturalT u
       , (s > u)∼True) ⇒ Vector s a → RangedWord u → a
```

Corresponding VHDL:

```
function exclamation (veci : array_of_a;
                      ix   : natural    )
  return a is
begin
  return veci (ix);
end;
```

Element Replacement

The type signature:

$$\begin{aligned}&\mathit{replace} :: (\mathit{PositiveT}\ s\\ &\qquad ,\mathit{NaturalT}\ u\\ &\qquad ,(s > u) \sim \mathit{True}) \Rightarrow \mathit{Vector}\ s\ a \rightarrow \mathit{RangedWord}\ u \rightarrow a \rightarrow \mathit{Vector}\ s\ a\end{aligned}$$

Corresponding VHDL:

```
function replace (veci : array_of_a;
                  ix   : unsigned   ;
                  el   : a          )
  return array_of_a is
  variable res : array_of_a (0 to veci'length − 1);
begin
  res := veci;
  res (to_integer (i)) := a;
  return res;
end;
```

Return first i Elements

The type signature:

$$\mathit{take} :: \mathit{NaturalT}\ i \Rightarrow i \rightarrow \mathit{Vector}\ s\ a \rightarrow \mathit{Vector}\ (\mathit{Min}\ s\ i)\ a$$

Corresponding VHDL:

```
function minimum  ( nLeft  : natural;
                    nRight : natural)
  return natural is
begin
  if nLeft < nRight then
    return nLeft;
  else
    return nRight;
  end if ;
end;
function take (n       : natural    ;
               veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to (minimum (n, veci'length)) − 1);
begin
  res := veci (0 to (minimum (n, veci'length)) − 1)
  return res;
end;
```

Return last (s - i) Elements

The type signature:

$$drop :: NaturalT\ i \Rightarrow i \rightarrow Vector\ s\ a \rightarrow Vector\ (s - (Min\ s\ i))\ a$$

Corresponding VHDL:

```
function drop (n    : natural   ;
               veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci'length - n - 1);
begin
  res := veci (n to veci'length - 1)
  return res;
end;
```

Add element at the start of the vector

The type signature:

$$(+>) :: a \rightarrow Vector\ s\ a \rightarrow Vector\ (Succ\ s)\ a$$

Corresponding VHDL:

```
function plusgt (el   : a         ;
                 veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci'length);
begin
  res := el & veci;
  return res;
end;
```

Add element at the end of the vector

The type signature:

$$(<+) :: Vector\ s\ a \rightarrow a \rightarrow Vector\ (Succ\ s)\ a$$

Corresponding VHDL:

```
function ltplus (veci : array_of_a;
                 el   : a         )
  return array_of_a is
  variable res : array_of_a (0 to veci'length);
begin
  res := veci & el;
  return res;
end;
```

Concatenate two vector

The type signature:

$$(+\!\!+) :: \mathit{Vector}\ s\ a \rightarrow \mathit{Vector}\ s2\ a \rightarrow \mathit{Vector}\ (s + s2)\ a$$

Corresponding VHDL:

```
function plusplus (veci1 : array_of_a;
                   veci2 : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci1'length + veci2'length − 1);
begin
  res := veci1 & veci2;
  return res;
end;
```

Zip two vectors to a vector of tuples

The type signature:

$$\mathit{zip} :: \mathit{Vector}\ s\ a \rightarrow \mathit{Vector}\ s\ b \rightarrow \mathit{Vector}\ s\ (a, b)$$

Corresponding VHDL:

```
zipVector : for n in 0 to (veco'length − 1) generate
  veco (n).A ⇐ veci1 (n);
  veco (n).B ⇐ veci2 (n);
end generate zipVector;
```

Unzip vector of tuples to a tuple of vectors

The type signature:

$$\mathit{unzip} :: \mathit{Vector}\ s\ (a, b) \rightarrow (\mathit{Vector}\ s\ a, \mathit{Vector}\ s\ b)$$

Corresponding VHDL:

```
unzipVector : for n in 0 to (veco'length − 1) generate
  veco.A (n) ⇐ veci (n).A;
  veco.B (n) ⇐ veci (n).B;
end generate unzipVector;
```

Select every s'th Element

The type signature:

$$\begin{aligned}\mathit{select} :: {} & (\mathit{NaturalT}\ f\\ & , \mathit{NaturalT}\ s\\ & , \mathit{NaturalT}\ n\\ & , (f < i) \sim \mathit{True}\\ & , (((s * n) + f) \leqslant i) \sim \mathit{True}) \Rightarrow f \rightarrow s \rightarrow n \rightarrow \mathit{Vector}\ i\ a \rightarrow \mathit{Vector}\ n\ a\end{aligned}$$

Corresponding VHDL:

```
function select (f    : natural    ;
                 n    : natural    ;
                 s    : natural    ;
                 veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to n − 1);
begin
  for i in res'range loop
    res (i) := veci (f + i ∗ s);
  end loop;
  return res;
end;
```

Map function on all elements

The type signature:

$$\textbf{map} :: (a \rightarrow b) \rightarrow Vector\ s\ a \rightarrow Vector\ s\ b$$

Corresponding VHDL:

```
mapVector : for n in 0 to (veco'length − 1) generate
begin
  comp_ins : entity f
    port map (input  ⇒ veci (n) ,
              output ⇒ veco (n),
              clock  ⇒ clock    );
end generate mapVector;
```

Zip two vector with a function

The type signature:

$$zipWith :: (a \rightarrow b \rightarrow c) \rightarrow Vector\ s\ a \rightarrow Vector\ s\ b \rightarrow Vector\ s\ c$$

Corresponding VHDL:

```
zipWithVector : for n in 0 to (veco'length − 1) generate
begin
  comp_ins : entity f
    port map (input1 ⇒ veci1 (n),
              input2 ⇒ veci2 (n),
              output ⇒ veco (n) ,
              clock  ⇒ clock    );
end generate zipWithVector;
```

Shift vector 1 position to the left

The type signature:

$$\begin{aligned}shiftl :: (&PositiveT\ s\\ ,&NaturalT\ n\\ ,&n{\sim}Pred\quad s\\ ,&s{\sim}Succ\quad n) \Rightarrow Vector\ s\ a \rightarrow a \rightarrow Vector\ s\ a\end{aligned}$$

Corresponding VHDL:

```
function shiftl (veci : array_of_a;
                 el   : a         )
   return array_of_a is
   variable res : array_of_a (0 to veci'length − 1);
begin
   res := el & init (veci);
   return res;
end;
```

Shift vector 1 position to the right

The type signature:

$$\begin{aligned}shiftr :: (&PositiveT\ s\\ ,&NaturalT\ n\\ ,&n{\sim}Pred\quad s\\ ,&s{\sim}Succ\quad n) \Rightarrow Vector\ s\ a \rightarrow a \rightarrow Vector\ s\ a\end{aligned}$$

Corresponding VHDL:

```
function shiftr (veci : array_of_a;
                 el   : a         )
   return array_of_a is
   variable res : array_of_a (0 to veci'length − 1);
begin
   res := tail (veci) & el;
   return res;
end;
```

Check if vector is null

The type signature:

$$null :: Vector\ D0\ a \rightarrow Bool$$

Corresponding VHDL:

```
function isnull (veci : array_of_a)
   return boolean is
begin
   return veci'length = 0;
end;
```

Rotate vector 1 position to the left

The type signature:

$$rotl :: NaturalT\ s \Rightarrow Vector\ s\ a \rightarrow Vector\ s\ a$$

Corresponding VHDL:

```
function rotl (veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci'length − 1);
begin
  if isnull (veci) then
    res := veci;
  else
    res := last (veci) & init (veci)
  return res;
end;
```

Rotate vector 1 position to the right

The type signature:

$$rotr :: NaturalT\ s \Rightarrow Vector\ s\ a \rightarrow Vector\ s\ a$$

Corresponding VHDL:

```
function rotr (veci : array_of_a)
  return array_of_a is
  variable res : array_of_a (0 to veci'length − 1);
begin
  if isnull (veci) then
    res := veci;
  else
    res := tail (veci) & head (veci)
  return res;
end;
```

Generate a vector of 's' copies of element 'a'

The type signature:

$$copy :: NaturalT\ s \Rightarrow s \rightarrow a \rightarrow Vector\ s\ a$$

Corresponding VHDL:

```
function copy (n : natural;
               el : a       )
  return array_of_a is
  variable res : array_of_a (0 to n − 1) := (others ⇒ el);
begin
  return res;
end;
```

Fold(l) function over a Vector

The type signature:

$$foldl :: (a \to b \to a) \to a \to Vector\ s\ b \to a$$

Corresponding VHDL, if the vector is null:

```
a ⇐ start;
```

Corresponding VHDL, otherwise:

```
foldlVectorBlock : block
  signal tmp : array_of_a_with_length_of_vec;
begin
  foldlVector : for n in 0 to (veci'length − 1) generate
  begin
    firstcell : if n = 0 generate
    begin
      comp_ins : entity f
        port map (output ⇒ tmp (n),
                  input1 ⇒ start    ,
                  input2 ⇒ veci (n),
                  clock  ⇒ clock    );
    end generate firstcell;
    othercells : if n /= 0 generate
    begin
      comp_ins : entity f
        port map (output ⇒ tmp (n)      ,
                  input1 ⇒ tmp (n − 1),
                  input2 ⇒ veci (n)     ,
                  clock  ⇒ clock        );
    end generate othercells;
  end generate foldlVector;
  a ⇐ tmp (veci'length − 1);
end block foldlVectorBlock;
```

Concatenate vector of vectors

The type signature:

$$concat :: Vector\ s1\ (Vector\ s2\ a) \to Vector\ (s1 * s2)\ a$$

Corresponding VHDL:

```
concatVector : for n in 0 to veci'length generate
  veco ((n ∗ veci_1'length) to ((n ∗ veci_1'length) + (veci_1'length − 1))) ⇐ veci (n)
end generate concatVector;
```

Fold(r) function over a Vector

The type signature:

$$foldr :: (b \to a \to a) \to a \to Vector\ s\ b \to a$$

Corresponding VHDL, if the vector is null:

```
a <= start;
```

Corresponding VHDL, otherwise:

```
foldrVectorBlock : block
   signal tmp : array_of_a_with_length_of_vec;
begin
   foldrVector : for n in (veci'length - 1) downto 0 generate
   begin
      firstcell : if n = (veci'length - 1) generate
      begin
         comp_ins : entity f
            port map (output => tmp (n),
                      input1 => start    ,
                      input2 => veci (n),
                      clock  => clock    );
      end generate firstcell;
      othercells : if n /= (veci'length - 1) generate
      begin
         comp_ins : entity f
            port map (output => tmp (n)     ,
                      input1 => tmp (n + 1),
                      input2 => veci (n)    ,
                      clock  => clock        );
      end generate othercells;
   end generate foldrVector;
   a <= tmp (0);
end block foldrVectorBlock;
```

Reverse vector

The type signature:

$$reverse :: Vector\ s\ a \to Vector\ s\ a$$

Corresponding VHDL:

```
function reverse (veci : array_of_a)
   return array_of_a is
   variable res : array_of_a (0 to veci'length - 1);
begin
   for i in res'range loop
      res (vec'length - i - 1) := veci (i)
   end loop;
   return res;
end;
```

Iterate function 's' amount of times

The type signature:

$$iterate :: NaturalT\ s \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow Vector\ s\ a$$

Corresponding VHDL, if the vector is null:

```
veco ⇐ "";
```

Corresponding VHDL, otherwise:

```
iterateVectorBlock : block
  signal tmp : array_of_a_with_length_of_veco;
begin
  iterateVector : for n in 0 to (veco'length − 1) generate
  begin
    firstcell : if n = 0 generate
    begin
      tmp (n) ⇐ start
    end generate firstcell;
    othercells : if n /= 0 generate
    begin
      comp_ins : entity f
        port map (output ⇒ tmp (n)      ,
                  input  ⇒ tmp (n − 1),
                  clock  ⇒ clock        );
    end generate othercells;
  end generate iterateVector;
  veco ⇐ tmp;
end block iterateVectorBlock;
```

Generate vector of Size 's' by applying function 'f', 's' times

The type signature:

$$generate :: NaturalT\ s \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow Vector\ s\ a$$

Corresponding VHDL, if the vector is null:

```
veco ⇐ "";
```

Corresponding VHDL, otherwise:

```
generateVectorBlock : block
   signal tmp : array_of_a_with_length_of_veco;
begin
   generateVector : for n in 0 to (veco'length − 1) generate
   begin
      firstcell : if n = 0 generate
      begin
         comp_ins : entity f
            port map (output ⇒ tmp (n)     ,
                      input  ⇒ start       ,
                      clock  ⇒ clock       );
      end generate firstcell;
      othercells : if n /= 0 generate
      begin
         comp_ins : entity f
            port map (output ⇒ tmp (n)     ,
                      input  ⇒ tmp (n − 1),
                      clock  ⇒ clock       );
      end generate othercells;
   end generate generateVector;
   veco ⇐ tmp;
end block generateVectorBlock;
```

Appendix B

Test bench Generation

As the designs made in CλasH got larger, and the translation of Haskell code to generated VHDL needed to be checked for correctness, so too grew the need for automated test bench generation. Especially when we look at the number of lines of generated VHDL code produced from the translation of the reduction circuit design, we see that manual verification is no longer an option.

Another reason why you want a test bench in VHDL (when you are convinced the generated VHDL is correct), is if you want to do a post-synthesis simulation. You want to be able to quickly verify that the synthesized design behaves still behaves correctly when delay information is included.

The test bench generation design is split into two parts: The first part is the translation from Haskell input values to VHDL input values. The second part is the output format; you want to be able to quickly, and preferably mechanically, verify that the Haskell simulation output is equal to the VHDL output.

B.1 Input generation

The input generation part is concerned with translating the input values that are used for the Haskell simulation to input values used for the VHDL testbench. The simulation function (*run*) expects a list of values, one value to match each clock cycle. The problem is that CλasH does not support the translation of lists, only the translation of vectors from the TFVec library. But for our purposes here, we are not really interested in the list itself, only the values it contains. Lucky for us, when GHC desugars lists to Core [26], all the cons (:) operators are exposed. So we can now easily traverse the application of the cons operators and extract all the Core expressions that correspond with values contained in the list. This means that at the moment, CλasH partially supports Haskell lists, but only when the lists are used as containers for the test input.

As the values contained in the input list are supposed to be of a translatable type, the translation functionality that is already present for the translation of hardware descriptions can be re-used for the generation of the test bench inputs. So all that remains is a way to organize all the signals inside the test bench. This is done by generating signal declarations for all the test values inside the declaration part of the test bench architecture. Then, for all of these signals we generate a *block* statement, where we use the available translation code to give the signals the actual input values. The reason we make a *block* statement is so that we can generate extra signal declaration in case the value, is for example, a tuple: We then generate a signal declaration for both tuple elements, assign the values to these signals, and assign these signal to the actual test input signal.

All these test value signals are aggregated to form one big assignment to the input port of the generated design. The signals are aggregated so that the value at the input port changes every cycle of the clock, to correspond with the Haskell simulation. The signal assignment to the input port looks similar to the code below:

Code Snippet B.1 (*Test bench, input port assignment).*

```
inputport ⇐ testvalue0 after 0 ns,
            testvalue1 after 10 ns,
            testvalue2 after 20 ns,
            ...
```

B.2 Output format

To mimic the behaviour of the *run* function in Haskell, the test bench should print the value of the output port. If we route this output to a file, we can then easily, mechanically, compare it to the output of the *run* function. So the challenge is to find a general way to convert the values belonging to the generated VHDL types to a string.

In most Haskell designs, values of data structures can be converted to a printable string using the *show* function. This function can be specified by a developer, but in most cases it can also be automatically derived by a compiler. The idea behind automated derivation of the *show* function forms the basis for generating string output functions for the VHDL types.

The CλasH compiler has VHDL templates for *show*-like functions for all of the built-in types. The compiler can also generate extra *show* functions for aggregates (tuples and vectors) of these built-in types. These *show* functions output the same strings as their Haskell counterparts. Currently these *show* functions are only generated for: Bit, SignedWord, SignedInt, RangedWord, tuples and vectors.

Although custom datatypes are (partially) supported in CλasH, there is currently no proper support for generating VHDL *show* functions for these custom datatypes.

Unlike the built-in types, for which we know the exact implementation details of their *show* function, we do not now in advance what the implementation details are for the *show* functions belonging to the custom datatypes. This means we can not make a default VHDL template for these *show* functions up front, as we did for the *show* functions belonging to the built-in types. The temporary 'solution' that is currently in place is just to print the name of the custom datatype when the corresponding *show* function is called. In this sense, custom datatypes are improperly supported by the current CλasH compiler.

A real solution is to examine the automated derivation mechanism of GHC, and use something similar to generate the VHDL functions. In addition we can have the CλasH compiler search source-files of the circuit design for an implementation of the **Show** class, and try to translate the corresponding *show* function to VHDL. Both the research of further details, and an implementation, are left as future work.

Appendix C

Solutions for the Node Sharing Problem

In Chapter 2 we saw how feedback loops in many existing hardware Hardware Description Languages can cause problems when we try to evaluate the corresponding structure of a hardware description. The general solution is that, as we traverse the hardware graph we have to find out if we already visited a node, so that we know it is part of a feedback loop. The following sections give four possible solutions as to how we might determine if we visited a node earlier in the traversal.

C.1 Solution 1: Explicit Tagging

The fundamental difficulty is that we need a way to identify uniquely at least one node in every feedback loop, so that the graph traversal algorithms can determine whether a node has been seen before. This can be achieved by decorating the circuit specification with an explicit labeling function:

$$\mathit{label} :: \mathit{Signal}\ a \Rightarrow \mathit{Int} \rightarrow a \rightarrow a$$

Now labels can be introduced into a circuit specification; for example a labeled version of the *flip-flop* circuit might be written as follows:

$\mathit{flipflop}'\ x = r$
  where $r = \mathit{label}\ 100\ (\mathit{flipflop}\ x)$

The use of labeling solves the problem of traversing circuit graphs with feedback loops, at the cost of introducing two new problems. It forces a notational burden onto the circuit designer which has nothing to do with the hardware. Even worse, the labeling must be done correctly and yet cannot be checked by traversal algorithms.

Suppose that a specification contains two different components that were mistakenly given the same label. Simulation will not bring out this error, but the netlist will actually describe a different circuit than the one that was simulated. Later on the circuit will be fabricated using the erroneous netlist. No amount of simulation or formal methods will help if the circuit that is built does not match the one that was designed.

C.2 Solution 2: Monads

Monads have the potential to solve the problem of manually labeling circuits. Monads can be used to automate the passing of the traversal state from one computation to the next, while avoiding the naming errors. Thus a circuit specification might be written in a form like this:

```
circ a b = do
  p ← flipflop a
  q ← flipflop b
  x ← and p q
  return x
```

The monad would be defined so that a unique label is generated for each operation. That is enough to detect the feedback loops, and thus avoiding the problem of not knowing which exact structure corresponds with the circuit specification.

A problem is that the circuit specification is no longer a system of simultaneous equations. Instead, the specification is now a sequence of computations that, when executed, will yield the desired circuit. It now feels more like writing an imperative program to draw a circuit (as the sequence of computations is not allowed to be arbitrary due to the local scopes of the monadic operators), instead of defining the circuit directly.

C.3 Solution 3: Functional Meta-langauge

Instead of requiring the designer to insert labels by hand, or using monads, the labels could be inserted automatically by a program transformation [33]. Template Haskell provides the ability for a Haskell program to perform computations at compile time, which generate new code that can then be spliced into the program. Code Snippet C.1 shows a piece of code that uses the quotation brackets [*d*|...⟧ to define *circ_reg_rep* as an algebraic data type representing the code for a definition. The letter *d* inside the quotation brackets indicates that it is quoting a (function) definition. There are other quotation brackets as well, they are described in Appendix D, the appendix on Haskell and GHC constructs relevant to this thesis.

Code Snippet C.1 (*Register Definition in Template Haskell).*

```
circ_reg_rep =
[d|reg :: a → a → a
  reg enable x = r
    where
      r = flipflop (mux enable r x)⟧
```

So the function *reg* describes a register with an *enable* signal and an input *x*. By placing the *reg* function inside the [*d*|...⟧ quotation brackets, the value of *circ_reg_rep* is now the AST of the function *reg*, containing both the representation for the type declaration and the equation.

In Template Haskell, all aspects of Haskell which the ordinary programmer can use are also available to process the AST at program generation time. Thus a function that works on these ASTs, e.g. *transform_circuit*, is just an ordinary Haskell function definition. This *transform_circuit* function, could now execute a generalization of the manual labeling transformation described in Section C.1. The result of this transformation is a new code tree. Using the \$(...) syntax, we can then splice the new code into the program, and resume the compilation:

```
$(transform_circuit circ_defs_rep)
```

The designer could of course write this final transformed code directly, bypassing the need for metaprogramming. However, besides the negative consequences this might have (described in

Section C.1), using metaprogramming has another advantage over manual transformation: As the system evolves, the *transform_circuit* function can be updated to provide whatever new capabilities are found necessary (like logic probes to access internal signals), without the programmer having to rewrite all their existing circuit descriptions.

C.4 Solution 4: Observable Sharing

Observable sharing is an extension to call-by-need languages (like Haskell), which makes graph sharing observable [9], meaning we can see feedback loops in a graph. Observable sharing is added to Haskell by providing immutable reference cells, together with a reference equality test. A problem with *observable sharing* is that it is not an conservative extension of a pure functional language. It is a *side effect*, although in limited form, for which the semantic implications are not immediately apparent. This subsection will briefly describe how *observable sharing* could be implemented; for the consequences to the semantics of Haskell the reader is referred to the paper of Claessen and Sands [9]. Code Snippet C.2 shows the interface to provide the aforementioned references.

Code Snippet C.2 (*Reference Interface*).

```
type Ref a = ...
ref      :: a → Ref a
deref    :: Ref a → a
(⇔)      :: Ref a → Ref a → Bool
```

The next two examples show how to use the new constructs to detect sharing:

$$\textbf{let}\ x = \bot\ \textbf{in}\ (\textbf{let}\ r = \mathit{ref}\ x\ \textbf{in}\ r \Leftrightarrow r)$$
$$\textbf{let}\ x = \bot\ \textbf{in}\ \mathit{ref}\ x \Leftrightarrow \mathit{ref}\ x$$

In the first equation we create one reference, and compare it with itself, which of course yields *True*. In the second equation, we create two different references to the same variable, and so the comparison yields *False*. Now let us take a look how we can use this extension to help us symbolically evaluate circuits. Observe the two circuit descriptions in Code Snippet C.3.

Code Snippet C.3 (*Observable Circuits*).

```
circ1 = let output = dff output in output
circ2 = let output = dff (dff output) in output
```

In Haskell's denotational semantics, these two circuits are identical, since *circ2* is just recursive unfolding of *circ1*. But we would like to represent two different circuits; *circ1* has one flipflop and a loop, whereas *circ2* has two flipflops and a loop. If the type for signal representation included a reference, we could compare the identities of the flipflop components and conclude that in *circ1* all flipflops are identical, whereas in *circ2* we have two different flipflops.

We can now define a signal datatype in such a way that the creation of identities (references) happens transparently to the programmer. Code Snippet C.4 shows the Signal datatype from the paper of Claessen and Sands [9] in which Signal is symbolic, where it is either a variable name (a wire), or the result of a component which has been supplied with its input signals.

Code Snippet C.4 (*Signal Datatype*).

```
data Signal     = Var String | Comp (Ref (String, [Signal]))
comp name args = Comp (ref (name, args))
inv b           = comp "inv"  b
dff b           = comp "dff"  b
and a b         = comp "and" [a, b]
xor a b         = comp "xor"  [a, b]
```

In this way, a circuit like the *oscillate* circuit of Chapter 2 still creates a cyclic structure, but it is now possible to define a function which *observes* this cycle (using the $\Leftrightarrow$ operator) and therefor terminates when generating a netlist for the circuit.

Appendix D

Haskell Constructs & Extensions

This appendix serves as an introduction to certain language constructs in Haskell, and the GHC Extensions to Haskell, for programmers who have programmed with functional languages, but not with Haskell in particular. A reader completely unfamiliar with functional programming is therefor encouraged to read the introductory book on functional programming (in Haskell) by Hutton [18] (or any other introductory text) before continuing with this appendix.

Much of the information comes from either the Haskell 98 Report [34], or the Wiki-pages on **Haskell.org**. Most of the information found in the following sections usually feature only minor modifications to the original text; therefor, each section is marked with the author, title, and reference number of the original text.

D.1 Type Classes

Mark P. Jones - Type Classes with Functional Dependencies [23]

This section describes the *class declarations* that are used to introduce new type classes in Haskell, and the *instance declarations* that are used to populate them. Haskell uses a traditional Hindley-Milner [10, 17] polymorphic type system to provide static type semantics, but the type system has been extended with *type classes* (or just *classes*) that provide a structured way to introduce *overloaded* functions.

Class Declarations:

A class declaration specifies the name for a class and lists the member functions that each type in the class is expected to support. The actual types in each class—which are normally referred to as the *instances* of the class—are described using separate declarations. For example, an **Eq** type class, representing the set of equality types, might be introduced by the following declaration:

$$\begin{array}{l}\textbf{class}\ Eq\ a\ \textbf{where}\\ \quad (\equiv) :: a \to a \to Bool\end{array}$$

The type variable a that appears in both lines represents an arbitrary instance of the class. The intended reading of the declaration is that , if a variable is a particular instance of **Eq** type, then we can use the $(\equiv)$ operator at type $a \to a \to Bool$ to compare values of type **a**.

Qualified Types:

As we have already indicated, the restriction on the use of the equality operator is reflected in the type that is assigned to it: $(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$
Types that are restricted by a predicate like this are referred to as *qualified types* [24]. Such types will be assigned to any function that takes either direct or indirect use of the member functions of a type class at some unspecified type. For example, the functions:

$$\begin{aligned}&member\ x\ xs = any\ (x \equiv)\ xs\\&subset\ xs\ ys - all\ (\lambda a \rightarrow member\ x\ ys)\ xs\end{aligned}$$

will be assigned types:

$$\begin{aligned}&member :: Eq\ a \Rightarrow a \rightarrow [a] \rightarrow Bool\\&subset :: Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Bool\end{aligned}$$

Superclasses:

Classes may be arranged in a hierarchy, and may have multiple member functions. The following example illustrates a declaration of the **Ord** type class, which contains the types whose elements can be ordered using strict $(<)$ and non-strict $(\leqslant)$ comparison operators:

$$\begin{aligned}&\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where}\\&\quad (<), (\leqslant) :: a \rightarrow a \rightarrow Bool\end{aligned}$$

In this particular context, the $\Rightarrow$ symbol should not be read as implication; in fact reverse implication would be a more accurate reading, the intention being that every instance of **Ord** is also an instance of **Eq**. Thus **Eq** plays the role of a *superclass* of **Ord**. This mechanism allows the programmer to specify an expected relationship between classes: it is the compiler's responsibility to ensure that this property is satisfied, or to produce an error diagnostic if it is not.

Instance Declarations:

The instances of any given class are described by a collection of instance declarations. For example, the following declarations show how one might define equality for booleans, and for pairs:

$$\begin{aligned}&\textbf{instance}\ Eq\ Bool\ \textbf{where}\\&\quad x \equiv y = \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ \neg\ y\\&\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)\ \textbf{where}\\&\quad (x, y) \equiv (u, v) = (x \equiv u \wedge y \equiv v)\end{aligned}$$

The first line of the second instance declaration tells us that an equality on values of types **a** and **b** is needed to provide an equality on pairs of type **(a, b)**. No such preconditions are needed for the definition of equality on booleans. Even with just these two declarations, we have already specified an equality operation on the infinite family of types that can be constructed from **Bool** by repeated use of pairing. Additional declarations, which may be distributed over many modules, can be used to extend the class to include other data types.

Extension: Multi-parameter type classes

Type classes permit multiple type parameters, and so type classes can be seen as relations on types. For example, in a predicate of the form $R\ a\ b$, **R** is interpreted as a two-place relation between types, so $R\ a\ b$ has to be read as the assertion that **a** and **b** are related by **R**. This is a natural generalization of the one parameter case because sets are just one-place relations. More generally, we can interpret an n parameter class by an n-place relation on types.

One of the most commonly suggested applications for multiple parameter type classes is to provide uniform interfaces to a wide range of collection types [25]. Such types might be expected to offer ways to construct empty collections, to insert values, to test for membership, and so on. The following declaration, greatly simplified for the purposes of presentation, introduces a two parameter class Collects that could be used as the starting point for such a project:

```
class Collects e ce where
empty  :: ce
insert :: e → ce → ce
member :: e → ce → Bool
```

The type variable **e** used here represents the element type, while **ce** is the type of the collection itself. Within this framework, we might want to define instances of this class for lists or characteristic functions (both of which can be used to represent collections of any equality type), bit sets (which can be used to represent collections of characters), or hash tables (which can be used to represent any collection whose elements have a hash function). Omitting standard implementation details, this would lead to the following declarations:

```
instance Eq e ⇒ Collects e [e] where ...
instance Eq e ⇒ Collects e (e → Bool) where ...
instance Collects Char BitSet where ...
instance (Hashable e, Collects e ce) ⇒ Collects e (Array Int ce) where ...
```

All this looks quite promising; we have a class and a range of interesting implementations. Unfortunately, there are some serious problems with the class declaration. First, the *empty* function has an ambiguous type:

```
empty :: Collects e ce ⇒ ce
```

By 'ambiguous' we mean that there is a type variable **e** that appears on the left of the ⇒ symbol, but not on the right. The problem with this is that, according to the theoretical foundations of Haskell overloading, we cannot guarantee a well defined semantics for any term with an ambiguous type. For this reason, a Haskell system will reject any attempt to define or use such terms.

We can sidestep this specific problem by removing the *empty* member from the class declaration. However, although the remaining members, *insert* and *member*, do not have ambiguous types, we still run into problems when we try to use them. For example, consider the following two functions:

```
f x y coll = insert x (insert y coll)
g coll     = f True 'a' coll
```

for which the following types are inferred:

```
f :: (Collects a c, Collects b c) ⇒ a → b → c → c
g :: (Collects Bool c, Collects Char c) ⇒ c → c
```

Notice that the type for f allows the parameters x and y to be assigned different types, even though it attempts to insert each of the two values, one after the other, into the same collection, *coll*. If we hope to model collections that contain only one type of value, then this is clearly an inaccurate type. Worse still, the definition for g is accepted, without causing a type error. Thus the error in this code will not be detected at the point of definition, but only at the point of use, which might not even be in the same module. Obviously, we would prefer to avoid these problems, eliminating ambiguities, inferring more accurate types, and providing earlier detection of type errors.

D.2 Extension: Functional Dependencies

Mark P. Jones - Type Classes with Functional Dependencies [23]

Many of the problems of *Multi-Parameter Type Classes* can be avoided by giving programmers an opportunity to specify the desired relations on types more precisely. The key idea is to allow the definitions of type classes to be annotated with *Functional Dependencies*—an idea that originates in the theory of relational databases. Functional dependencies are used to constrain the parameters of type classes. They let you state that in a *Multi-Parameter Type Classes*, one of the parameters can be determined from the others, so that the parameter determined by the others can, for example, be the return type but none of the argument types of some of the methods.

For example, we can annotate the original class definition of **Collects** with a dependency $ce \leadsto e$, to be read as "*ce* uniquely determines *e*":

```
class Collects e ce | ce ⇝ e where
empty  :: ce
insert :: e → ce → ce
member :: e → ce → Bool
```

Now, given two predicates *Collects a c* and *Collects b c* with the same collection type **c**, we can immediately infer from the *Functional Dependencies* that $a \equiv b$. This simple improvement allows us to infer a more specific type for f:

$$f :: (Collects\ e\ c) \Rightarrow e \rightarrow e \rightarrow c \rightarrow c$$

An immediate consequence is that the body of *g*, *f True* **'a'**, will now trigger a type error.

D.3 Extension: Type Families

Haskell.org - GHC/Type families [16]

Indexed type families, or *type families* for short, are type constructors that represent *sets of types*. Set member are denoted by supplying the type family constructor with type parameters, which are called *type indices*. The difference between 'vanilla' parameterized type constructors and family constructors is much like between parametrically polymorphic functions and methods of type classes. Parametric polymorphic functions behave the same at all type instances, whereas class methods can change their behavior in dependence of the class type parameters, as can be seen in Figure D.1.

```
(++) :: [a] → [a] → [a]               class Append a where
[]       ++ ys = ys                      append :: a → a → a
(x : xs) ++ ys = x : (xs ++ ys)        instance Append Integer where
                                         append a b = (10 * a) + b
                                       instance Append [a] where
                                         append a b = a ++ b
```

Figure D.1: Parametric Polymorphism vs Type Class Polymorphism

Similarly, 'vanilla' type constructors imply the same data representation for all type instances, but family constructors can have varying representations types for varying type indices. Figure D.2 shows us an example.

$$\begin{array}{ll}
\textbf{type}\ \mathit{Edge}\ g = (\mathit{Node}\ g, \mathit{Node}\ g) & \textbf{type}\ \mathit{family}\ \mathit{Edge}\ g :: \star \\
 & \textbf{type instance}\ \mathit{Edge}\ \mathit{Int} \quad = \\
 & \quad (\mathit{Node}\ \mathit{Int}, \quad \mathit{Node}\ \mathit{Int}\ \ ) \\
 & \textbf{type instance}\ \mathit{Edge}\ \mathit{String} = \\
 & \quad (\mathit{Node}\ \mathit{Char}, \mathit{Node}\ \mathit{Char})
\end{array}$$

Figure D.2: Type Constructors vs Family Constructors

Indexed type families come in two flavors: *data families* and *type synonym families* (the above example). They are the indexed family variants of Algebraic Data Type (ADT) and type synonyms, respectively. The instances of data families can be data types and *newtypes*.

D.4 NewType - Datatype Renamings

Simon P. Jones, editor - Haskell 98 Language and Libraries: the Revised Report [34]

A declaration of the form:

$$\textbf{newtype}\ cx \Rightarrow T\, U_1\, ...\, U_k = N\, t$$

introduces a new type whose representation is the same as an existing type. The type $(T\, U_1\, ...\, U_k)$ renames the data type **t**. It differs from a type synonym in that it creates a distinct type that must be explicitly coerced to or from the original type. Also, unlike type synonyms, *newtype* may be used to define recursive types. The constructor **N** in an expression coerces a value from type **t** to type $(T\, U_1 ... U_k)$. Using **N** in a pattern coerces a value from type $(T\, U_1 ... U_k)$ to type **t**. These coercions may be implemented without execution time overhead; *newtype* does not change the underlying representation of an object. A *newtype* declaration may use field-naming syntax, though of course there may only be one field. Thus:

$$\textbf{newtype}\ \mathit{Age} = \mathit{Age}\ \{\mathit{unAge} :: \mathit{Int}\}$$

brings into scope both a constructor and a de-constructor:

$$\begin{array}{l}
\mathit{Age} \quad\ :: \mathit{Int} \rightarrow \mathit{Age} \\
\mathit{unAge} :: \mathit{Age} \rightarrow \mathit{Int}
\end{array}$$

D.5 Extension: Template Haskell

Template Haskell [37] provides the ability for a Haskell program to perform computations at compile time which generate new code that can then be *spliced* into the program. Template Haskell defines a standard algebraic data type for representing the abstract syntax of Haskell programs, and a set of monadic operations for constructing programs. These are expressible in pure Haskell. Several syntactic constructs are introduced:

- $[\![...]\!]$ A quotation construct that gives the AST representation of the enclosed expression.
- $[d|...]\!]$ A quotation construct that gives the AST representation of the enclosed declaration.
- $[t|...]\!]$ A quotation construct that gives the AST representation of the enclosed type.
- $(...) A splicing construct that takes a code representation tree and effectively inserts it into a program.

So by 'quoting' a piece of Haskell code a programmer can get access to the AST that represents the 'quoted' code. In Template Haskell, all aspects of Haskell which the ordinary programmer can use are also available to process the AST at program generation time. Thus a function that works on these ASTs is just an ordinary Haskell function definition. A programmer can also construct an AST (if he does not wish to use the quotation mechanism) using monadic operations provided by the Template Haskell library. As said, the AST of the Haskell code can be *spliced* back into a program at compile time. So *splicing* is the act of inserting a generated AST in the AST of the original program.

To get a better feeling for the functionality that Template Haskell offers we will examine an example in which we both the *quotation* and *splicing* syntax are used. The function we will examine, *sel*, can select any element from a tuple of arbitrary size:

```
GHCi> $(sel 3 4) ('a', 'b', 'c', 'd')
'c'
GHCi> $(sel 2 3) ('a', 'b', 'c')
'b'
```

So what happens in the first example is that we insert the AST of the *sel* function that is parameterized to take the third element out of tuple of size four. In the second example should then speak for itself. Below we can see the code that is used to construct the AST for the *sel* function.

```
sel :: Int → Int → ExpQ
sel i n = ⟦λx → $(caseE ⟦ x ⟧[alt])⟧
  where
    alt :: MatchQ
    alt = match pat (normalB rhs) [ ]
    pat :: Pat
    pat = tupP (map varP as)
    rhs :: ExpQ
    rhs = varE (as !! (i − 1))
    as :: [String]
    as = ["a" ++ show i | i ← [1 . . n]]
```

The functions *caseE*, *varE* function, etc. are the monadic constructor functions mentioned earlier. The exact working of the above code is beyond the scope of this introductory appendix. However, to get an idea of how the the above Template Haskell function works, the spliced code for $(sel 3 4) is shown below:

```
λx → case x of (a1, a2, a3, a4) → a3
```

Information on Template Haskell is spread over multiple webpages and (un)published papers, as such, a reader in search of more details on Template Haskell can not be referred to a single source. Most of the information on Template Haskell is however aggregated on the Haskell.org wiki-page dedicated to Template Haskell [15].

Appendix E

CλasH Generated VHDL

This appendix is included to give a reader a feel for the kind of VHDL that is generated from a CλasH design. The specific design translated is a 4-tap Finite Impulse Response (FIR) filter, and it is assumed that the reader is familiar with such filters.

E.1 CλasH Design

The CλasH description of the 4-tap FIR filter found in Code Snippet E.1 starts with two type aliases, which are defined for convenience. The type aliases are followed by the definition of the dot-product operator, named *+*, which is parametric in both the length of the two input vectors, as the size of the integers within these vectors. Next follows the *monomorphic*, *first-order*, 4-tap FIR filter. It uses the *shiftr* function to implement a shift register. The first annotation indicates that it is the top entity, the second tells that *initfir* holds the initial value of the state of the FIR filter.

Code Snippet E.1 *(4-tap FIR-filter in CλasH).*

```
type Int8 = SizedInt D8
type Vec4 = Vector D4 Int8
xs *+* ys = foldl (+) 0 (zipWith (*) xs ys)
{-# ANN fir TopEntity #-}
{-# ANN fir (InitState 'initfir) #-}
fir :: State (Vec4, Vec4) → Int8 → (State (Vec4, Vec4), Int8)
fir (State (hs, us)) x = (State (hs, shiftr us x), us *+* hs)
initfir :: (Vec4, Vec4)
initfir = ($(vectorTH [2 :: Int8, 3, −2, 4]), copy 0)
```

E.2 Generated VHDL

Translation starts with the creation of a *types* package (Figure E.1), which defines all the translated types (vector, tuples, custom datatypes, etc.) and certain built-in functions. The built-in functions are generated according to need, except for the *show* function, which is generated alongside *every* generated type. The unused *show* functions do not generate extra overhead in the hardware.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

package types is
  subtype tfvec_index is integer range −1 to integer'high;

  subtype \signed_7\ is signed (0 to 7);

  type \vector_signed_7\ is array (tfvec_index range <>) of
   \signed_7\;

  subtype \vector—signed_7—0_to_4\ is \vector_signed_7\ (0
  to 3);

  type \(,)vector—signed_7—0_to_4vector—signed_7—0_to_4\ is
    record A : \vector—signed_7—0_to_4\;
           B : \vector—signed_7—0_to_4\;
    end record;

  subtype \vector—signed_7—0_to_1\ is \vector_signed_7\ (0
  to 0);

  subtype \vector—signed_7—0_to_2\ is \vector_signed_7\ (0
  to 1);

  subtype \vector—signed_7—0_to_3\ is \vector_signed_7\ (0
  to 2);

  function \show\ (s : std_logic)
               return string;

  function \show\ (b : boolean)
               return string;

  function \show\ (sint : signed)
               return string;

  function \show\ (uint : unsigned)
               return string;

  function \show\ (tup : \(,)vector—signed_7—0_to_4vector—
  signed_7—0_to_4\)
               return string;

  function \+>\ (a : \signed_7\;
               vec : \vector_signed_7\)
             return \vector_signed_7\;

  function \head\ (vec : \vector_signed_7\)
               return \signed_7\;

  function \shiftr\ (vec : \vector_signed_7\;
                 a : \signed_7\)
               return \vector_signed_7\;

  function \show\ (vec : \vector_signed_7\)
               return string;

  function \singleton\ (a : \signed_7\)
                   return \vector_signed_7\;

  function \tail\ (vec : \vector_signed_7\)
               return \vector_signed_7\;
end package types;

package body types is
  function \show\ (s : std_logic) return string is
  begin
    if s = '1' then
      return "High";
    else
      return "Low";
    end if;
  end;

  function \show\ (sint : signed) return string is
  begin
    return integer'image(to_integer(sint));
  end;
```

```
  function \show\ (uint : unsigned) return string is
  begin
    return integer'image(to_integer(uint));
  end;

  function \show\ (b : boolean) return string is
  begin
    if b then
      return "True";
    else
      return "False";
    end if;
  end;

  function \show\ (tup : \(,)vector—signed_7—0_to_4vector—
  signed_7—0_to_4\)
               return string is
  begin
    return '(' & (\show\(tup.A) & ',' & \show\(tup.B)) & ')
    ';
  end;

  function \+>\ (a : \signed_7\;
               vec : \vector_signed_7\)
               return \vector_signed_7\ is
    variable res : \vector_signed_7\ (0 to vec'length);
  begin
    res := a & vec; return res;
  end;

  function \head\ (vec : \vector_signed_7\)
               return \signed_7\ is
  begin
    return vec(0);
  end;

  function \shiftr\ (vec : \vector_signed_7\;
                 a : \signed_7\)
               return \vector_signed_7\ is
    variable res : \vector_signed_7\ (0 to vec'length − 1);
  begin
    res := \tail\(vec) & a;
    return res;
  end;

  function \show\ (vec : \vector_signed_7\)
               return string is
    function \doshow\ (vec : \vector_signed_7\)
                   return string is
    begin
      case vec'length is
        when 0 =>
          return "";
        when 1 =>
          return \show\(\head\(vec));
        when others =>
          return \show\(\head\(vec)) & ',' &
                 \doshow\(\tail\(vec));
      end case;
    end;
  begin
    return '<' & \doshow\(vec) & '>';
  end;

  function \singleton\ (a : \signed_7\)
                   return \vector_signed_7\ is
    variable res : \vector_signed_7\ (0 to 0) := (others =>
     a);
  begin
    return res;
  end;

  function \tail\ (vec : \vector_signed_7\)
               return \vector_signed_7\ is
    variable res : \vector_signed_7\ (0 to vec'length − 2);
  begin
    res := vec(1 to vec'length − 1);
    return res;
  end;
end package body types;
```

Figure E.1: Types package

In Figure E.2 we see the top entity: the 4-taps FIR filter. A reader will immediately notice that the automated name generation / derivation does not currently emphasize on readability, instead, the uniqueness of the names is its primary concern. Perhaps a future version of the CλasH compiler will improve on the readability of the generated names. The important parts of the VHDL architecture are the component instantiation of the dot-product component (*Component_1*), the call to the *shiftr* function (which is defined in the *types* package), and the *state* process. Also note that the reset value of the state is contained in a component (*initfirComponent_2*).

```
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity firComponent_0 is
     port (\xzedo4\ : in \signed_7\;
           \foozeegzeeg4\ : out \signed_7\;
           clock : in std_logic;
           resetn : in std_logic);
end entity firComponent_0;

architecture structural of firComponent_0 is
  signal \dszedq4\ : \(,)vector-signed_7-0_to_4vector-signed_7-0_to_4\;
  signal \hszedu4\ : \vector-signed_7-0_to_4\;
  signal \uszedw4\ : \vector-signed_7-0_to_4\;
  signal \argzee5zee53\ : \signed_7\;
  signal \reszedGzedK3\ : \vector-signed_7-0_to_4\;
  signal \foozeeczeec4\ : \(,)vector-signed_7-0_to_4vector-signed_7-0_to_4\;
begin
  \hszedu4\ <= \dszedq4\.A;
  \uszedw4\ <= \dszedq4\.B;

  \comp_ins_argzee5zee53\ : entity Component_1
    port map (\paramzefczefc4\ => \uszedw4\,
              \paramzefezefe4\ => \hszedu4\,
              \reszefgzefg4\ => \argzee5zee53\,
              clock => clock,
              resetn => resetn);

  \reszedGzedK3\ <= \shiftr\(\uszedw4\, \xzedo4\);

  \foozeeczeec4\.A <= \hszedu4\;
  \foozeeczeec4\.B <= \reszedGzedK3\;

  \foozeegzeeg4\ <= \argzee5zee53\;

  state : block
      signal \initfirval\ : \(,)vector-signed_7-0_to_4vector-signed_7-0_to_4\;
  begin
    \resetval_initfirrcSJ2\ : entity initfirComponent_2
       port map (\foozegpzegp4\ => \initfirval\,
                 clock => clock,
                 resetn => resetn);
    stateupdate : process (clock, resetn, \initfirval\)
    begin
      if resetn = '0' then
        \dszedq4\ <= \initfirval\;
      elsif rising_edge(clock) then
        \dszedq4\ <= \foozeeczeec4\;
      end if;
    end process stateupdate;
  end block state;
end architecture structural;
```

Figure E.2: 4-tap FIR Filter

Figure E.3 shows us the VHDL code for the dot-product component (*Component_1*). The first thing a reader will most likely notice is that the only thing *Component_1* does is instantiate *Component_3*, which is the component that holds the actual logic. This is most likely caused by the *de-sugaring* process of the compiler (described in more detail by Kooijman [26]). Though it is a minor inconvenience, it causes no additional overhead in the eventual hardware, as such, trying to remove these 'wrappers' during VHDL generation is of low priority.

One will also notice that both components have a clock and a reset port, even though both are purely combinatorial entities. This is because every entity, whether they are stateless or stateful, has a clock and reset port. As the unused ports will not generated any overhead in the eventual hardware, removing them in a future version of the CλasH compiler is considered a low priority task. Examining the dot-product component we see how the templates for the *zipWith* function and the *foldl* function shown in appendix A are instantiated for the specific vector lengths.

```
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity Component_1 is
  port (\paramzefczefc4\ : in \vector—signed_7—0_to_4\;
        \paramzefezefe4\ : in \vector—signed_7—0_to_4\;
        \reszefgzefg4\ : out \signed_7\;
        clock : in std_logic;
        resetn : in std_logic);
end entity Component_1;

architecture structural of Component_1 is
begin
\comp_ins_reszefgzefg4\ : entity Component_3
  port map (\paramzegJzegJ3\ => \paramzefczefc4\,
            \paramzegLzegL3\ => \paramzefezefe4\,
            \foozehfzehf4\ => \reszefgzefg4\,
            clock => clock,
            resetn => resetn);
end architecture structural;
```

```
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity Component_3 is
  port (\paramzegJzegJ3\ : in \vector—signed_7—0_to_4\;
        \paramzegLzegL3\ : in \vector—signed_7—0_to_4\;
        \foozehfzehf4\ : out \signed_7\;
        clock : in std_logic;
        resetn : in std_logic);
end entity Component_3;
```

```
architecture structural of Component_3 is
  signal \reszeh9zeh93\ : \signed_7\;
  signal \argzeh3zeh33\ : \vector—signed_7—0_to_4\;
begin

  \reszeh9zeh93\ <= to_signed(0, 8);

  \zipWithVectorargzeh3\ : for n in 0 to 3 generate
  begin
    \comp_ins_argzeh3zeh33(n)\ : entity funzehbComponent_3
        port map (\paramzehtzeht4\ => \paramzegJzegJ3\(n),
                  \paramzehvzehv4\ => \paramzegLzegL3\(n),
                  \reszeh6zehr4\ => \argzeh3zeh33\(n),
                  clock => clock,
                  resetn => resetn);
  end generate \zipWithVectorargzeh3\;

  \foldlVectorfoozehf\ : block
    signal tmp : \vector—signed_7—0_to_4\;
  begin
    \foldlVectorargzeh3\ : for n in 0 to 3 generate
    begin
      \firstcell\ : if n = 0 generate
      begin
        \comp_ins_tmp(n)\ : entity funzehdComponent_4
            port map (\paramzehFzehF3\ => \reszeh9zeh93\,
                      \paramzehHzehH3\ => \argzeh3zeh33\(n),
                      \reszeh8zehD3\ => tmp(n),
                      clock => clock,
                      resetn => resetn);
      end generate \firstcell\;
      \othercell\ : if n /= 0 generate
      begin
        \comp_ins_tmp(n)\ : entity funzehdComponent_4
            port map (\paramzehFzehF3\ => tmp(n — 1),
                      \paramzehHzehH3\ => \argzeh3zeh33\(n),
                      \reszeh8zehD3\ => tmp(n),
                      clock => clock,
                      resetn => resetn);
      end generate \othercell\;
    end generate \foldlVectorargzeh3\;
    \foozehfzehf4\ <= tmp(3);
  end block \foldlVectorfoozehf\;
end architecture structural;
```

Figure E.3: Dot-product

Again, as a result of the *de-sugaring* process, the multiplication and addition operator are translated to separate components, instead of being inlined in the dot-product component. These components are listed in Figure E.4, where the *funzehbComponent_3* component and the *funzehdComponent_4* component are the multiplication and addition operator respectively. The reader will notice that the result of the multiplication has to be resized to the same number of bits as the number of bits in the input values. This is done so that the resulting hardware corresponds with the CλasH design, where the result of a multiplication also has the same number of bits as the number of bits in the input values.

```
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity funzehbComponent_3 is
  port (\paramzehtzeht4\ : in \signed_7\;
        \paramzehvzehv4\ : in \signed_7\;
        \reszeh6zehr4\ : out \signed_7\;
        clock : in std_logic;
        resetn : in std_logic);
end entity funzehbComponent_3;

architecture structural of funzehbComponent_3 is
begin
  \reszeh6zehr4\ <=
     resize(\paramzehtzeht4\ * \paramzehvzehv4\, 8);
end architecture structural;
```

```
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;

entity funzehdComponent_4 is
  port (\paramzehFzehF3\ : in \signed_7\;
        \paramzehHzehH3\ : in \signed_7\;
        \reszeh8zehD3\ : out \signed_7\;
        clock : in std_logic;
        resetn : in std_logic);
end entity funzehdComponent_4;

architecture structural of funzehdComponent_4 is
begin
  \reszeh8zehD3\ <=
     \paramzehFzehF3\ + \paramzehHzehH3\;
end architecture structural;
```

Figure E.4: Multiplication & Addition

Finally, in Figure E.5, we see the component that holds the initial values for the registers of the FIR filter (*initfirComponent_2*). We see how the *vectorTH* function is translated to a series of value-to-vector concatenation functions (+>) and a call to the *singleton* function.

```
use work.types.all;
use work.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use std.textio.all;


entity initfirComponent_2 is
  port (\foozegpzegp4\ : out \(,)vector—signed_7—0
  _to_4vector—signed_7—0_to_4\;
        clock : in std_logic;
        resetn : in std_logic);
end entity initfirComponent_2;

architecture structural of initfirComponent_2 is
  signal \reszefAzefH3\ : \signed_7\;
  signal \foozegdzegd4\ : \vector—signed_7—0_to_4\;
  signal \argzefPzefP3\ : \signed_7\;
  signal \foozegfzegf4\ : \vector—signed_7—0_to_1\;
  signal \argzefNzefN3\ : \signed_7\;
  signal \foozeghzegh4\ : \vector—signed_7—0_to_2\;
```

```
  signal \argzefLzefL3\ : \signed_7\;
  signal \foozegjzegj4\ : \vector—signed_7—0_to_3\;
  signal \argzefJzefJ3\ : \signed_7\;
  signal \foozeglzegl4\ : \vector—signed_7—0_to_4\;
  signal \foozegnzegn4\ : \vector—signed_7—0_to_4\;
begin
  \reszefAzefH3\ <= to_signed(0, 8);
  \foozegdzegd4\ <= (others => \reszefAzefH3\);

  \argzefPzefP3\ <= to_signed(4, 8);
  \foozegfzegf4\ <= \singleton\(\argzefPzefP3\);
  \argzefNzefN3\ <= to_signed(254, 8);
  \foozeghzegh4\ <= \+>\(\argzefNzefN3\, \foozegfzegf4\);
  \argzefLzefL3\ <= to_signed(3, 8);
  \foozegjzegj4\ <= \+>\(\argzefLzefL3\, \foozeghzegh4\);
  \argzefJzefJ3\ <= to_signed(2, 8);
  \foozeglzegl4\ <= \+>\(\argzefJzefJ3\, \foozegjzegj4\);
  \foozegnzegn4\ <= \foozeglzegl4\;

  \foozegpzegp4\.A <= \foozegnzegn4\;
  \foozegpzegp4\.B <= \foozegdzegd4\;
end architecture structural;
```

Figure E.5: Initial register values

Bibliography

[1] Alfonso Acosta, Oleg Kiselyov, and Wolfgang Jeltsch. Fixed sized vectors. Vectors with numerically parameterized size. February 2008. Available from: http://code.haskell.org/parameterized-data/. [CITED: 18-03-2009].

[2] Alfonso Acosta, Oleg Kiselyov, and Wolfgang Jeltsch. type-level: Type-level programming library. 2008. Available from: http://code.haskell.org/type-level/. [CITED: 19-03-2009].

[3] J. L. Armstrong and R. Virding. Erlang — An Experimental Telephony Switching Language. In *XIII International Switching Symposium*, Stockholm, Sweden, May 27 – June 1, 1991.

[4] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. *Formal Methods and Models for Co-Design, ACM/IEEE International Conference on*, page 249, 2003.

[5] Christiaan Baaij. CλasH: Functional Hardware Description Languages. Individual assignment, Twente University, March 2009.

[6] John Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[7] H. P. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 117–309, New York, NY, USA, 1992. Oxford University Press, Inc. ISBN 0-19-853761-1.

[8] Koen Claessen. *An Embedded Language Approach to Hardware Description and Verification*. PhD thesis, Chalmers University, Götenberg, Sweden, September 2000.

[9] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Asian Computing Science Conference*, pages 62–73. Springer Verlag, 1999.

[10] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. Albuquerque, N.M., U.S.A., January 1982.

[11] D.D. Gajski and R.H. Kuhn. Guest Editor's Introduction: New VLSI Tools. *IEEE Computer*, 1983.

[12] Peter Gavin. Type Family Programming (TFP). 2008. Available from: http://code.haskell.org/~pgavin/tfp. [CITED: 25-03-2009].

[13] Marco Gerards. Streaming Reduction Circuit for Sparse Matrix Vector Multiplication in FPGAs. Master's thesis, University of Twente, Enschede, The Netherlands, August 2008.

[14] Thomas Hallgren. Fun with Functional Dependencies. In *Proc. of the Joint CS/CE Winter Meeting*, 2000.

[15] Haskell.org. Template Haskell. 2009. Available from: http://www.haskell.org/haskellwiki/Template_Haskell. [CITED: 02-11-2009].

[16] Haskell.org. GHC/Type Families. 2009. Available from: http://www.haskell.org/haskellwiki/GHC/Type_families. [CITED: 04-05-2009].

[17] J.R. Hindley. The principal type scheme of an object in combinatory logic. In *Transactions of the American Mathematical Society*, volume 146, pages 29–60, December 1969.

[18] Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[19] IEEE. 1364-2005 IEEE Standard for Verilog Hardware Description Languages, 2005.

[20] IEEE. 1076-2008 IEEE Standard VHDL Language Reference Manual, 2008.

[21] Steven D. Johnson. *Synthesis of Digital Design from Recursive Equations.* MIT Press, Cambridge, MA, USA, 1984.

[22] G. Jones and M. Sheeran. Circuit Design in Ruby. In *Formal Methods for VLSI Design.* Elsevier Science Publishers, 1990.

[23] Mark P. Jones. *Programming Languages and Systems*, chapter Type Classes with Functional Dependencies, pages 230–244. Sprinker Berline / Heidelberg, 2000.

[24] M.P. Jones. *Qualified Types: Theory and Practice.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992.

[25] Simon Peyton Jones. Bulk types with class. In *Proceedings of the Second Haskell Workshop*, 1996.

[26] Matthijs Kooijman. Haskell as a higher order structural hardware description language. Master's thesis, Twente University, 2009.

[27] Zhaohui Luo. *Computation and reasoning: a type theory for computer science.* Oxford University Press, Inc., New York, NY, USA, 1994. ISBN 0-19-853835-9.

[28] Ian Lynagh. Unrolling and Simplifying Expressions with Template Haskell. May 2003. Available from: http://www.haskell.org/th/papers/Unrolling_and_Simplifying_Expressions_with_Template_Haskell.ps. [CITED: 13-10-2009].

[29] Conor McBride. Faking it: Simulating dependent types in Haskell. *Journal on Functional Programming*, 12(5):375–392, 2002. ISSN 0956-7968.

[30] George H. Mealy. A Method to Synthesizing Sequential Circuits. *Bell Systems Technical Journal*, pages 1045–1079, 1955.

[31] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[32] Ulf Norell. *Advanced Functional Programming*, chapter Dependently Typed Programming in Agda, pages 230–266. Springer Berlin / Heidelberg, 2009.

[33] John T. O'Donnell. *LNCS 3016*, chapter Embedding a Hardware Description Language in Template Haskell, pages 143–164. Springer Verlag, 2004.

[34] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: the Revised Report.* 2003. [CITED: 03-04-2009].

[35] Ingo Sander and Axel Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004.

[36] Tom Schrijvers, Louis-Julien Gauillemette, and Stefan Monnier. Type Invariants for Haskell. In *Programming Languages meets Program Verification (PLPV)*, 2009.

[37] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *ACM SIGPLAN Haskell Workshop 02*, 2002.

[38] Mary Sheeran. *μFP, an algebraic VLSI design language*. PhD thesis, Programming Research Group, Oxford University, 1983.

[39] Satnam Singh. The Lava Hardware Description Language. 2009. Available from: http://www.raintown.org/lava/. [CITED: 12-03-2009].

[40] Guy L. Steele, Jr., and Richard P. Gabriel. The Evolution of Lisp. In *ACM SIGPLAN Notices*, pages 231–270, 1993.

[41] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 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, 2007. ACM. ISBN 1-59593-393-X.