Recursive Functional Hardware Descriptions using CλaSH

Ingmar te Raa

UNIVERSITY OF TWENTE.

University of Twente

Master Thesis

Recursive Functional Hardware Descriptions using CλaSH

Author:
Ingmar te Raa

Supervisors:
Dr.Ir. J. Kuper
Dr.Ir. J.F. Broenink
Dr.Ir. C.P.R. Baaij
Dr.Ir. R. Wester

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS),
Computer Architecture for Embedded Systems (CAES) group

November 20, 2015

Abstract

CλaSH is a functional hardware description language in which structural descriptions of combinational and synchronous sequential hardware can be expressed. The language is based on Haskell, from which it inherits abstraction mechanisms such as, the support of polymorphism and higher-order functions. Recursion is another fundamental and commonly used abstraction mechanism in Haskell. In contrast with Haskell, the support of recursion in CλaSH is currently limited. This is considered a shortcoming by many CλaSH users.

Data-dependent recursive functions pose a problem for the current implementation of CλaSH. Currently, these recursive function definitions are unrolled by the compiler, in an attempt to produce finite circuits. In the case of data-dependent recursive functions, such finite circuit descriptions often cannot be found using unrolling, as it would require infeasibly large circuits, capable of handling all possible arguments.

This thesis focuses on extending the CλaSH compiler with support of data-dependent recursion. This is established by describing a formal rewrite method, based on the continuation passing style transformation. This method transforms recursive function descriptions to a corresponding circuitry, capable of executing the recursive function. A detailed description of the generated stack architecture is provided in the form of CλaSH descriptions. The resulting circuits, produced by applying the methodology, are elaborated and synthesis results of those circuitries are discussed.

Dankwoord

Allereerst wil ik de commissie bedanken voor het ondersteunen van mijn afstuderen. Jan, bedankt voor de uitvoerige gesprekken, waarbij ik volledig de aandacht kreeg. Graag bedank ik ook Christiaan voor zowel de waardevolle en uitvoerige feedback als inspiratie met betrekking tot hardware ontwerp. Rinse, bedankt voor de goede begeleiding en gezelligheid op het kantoor.

Daarnaast wil ik de leden van de vakgroep CAES bedanken voor hun inspiratie en gezelligheid. Ik heb een goede indruk kunnen krijgen van zowel het leven van een promovendus, als de wetenschappelijke onderwijsvoering vanuit deze vakgroep. De gezelligheid tijdens de koffiepauzes en borrels die gepaard gingen met zowel goede discussies als droge humor: dit zorgde voor een werksfeer die nooit ging vervelen. In het bijzonder wil ik Arjan bedanken voor de hulp en inzichten die hij heeft gegeven met betrekking tot mijn afstuderen. En natuurlijk mijn andere kantoorgenoot Guus. Samen met Rinse hebben wij heel wat lol gehad: essentieel voor de sfeer binnen het kantoor.

Ten slotte wil ik mijn vriendin Janneke bedanken, voor het onvoorwaardelijk steunen tijdens deze drukke periode.

Ingmar,
Enschede, november 2015

Contents

List of Figures

List of Tables

1 Introduction

Computing devices are used to accomplish an ever increasing number of tasks. In the digital age we currently live in, these computer devices not only are omnipresent, the tasks they perform also evolve rapidly. The hardware that is used to accomplish these tasks, grows alongside with this trend. Due to innovations in fabrication techniques of transistors, used in for example *Central Processing Units (CPUs)*, *Graphics Processing Units (GPUs)*, and *Field Programmable Gate Arrays (FPGAs)*, a larger number of these transistors can be packed into such chips.

To illustrate the trend that currently takes place in the evolution of computing devices, the number of transistors used in CPUs, GPUs, and FPGAs are shown in Figure 1.1. A period of 50 years show a rapid increase in transistor count. The largest transistor count displayed in Figure 1.1 contains more than twenty-billion transistors; an FPGA produced in 2014 by Xilinx [30]. To put that number in context, this is about 2.8 times the world population in 2014.

Figure 1.2 illustrates the wide variety of applications in which FPGAs are currently used. These applications vary from low demanding consumer applications to high demanding aerospace applications. In the early 90s the application domains of FPGAs were mainly networking and telecommunication technologies. This indicates that: not only the capabilities of the FPGAs grow, but they are also deployed in a wider variety of application domains.

The *Hardware Description Languages (HDLs)*, that are used to implement digital circuits on FPGAs, are subject to both of these trends: digital circuits described by these languages become larger as resources on FPGAs increase, and the HDLs used to implement digital circuits are used in more and more domains. This requires the HDLs to be both scalable and flexible.

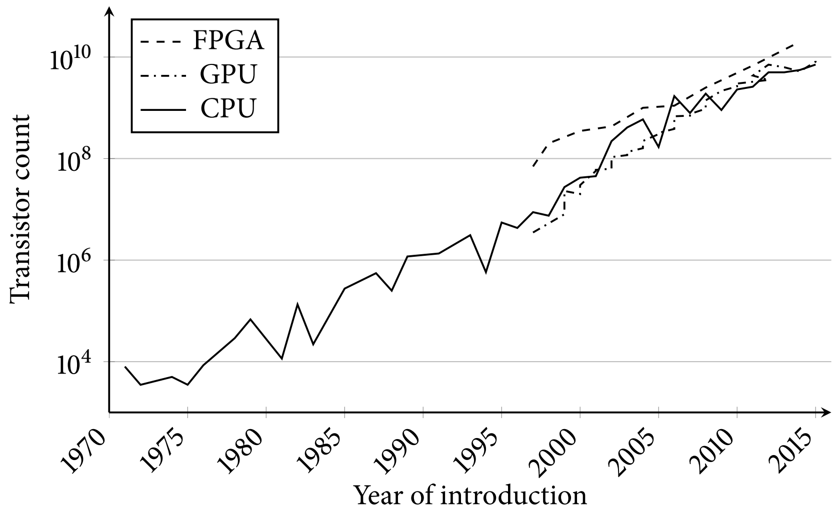

FIGURE 1.1 – Number of transistors in CPUs, GPUs, and FPGAs [4].

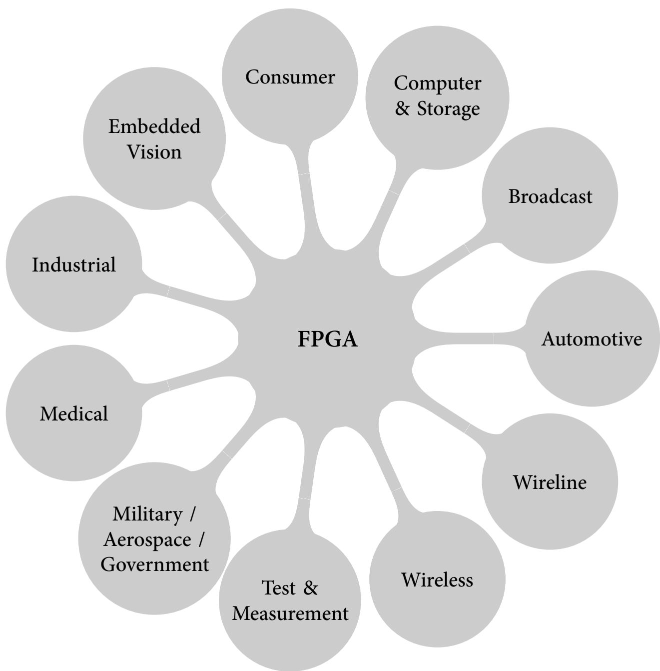

FIGURE 1.2 – Applications domains of FPGAs in the industry, advertised by Altera Corperation [3].

Currently the most common HDLs that are used in the industry are *VHSIC Hardware Description Language (VHDL)* [1] and *Verilog* [2]. These languages have proven their power in the industry. It is however important to keep improving these languages, and exploring alternative languages compared to the existing ones.

In this thesis, the focus will lie on such an alternative: CλaSH [7]. CλaSH is a Functional HDL (FHDL) based on the semantics of the Haskell language in which structural descriptions of combinational and synchronous sequential hardware can be expressed. The language supports polymorphism and higher-order functions, properties inherited from the Haskell language.

1.1 Problem statement and approach

The ability to express recursive function definitions is fundamental in the Haskell language, and commonly used by developers using this language. In the CλaSH language, however, the ability to express recursive function definitions is limited. This is considered as a shortcoming by many CλaSH users [20, 26, 36, 37].

Research is conducted in this thesis to extend the support of recursion. As will be elaborated in this thesis, the ability to express recursion present in so-called *data-dependent* recursive functions, is currently unsupported in CλaSH. The research question central to this thesis will therefore be:

» *How can data-dependent recursive function definitions be supported by the CλaSH compiler?*

Several aspects related to this question need to be clarified, before this research question can be addressed. For instance, the exact limitations of CλaSH need to be identified. Furthermore, a type of hardware architecture need to be identified, capable of handling the recursive algorithms described in the CλaSH language. These structures must be derived automatically in order to be part of the CλaSH compiler.

1.2 Outline of this thesis

This thesis is structured as follows. In Chapter 2, background and related work are discussed. This gives the reader the required background knowledge to read the rest of this thesis and it provides the current status of related work. In Chapter 3, a methodology is developed to generate hardware from recursive equations. An guiding example is used in this chapter to illustrate the presented methodology. The presented methodology is implemented by means of a proof of concept, which is presented in Chapter 4. This chapter contains implementation details of the generated hardware. Experimental results are evaluated in Chapter 5. It contains both results of applying the methodology presented in Chapter 3 to translate recursive descriptions, and synthesis results of the generated hardware structures. Finally, in Chapter 6, the work presented thesis is discussed and conclusions are drawn.

2

Background and Related Work

In this chapter, relevant background information and related work is elaborated. A basic understanding of the relevant topics discussed in this thesis, is established. Furthermore, relevant work is elaborated in form of a discussion. This provides the required knowledge which is needed to read the rest of this thesis.

Because CλaSH is central to this thesis, both the language and the compiler are elaborated. The reader should be able to understand how hardware is developed using the CλaSH language and the CλaSH compiler. The inner workings of the CλaSH compiler pipeline are also roughly discussed, without going into to much detail. Additionally, the current status of the support of recursion in CλaSH is elaborated.

Several different properties of recursive functions are distinguished in this research, which are also elaborated within this chapter. The properties of these recursive functions are explained with the use of examples of such functions. Throughout the rest of this thesis, these properties are used to identify specific forms of recursion, for which these properties hold. Furthermore, the provided examples are used throughout this thesis to show the effects of the implementation of such recursive forms in reconfigurable hardware.

Relevant literature in the field of reconfigurable hardware and HDLs is covered. This provides the reader with the knowledge and the status of the research already conducted in these fields. Initially the focus will be on the broad field of recursion in reconfigurable hardware. Later on in this chapter, the scope will be narrowed down to a particular kind of HDLs, which is more relevant to this thesis: FHDL compilers. Within this relevant work, a specific concept is used, called Continuation Passing Style (CPS). This concept is explained in further detail in this chapter, as it is used in the rest of this thesis.

An overview of the CλaSH language and the CλaSH compiler, is provided in section §2.1. Background on recursion is elaborated in section §2.2: to enable the reader to distinguishes between various kinds of recursion. Then, in section §2.3, related work is evaluated. In this evaluation it will become clear that particular work is especially relevant to this thesis. Therefore a specific concept, used in the rest of this thesis, is further elaborated in section §2.4 to provide the necessary background to understand the rest of this thesis.

2.1 CλaSH

CAES language for asynchronous hardware (CλaSH) [6, 7, 14] is a FHDL which borrows syntax and semantics from Haskell. The language allows a circuit designer to describe hardware using advanced Haskell language constructs like polymorphism and higher-order functions. Netlist of the circuits designed in CλaSH are produced by the compiler in commonly used HDLs like VHDL and Verilog. A circuit designer can use commonly available synthesis tooling, like Altera Quartus or Xilinx Vivado, to further synthesize the VHDL (or Verilog) produced by CλaSH, to a digital circuit. The CλaSH compiler also includes an interactive environment allowing a hardware developer to simulate the circuits developed in CλaSH, without the need of specifying a seperate test bench.

Throughout the past several years, CλaSH is used to describe circuits for applications in varying domains. This includes, domain specific processors: a Data-flow processor [27] and a Very Long Instruction Word (VLIW) processor [10]; the domain of computer algorithms: the n-queens algorithm [22] and the MUltiple SIgnal Classification (MUSIC) algorithm [21]; the domain of state space estimation using a particle filter [38], the domain of astronomy poly-phase filter bank [39], and an application in the domain of biology by means of an auditory model of a cochlea [11].

2.1.1 Hardware Design using CλaSH

In CλaSH, functions are used to describe hardware. A basic set of functions is provided in the CλaSH prelude library. This enables a circuit designer to design both combinational and synchronous sequential hardware. Types are used in CλaSH, to specify what kind of hardware needs to be compiled. One can for example use an `Unsigned` 32 type to specify wires that can handle a 32 bit unsigned integer.

A special type, called a `Signal`, is used when a sequential synchronized circuits is described in CλaSH. A `Signal` can be seen as an *infinite* list of samples, where each sample corresponds to a value at a specific moment in time. These moments are synchronized by a clock. Registers are used to capture the values of the samples. In other words, the state of the `Signal` is captured via registers. Combinational circuits are described, without the use of the `Signal` data-type.

The CλaSH prelude library contains a classic machine model: the Mealy machine. Figure 2.1 shows this Mealy machine. Both the input i and state s are input for the function f. The function f is the combinational function used to determine the output o and the next state s'. All the inputs and the output of f are of type `Signal`. The next state s' is captured in a register.

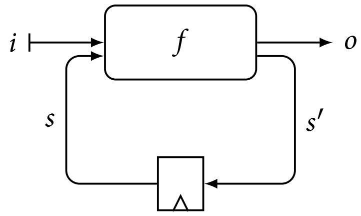

FIGURE 2.1 – Generic form of a Mealy machine as can be described by CλaSH

CλaSH hardware description example

To illustrate how CλaSH can be used to design circuits, an example is worked out in Listing 2.1 and Figure 2.2. Listing 2.1 shows a Mealy description of a Multiply ACcumulate (MAC) operation. The input of the Mealy description is a tuple (x, y) which contains the values that need to be multiplied. The state s of the Mealy machine consist of an accumulator. The output of the MAC function is equal to the next state s'.

Mathematically, one could express the result of the mac operation as: $s' = s + x \cdot y$. Note the similarities between the mathematical description and the hardware description in CλaSH. Figure 2.2 contains the resulting circuit corresponding to Listing 2.1.

```
mac s (x,y) = (s',o)
  where
    s' = s + x * y
    o = s'

mac' = mealy mac 0
```

LISTING 2.1 – MAC hardware description defined in the CλaSH language.

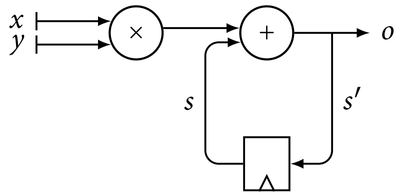

FIGURE 2.2 – MAC circuit corresponding to the CλaSH description in Listing 2.1.

2.1.2 Compiler pipeline

The CλaSH compiler produce a netlist in the form of other, more low-level, HDLs. This may be for example VHDL. Three subsequent steps are used to derive these netlists. These steps are depicted in Figure 2.3.

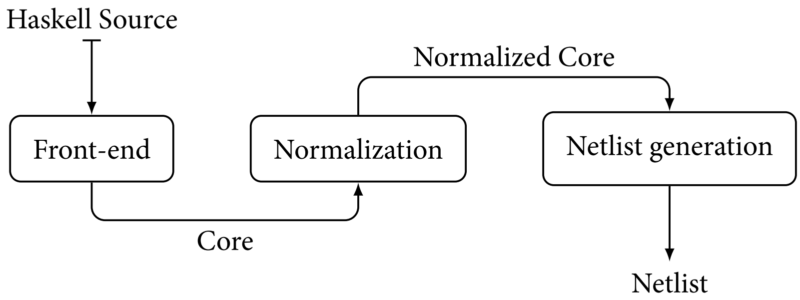

FIGURE 2.3 – CλaSH compiler pipeline

Front-end The CλaSH source code is presented to the front-end. This front-end processes the CλaSH source to an *Intermediate Representation (IR)* named Core. CλaSH uses the Glasgow Haskell Compiler (GHC) [35] for this step, which is an open source Haskell compiler. This Core IR is passed to the following step.

Normalization The *Core* produced by the front-end is fed to the normalization step. This normalization step produces *Normalised Core*. In essence, the CλaSH compiler uses the normalisation step to make last step, the netlist generation, trivial.

Netlist Generation In the last step netlists are generated in the form of other, more low-level, HDLs. Currently the compiler supports the generation of VHDL, Verilog, and SystemVerilog netlists.

Intermediate representation

An IR named *Core* is used in the GHC and CλaSH compiler to ease the rewriting and analysis of the Haskell source. It is an abstract representation of the source in the form of a data structure. In GHC, a so-called 'desugaring' step produces the Core IR from the Haskell source. This abstract representation is based on SystemFC [34]: a polymorphic typed λ-calculus. Details of SystemFC are not described in this thesis as these details fall outside the scope of this thesis. In the CλaSH compiler, GHCs Core IR is rewritten to a subset of SystemFC in the Normalisation step.

λ-calculus

λ-calculus is a formalism in the area of mathematical logic where computations can be expressed in function abstractions and function applications. Besides the field of computer science, λ-calculus has found applications in for example linguistics [13] and chemistry [8]. In the domain of computer science, it is the root of functional programming languages. This is also true for Haskell, and hence CλaSH.

e	$::=$	x	*Variable reference*
	$\mid$	$\lambda x \rightarrow e$	*Abstraction*
	$\mid$	$e_1\ e_2$	*Application*

FIGURE 2.4 – Lambda calculus.

An untyped λ-calculus expression grammar is shown in Figure 2.4, as a basic example. It shows the construction of the three different basic syntax structures in λ-calculus; variables, abstractions and applications. Using this grammar, a computational step is described by a so-called β-reduction. This is a formal step where a substitution is performed:

$$(\lambda x \rightarrow e_1)\ e_2 \implies e_1[e_2/x]. \tag{2.1}$$

In this computational step all occurrences of x in e_1, are substituted by e_2. This β-reduction can by applied to an example where a computational step is displayed:

$$(\lambda x \rightarrow x * 2)\ 5 \implies 5 * 2. \tag{2.2}$$

Here x is substituted by 5 resulting in $5 * 2$.

In this thesis, a simply typed λ-calculus [23], is used as an basis for the abstract syntax. In such calculi, primitive data types such as characters, integers, or booleans are defined. In section §3.1, a detailed description of this λ-calculus is provided.

2.1.3 Support for recursion in CλaSH

Currently recursion is supported by the CλaSH compiler to a certain degree. To determine to which extend support is currently available within the CλaSH compiler, two different kinds of supported recursion are distinguished; *value recursion* and *recursion via function definitions*. The two are elaborated separately in the following subsections.

Value recursion

Currently CλaSH does supports value recursion in the form of feedback [5]. An example of such feedback is shown in Listing 2.2. In this example a counter circuit is described which uses a register to capture the state of a *Signal s*. This *Signal* contains the value of the counter and is increased on each clock cycle.

```
counter = s
  where
    s = register 0 (s + 1)
```

LISTING 2.2 – Feedback in CλaSH using recursion on variabels.

Recursion via function definitions

The support of recursion via function definitions is however limited: currently the CλaSH compiler uses unrolling in an attempt to synthesize recursive functions [6, pp. 127], which cannot always produce a result. The procedure creates a specialised function of the original recursive function that can be used for this unrolling. A fixed number of successive unroll actions is tried before the compiler quits the process. This limits the compiler in compile time and possibly the size of the generated netlists. If the base case is not found within the attempt of unrolling, an error is produced by the compiler.

Generally, if a function is data-dependent (see recursion properties defined in section §2.2.5) and the argument of the function is unknown at compile time, inlining of the function often does *not* produce a desired result. The function must be able to handle all possible inputs of the function, which leads to an unfeasibly large hardware design, for even the simplest recursive functions. Thus, the support of generic data-dependent recursive descriptions is currently unsupported by the CλaSH compiler.

2.2 Recursion properties

Recursion is a central concept within this thesis. This section explains basic properties of recursion used in this thesis. This allows us to distinguish between several kinds of recursion. We focus on recursion via function definitions in the remaining parts of this thesis. From a mathematical point of view, a function is recursive if values in the function are calculated by using the same function: the function is defined in terms of itself. One may also speak of self referencing functions.

2.2.1 Linear, binary, and multiple recursion

Let n be the number of recursive calls present in a function. If $n = 1$ then one may speak of a linear recursive function. The factorial function, as defined in equation (2.3), is an example of such a linear recursive function. If $n = 2$ then the recursion function is called: a binary recursive function. Finally, when $n > 1$, the function is called: a multiple recursive function. The function that calculates the nth-Fibonacci's number, as expressed in equation (2.4), is called a multiple — but more often called — binary recursive function.

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \\ n \cdot f(n-1) & \text{if } n > 1 \end{cases}, \qquad n \in \mathbb{Z} \tag{2.3}$$

$$f(n) = \begin{cases} 1 & \text{if } n = 1,2 \\ f(n-1) + f(n-2) & n > 2 \end{cases}, \qquad n \in \mathbb{Z} \tag{2.4}$$

2.2.2 Nested recursion

A recursive call can be nested, which occurs when the value of an argument of a recursive call is also calculated recursively. An example of such a nested recursive function is the Ackermann function *acker* as defined in equation (2.5). If $m, n > 0$ then a nested recursive call is made.

$$acker(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ acker(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ acker(m-1, acker(m, n-1)) & \text{if } m, n > 0 \end{cases} \qquad n, m \in \mathbb{Z} \tag{2.5}$$

2.2.3 Tail recursion

A recursive function is tail recursive if a result of the function is directly determined by a recursive call. The factorial function described in (2.3) is *not* tail-recursive. However this function can be altered to become a tail recursive function. This is accomplished by means of an added argument that accumulates $m * n$ trough each iteration, as is shown in equation (2.6).

$$f(m,n) = \begin{cases} n & \text{if } m = 1 \\ f(m-1, m*n) & \text{if } m > 1 \end{cases}, \qquad n, m \in \mathbb{Z} \tag{2.6}$$

Developers often use this form of recursion, as compilers often can optimize this form of recursion. By means of a process called tail call elimination, tail recursive algorithms can sometimes be computed using only a fixed number of register's, without the use of a growing call stack.

2.2.4 Indirect or mutual recursion

Recursive behaviour can also occur indirectly: if a recursive call is made via another function which is called by the function being defined, indirect recursion occurs. Indirect recursion via functions calling each other is often called mutual recursion. An example is when two functions f and g are specified and f uses g to calculate a value and vice versa.

2.2.5 Data-dependent recursion

The number of recursive function calls can either be *dependent* or *independent* upon the data in the arguments. If the number of recursive function calls are dependent on the data of the argument, the recursion is called *data-dependent* recursion. If the number of recursive function calls are independent on the data, the recursion is called *data-independent*.

2.3 Recursion in Reconfigurable Hardware

In this section, relevant research is covered to gather knowledge of how recursion is used in reconfigurable hardware. Relevant literature is consulted to accomplish this. First, the implementation approaches of recursive algorithms in reconfigurable hardware are covered. Secondly, other FHDLs similar to CλaSH will covered, while paying special attention to the support of recursion in these compilers.

The implementation approaches describing how to implement recursive algorithms in hardware descriptions are researched in §2.3.1. Although these approaches make use of existing low-level HDLs, they are of interest because of the produced hardware architectures. Their approaches to create hardware descriptions in these languages may reveal how recursive algorithms can be implemented in CλaSH.

Besides CλaSH, other compilers exists for FHDLs. These compilers are investigated, while paying extra attention to the support of recursion. In these compilers, the handling of recursion may be interesting. If the compiler strategies from other compilers are applicable to CλaSH, it is highly relevant for this thesis.

2.3.1 Approaches of implementing recursive algorithms in reconfigurable hardware

Several approaches for implementing recursion in reconfigurable hardware are compared in [31]. According to the author, all covered implementation approaches fall into two broad categories: either recursive calls are unrolled into a pipeline circuit, or, a stack architecture is used to implement the recursion.

In the survey [31] several characteristics are compared, such as: applicability, ease of use, occupied hardware resources, and stack usage. Regarding these characteristics, the most promising approach seems the one of Sklyarov et al. [24, 32, 33]. This approach is the only approach which can be applied to any recursive function, is easy to use, occupies a medium number of hardware resources, and requires a stack [31]. This approach is covered in the next section.

Sklyarov et al.

Sklyarov et al. propose a method for implementing recursive algorithms in hardware using Hierarchical Finite-State Machines (HFSMs)[24, 32, 33]. Recursive functions are implemented using a call stack, similarly as used in software, but parallelization occurs between recursive calls. Each function, recursive or not, is referred to as a single module. The combination of multiple modules represent the full circuit.

Two stacks are used: one to preserve the order of function calls between modules, and the other to save the state of the computation described in the separate modules. The hierarchical aspect of this approach comes from the invocation order of different modules, which is maintained by the use of a stack.

Figure 2.5 shows a general outline of the hardware architecture used by Sklyarov et al. The two stacks are controlled by the combinational circuit that updates the stacks depending on the current module and current state. The stacks can also be controlled externally via reset, push, and pop control signals.

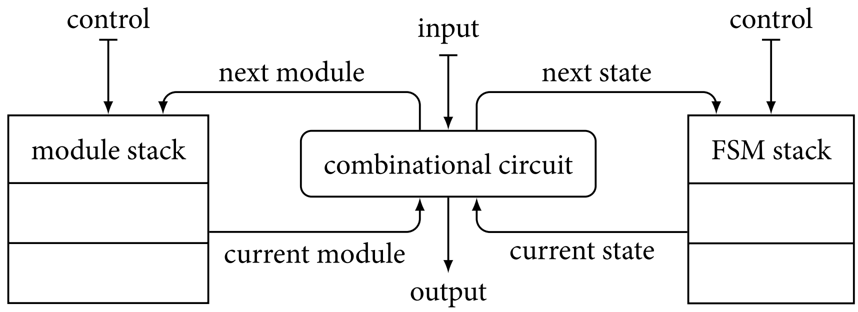

FIGURE 2.5 – Method of Sklyarov et al.

The method described by Sklyarov is useful for implementing recursive algorithms in VHDL. However, the methods are based on manually transforming Handel-C templates into VHDL, hence a language is used as reference which differs much compared to the functional approach as used in CλaSH. Furthermore, the method requires manual implementation steps. Because the focus in this research is to extend the CλaSH compiler with the support of date-dependent recursive functions, we are more interested in automatic transformations instead of manual ones. The featured HFSM architectures however are of interest for this research, as these architecture can be used as templates for transformed algorithms.

2.3.2 Recursion in Functional Hardware Description Languages

Several research projects similar to CλaSH also generate circuits from functional hardware descriptions. However, the support of recursion varies in each project. A comparison of this related work is made in this section.

Edwards et al.

Edwards et al. [40] produce Verilog descriptions out of Haskell sources in a very similar way as CλaSH does. They too use the GHC compiler in their front-end to produce GHC-Core, and from this IR they too use an custom IR to produce Verilog. This IR is similar to the IR used in CλaSH. However, their approach is more behavioral, rather than the structural way CλaSH is set up.

In their work, a series of rewrite steps is used to force the IR in a specific form called Continuation Passing Style (CPS) (explained in further detail in §2.4). This form of the IR allows the recursive algorithms to be handled in a stack architecture that is produced by the compiler.

Although the intention is clear in the papers, no formal rewrite rules are provided. The presented work provides sketches of the algorithms used to derive the stack architectures. Furthermore no details of the actual hardware architecture are provided. This makes it difficult to asses to which extend the research is conducted.

SAFL — Mycroft et al.

Statically Allocated parallel Functional Language (SAFL) [25] is a HDL in which each function is instantiated as a circuit at most once. The term *statically allocated* refers to this property. As a result of this property, the size of circuits solely depends on the size of the text. Only primitive functions and operations are allowed to be duplicated. All other functions are instantiated once and calls to these functions will occur via multiplexers and arbiters.

Feedback is modeled as recursion in SAFL. Only tail-recursive function calls are possible in this model, because only those are statically allocatable, i.e., they require no stack. Listing 2.3 contains an example, copied from [25], which shows such feedback. A shift-add multiplier is implemented using tail recursion.

```
fun mult(x, y, acc) =
  if (x=0 | y=0) then acc
   else mult(x<<1, y>>1, if y.bit 0 then acc+x else acc)
```

Listing 2.3 – shift add multiplier

If a circuit designer wants to compose the same circuit in parallel, the designer must duplicate the functions that describe the circuit. An example of this is shown in Listing 2.4 and Listing 2.5. In Listing 2.4, the function f is called twice in sequential order. The calls to this function are serialised and are handled mutually exclusively. This means that only one instance of the hardware is instantiated per function. One can use functions in parallel by duplicating the function definitions of the same function. In Listing 2.5 multiple instantiations of the same function f are created to obtain such parallelism.

```
fun f x = ...
fun main(x,y) = g(f(x),f(y))
```

Listing 2.4 – f serial execution

```
fun f x = ...
fun f' x = ...
fun main(x,y) = g(f(x),f'(y))
```

Listing 2.5 – f parallel execution

Because SAFL allows only for tail recursive linear recursion, its handling of recursion does not advance the current situation of CλaSH. Furthermore, the single assignment form of SAFL poses an alternative view of the relation between code and hardware. It differs with the view CλaSH has with respect to the formation of hardware.

Verity — Ghica et al.

Ghica et al. describes the synthesis scheme behind Verity in a series of papers called Geometry of Synthesis [15–18]. It is a language which supports higher-order functions, mutable references, and uses an affine type system. In affine type systems, values may not be duplicated. In Verity this only holds for parallel and nested contexts, whereas duplication may occur in sequential context.

Recursion in Verity is supported only with the use of a fixed-point combinator. A fixed point combinator `fix` is a higher-order function that satisfies: $\mathit{fix}\ f = f\ (\mathit{fix}\ f)$. The name is derived from the fixed-point equation: $x = f\ x$ because, when $x = \mathit{fix}\ f$, the fix point combinator satisfies the fix point equation. An example of the usage of this `fix` operator is depicted in Listing 2.6. It illustrates how the (recursive) factorial function is implemented in Verity. Currently, the fix point operator is only unrolled in time by the Verity compiler. However, unrolling in space is theoretically explained in [18].

```
let fact = fix \f.\n. if n == 0$32 then 1$32 else n * f (n-1)
```

LISTING 2.6 – Factorial in Verity, in this example 0$32 and 1$32 means a static 0 and 1 in a 32 bits integer.

Explicit constructs are used in Verity in order to indicate parallel or sequential operating hardware. A particular set of primitive types, called commands, are only allowed to be composed in parallel. For example, logical operations are not allowed to be composed in parallel, whereas for example memory assignment can be composed in parallel. Parallel constructs may not be used in fix-point combinators in Verity.

Recursion is treated as a special case in Verity. It requires the circuit designer to use specific construct to use recursion. Furthermore the explicit constructs for creating parallel and sequential circuits differs much from CλaSH, as CλaSH handles every description combinational by default and allow for sequential circuits trough the use of specific data-type constructions.

Lava — Bjesse et al.

Bjesse et al. describes an embedded language called Lava [9]. The language is called an embedded language because the language is not stand alone, but a library within another language, in this case Haskell. HDL circuit descriptions are produced by executing the program in a standard Haskell environment. The program produces circuits by means of standard execution of the program.

Internally all circuits in Lava eventually are described by a tree-like data-structure. These data-structures can however describe an infinite tree, for example in the case of loops. Therefore, the synthesis function converts these infinite data-structures to a graph representation. Infinite cycles can be detected with the use of observable sharing [19] to obtain these graph representations.

Since finite recursion can be executed by the Haskell compiler, recursive circuits are also produced in Lava. However, the Lava compiler does not support recursion forms that depend on values that are unknown at compile time.

An example of a counter implemented in Lava is listed in Listing 2.7. The function has two signals as argument one for incrementing the counter and the other for resetting it. A register acts as memory element in the circuit and is initially set to 0. Two multiplexer elements, created with a `mux` function, handle the input signals. If the restart signal is high a 0 is chosen, otherwise the register output is chosen. The other multiplexer handles the incrementation of the counter. If the increase signal is high, the value of the register is increased, otherwise not. The resulting value is fed back in the register completing the circuit.

The example contains value recursion for the `loop` and `reg` values. Like CλaSH it can handle such recursion, which is handled by the GHC compiler.

```
counter restart inc = loop
  where reg = register 0 loop
        reg' = mux2 restart (0, reg)
        loop = mux2 inc (reg' + 1, reg')
```

Listing 2.7 – Counter in Lava

Lava is an embedded language and produces circuits by the execution of Haskell programs. This is different from the approach CλaSH uses, as it uses a custom compiler to produce circuits. Recursive descriptions are supported at the level of execution of the Haskell program. This also means true data-dependent recursion cannot be expressed in Lava as it would require to inline all possible outcomes of the circuits. Furthermore, branching must be explicitly constructed in Lava. Branching in CλaSH leads to branching in the circuits, and no explicit constructions are needed.

2.3.3 Conclusion

Although there are many variations in the related field of FHDL compilers, the support of recursion is limited in most of the languages and does not advance CλaSH in the support of recursion. Only the work of Ghica et al. and Edwards et al. do surpass CλaSH current abilities in terms of support of recursion, as they do support the use of true data-dependent recursion. However, Verity is very different compared to the CλaSH language.

The work of Edwards et al. is very similar compared to the work of the CλaSH compiler. They also use an intermediate representation which is very similar to the one used in CλaSH. Their work enables data-dependent recursive descriptions to be used in reconfigurable hardware. Furthermore Edwards et al. also use Haskell as a source language as CλaSH also does. Therefore the method that is described by Edwards et al. is further researched as a basis for this thesis.

2.4 Continuation Passing Style

As previously mentioned in section §2.3.2, in the work of Edwards et al., a series of rewrite steps is performed on a IR to derive a special form, enabling them with the support of recursion. This special form is called *Continuation Passing Style* (CPS). The use of continuations was first described by A. van Wijngaarden in 1964. Later, van Wijngaarden would formulate what now is known as the continuation passing style [28].

CPS is a style of programming where each function call is accompanied with a continuation. A continuation is a description of what to do when a result of a function is ready — sometimes referred as the control. Instead of returning the result of the function, the function returns by calling this continuation with the result as argument. When a program is in CPS, the *control* is made *explicit*. As will become clear in the proceeding chapters, this *explicit control* property of the CPS, is used to derive a stack architecture for recursively defined functions.

Example: Factorial function in CPS

In Haskell, one can write a function in continuation passing style by adding an extra argument, for example k. This argument contains a continuation in the form of a lambda expression. This can be illustrated by the following example in Listing 2.8. In this example the factorial function `fact`, also shown in (2.3), is CPS transformed to `fact_cps`.

```
-- regular factorial
fact 0 = 1
fact n = n * (fact (n-1))

-- cps factorial
fact' n = fact_cps n id

fact_cps 0 k = k 1
fact_cps n k = fact_cps (n-1) (\r->k (n*r))
```

Listing 2.8 – Haskell CPS example. The function `id` is an identity function, which just passes a value.

In the case of $n = 0$ the function returns by applying the continuation k to the result 1. When $n > 0$ a recursive call is made to `fact_cps` applied to $n - 1$ and the continuation in the form of the lambda expression $\lambda\ r \rightarrow\ k\ (n * r)$. The continuation describes what to do when the result of the recursive call is available. In the case of the factorial function one should multiply the result with n. This is exactly what the lambda expression does: the lambda expression is applied to an argument r which contains the result of the recursive call. This result r is multiplied with n just as in the original `fact` description. A wrapper function, `fact'`, applies the CPS transformed function to n variable and to the identity function.

Listing 2.9 evaluates the example with $n = 3$. Continuations are nesting until the recursion ends when $n = 0$. The continuation is then applied to 1. After successively applying the continuation to the intermediate results a final result of 6 is obtained.

```
-- cps factorial
fact' 3 = fact_cps 3 id
      = fact_cps 2 (\r1->id (3*r1))
      = fact_cps 1 (\r2->(\r1->id (3*r1)) (2*r2))
      = fact_cps 0 (\r3->(\r2->(\r1->id (3*r1)) (2*r2)) ((1*r3)))
      = (\r3->(\r2->(\r1->id (3*r1)) (2*r2)) ((1*r3))) 1
      = (\r2->(\r1->id (3*r1)) (2*r2)) (1*1)
      = (\r1->id (3*r1)) (2*1*1)
      = id (3*2*1*1)
      = (3*2*1*1) = 6
```

Listing 2.9 – Haskell CPS example.

As can be seen in the listings, the original factorial function is transformed to a tail recursive function. However, while evolving this function, the added continuation argument increase and decrease in a stacked like manner. This CPS forms is not easily implemented in hardware. However, in the proceeding chapter, a formal methodology is presented derive a stack like architecture from a simply typed lambda calculus.

2.5 Conclusions

This chapter showed several topics of background information that is needed for the rest of this thesis. Two important conclusions can be distilled from the information provided in this chapter:

» A stack architecture can be used to implement data-dependent recursion in reconfigurable hardware. Stack architectures are used in both manually derived implementations of a data-dependent recursive algorithm (as described in §2.3.1), and automatically derived implementations of such algorithms (described in §2.3.2).

» CPS can be used to derive stack architectures from an IR, which is similar to the one used in CλaSH (section §2.3.2).

These conclusions are used in the rest of this thesis to develop a methodology that derives stack architectures from dependent-recursive functions.

3

Methodology

In the previous chapter, both relevant literature and relevant topics as: CλaSH, terminology, and CPS are covered. In this chapter a methodology is developed that will elaborate on how to derive a stack architecture from data-dependent recursive functions.

To derive stack architectures from data-dependent function, a methodology is developed which splits the problem in several steps. First a basic abstract syntax is presented in order to represent recursive functions. This syntax is then used in rewrite rules to force the syntax into a specific form. These rewrite rules are based on the CPS transform, introduced in section §2.4. When the syntax is rewritten to this specific form, one can derive a stack architecture by a procedure also covered in this methodology.

The general outline of this chapter is as follows. In section §3.1, an abstract syntax is presented. This syntax is a basis for the rewrite rules introduced in section §3.2. These rewrite rules force a specific form of the syntax which makes it possible to generate a stack architecture as the one described in section §3.3. The generated stack architecture can then be fed to the CλaSH compiler, which can be used to produce netlist.

3.1 Abstract Syntax

This section introduces a basic grammar for expressions, chosen as a basis for the rewrite rules discussed in this chapter. The syntax is chosen in such a way that recursive algorithms can be expressed and can be used as input for the rewrite rules described later in this chapter. The syntax is related to the abstract syntax used in the CλaSH and GHC (section §2.1.2). This makes it possible to write extensions for these compilers that perform the rewrite steps covered in this chapter.

3.1.1 Expression

Figure 3.1 shows the expressions included in the abstract syntax. The expression grammar e describes a basic typed lambda-calculus language extended with let expressions, case-statements, and a specific kind of application. Some expressions are not part of the allowed input syntax because these expressions play a specific role in the rewrite rules described in §3.2.

e	$::=$	x	*variable*
	$\mid$	i	*literal*
	$\mid$	$@\ e_1\ e_2{}^*$	*serious application*
	$\mid$	$e_1\ e_2$	*trivial application*
	$\mid$	$\textbf{let}\ (x : \tau) = e_1\ \textbf{in}\ e_2{}^*$	*let-expression*
	$\mid$	$\lambda(x : \tau) \to e$	*lambda abstraction*
	$\mid$	$\textbf{case}\ e_s\ \textbf{of}\ \overline{\rho \to e}$	*case-statement*
ρ	$::=$	$(x : \tau)$	*default pattern*
	$\mid$	i	*literal pattern*
	$\mid$	$K\ \overline{(x : \tau)}$	*data pattern*

Figure 3.1 – Expression grammar e. Expressions marked with $*$ exists only during the rewrite steps. They are not allowed as input grammar.

In the presented expression grammar, a distinction between different types of applications is made: applications can be either *trivial* or *serious*. This terminology is adopted from Reynolds [29]. Serious applications are marked with an extra @ sign before the application. This difference plays an important role in the rewrite steps discussed further in this chapter. Section §3.2.1 describes a rewrite step that marks the serious applications. In that section it will become clear how and why this notation is used. Serious applications are only present during the rewrite rules and are not allowed as input grammar.

In the case-expression, the scrutinee of the case-expression: e_s, is matched to the patterns defined as ρ. Three patterns are chosen to be included in the syntax. In the default pattern the scrutinee of the case-expression is simply bound to a variable. Another pattern is the comparison with a literal. If the scrutinee matches the literal i, then the expression e is matched. Finally, e_s can also be matched to data constructors of algebraic data types. In this case K contains the constructor identifier of the data type, and a list of binders $\overline{(x : \tau)}$ that bind the variables of the data constructor. The notation $\overline{(x : \tau)}$ expands to $(x_1 : \tau_1), (x_2 : \tau_2), \cdots, (x_n : \tau_n)$.

3.1.2 Type system

The CλaSH compiler uses types to determine what kind of hardware should be generated. It is therefore important to incorporate types in the aforementioned abstract syntax. A basic type-system is used in the chosen abstract syntax. Figure 3.2 shows the definition of type τ. A type atom w is used to identify different base types. For example Integers, Booleans, etcetera. A function operation on types: $w \to \tau$, is used to make function types. It is not possible to describe higher-order function using this typing system. This simplifies the handling of the abstract syntax used in this thesis.

τ	::=	w	*atom*
	|	$w \to \tau$	*function type*

FIGURE 3.2 – Definition of types τ used in the abstract syntax.

3.1.3 Function definitions

Function definitions are included in the syntax as shown in Figure 3.3. Each function definition consist of an unique variable function name x. This function is bound to a type $\overline{w_{arg}} \to w_{\text{ret}}$. The argument type $\overline{w_{arg}}$ can be used to declare multiple argument types and is expanded with a function type as: $\overline{w_{arg}} \to w_{\text{ret}} \equiv w_1 \to .. \to w_n \to w_{\text{ret}}$. The return type w_{ret} contains the type of the return value.

FunDef	::=	$x : \overline{w_{arg}} \to w_{\text{ret}} = e$	*Function definition*

FIGURE 3.3 – Function definition *FunDef* added to the abstract syntax.

This notation can be used to describe recursive functions. When the function name variable x is used in the function expression e then recursion occurs. This completes the abstract syntax used in the following sections for the rewrite rules.

3.2 Rewrite rules

It is now possible to describe rewrite rules using the abstract syntax constructed in the previous section. Sketches of the rewrite rules are provided in the paper of Edwards et al. [40]. However in order to formalize these steps, another paper from Danvy et al. [12] is used, that covers CPS transformation in great detail.

As mentioned in the background section §2.4, when CPS is applied to the syntax, continuations are passed along with each function call. The continuations describe what to do when the result of a function call is available. However, in the described method of this thesis, the transformation will only be applied to recursive function calls. This results in continuations that describe what to do when a result of a recursive call is available. Rewrite rules covered in this section allow to obtain these continuations.

Fibonacci example

Throughout this section, a recursive function calculating the n-th Fibonacci number, as formulated in equation (2.4), has been chosen as example for the rewrite rules. Using the syntax defined in section §3.1 this function can be described as follows in (3.1). The U_{32} type defines an unsigned 32-bits integer.

$$fib : U_{32} \to U_{32} = \lambda n \to \textbf{case}\ n\ \textbf{of} \begin{cases} 1 \to 1, \\ 2 \to 1, \\ n \to ((+)\ (fib\ (n-1)))\ (fib\ (n-2)) \end{cases} \tag{3.1}$$

3.2.1 Marking serious applications

In the subsequent sections it will become clear that applications that are serious, effectively mark the places where the CPS transform should occur. In the first step we mark serious applications with the notation as defined in section §3.1. The transformation is only applied to the recursive calls, and therefore these are the places that needs to be marked.

Only those applications of which the recursive function that needs to be transformed and are fully *saturated* are marked. A function is saturated if the function is applied to all arguments of the function, or in other words the arity of the function is equal to the number of applied arguments. Using this terminology, only fully saturated recursive function applications are marked as serious applications. This procedure is illustrated by means of the Fibonacci example.

Fibonacci example

We now continue with the Fibonacci example initially described in section §3.2. In this function, $x = fib$; meaning the function name is *fib*. Fibonacci has only one argument, therefore application is saturated when *fib* is applied to that argument. Following this rule, equation (3.2) shows the result of marking all saturated recursive function applications. The serious markings @ are placed at each recursive call at the right place.

$$fib : U_{32} \to U_{32} = \lambda n \to \textbf{case}\ n\ \textbf{of} \begin{cases} 1 \to 1, \\ 2 \to 1, \\ n \to ((+)\ (@\,fib\ (n-1)))\ (@\,fib\ (n-2)) \end{cases} \tag{3.2}$$

3.2.2 Naming serious applications

Serious applications, introduced in the previous rewrite step, are provided with a name using the *naming-step* introduced in this section. This step is executed to provide references to these expressions, which are used in the rewrite steps described later in this chapter.

The now following rewrite steps, follow a notation in the form of multiple rename rules $\mathcal{X}[\![e]\!] \hookrightarrow e'$. In this notation, $\mathcal{X}$ is the name of the rewrite step. Inside the double lined brackets $[\![\;]\!]$, an input expression e is placed. This expression e is rewritten to the term e', if the expression e matches a described pattern. Note that expression e' can also contain rewrite term $\mathcal{X}[\![e]\!]$ which need to be rewritten. The rewrite rules are applied recursively until no further rewrite rules can be applied.

Figure 3.4 shows the rewrite rules $\mathcal{N}[\![\;]\!]$ that form the *naming-step*. This rewrite step is based on the rewrite step described in the paper of Danvy et al. [12, p. 4].

$$
\begin{array}{rcl}
\mathcal{N}[\![x]\!] & \hookrightarrow & x \\
\mathcal{N}[\![i]\!] & \hookrightarrow & i \\
\mathcal{N}[\![\lambda(x:\tau) \to e]\!] & \hookrightarrow & \lambda(x:\tau) \to \mathcal{N}[\![e]\!] \\
\mathcal{N}[\![@\ e_1\ e_2]\!] & \hookrightarrow & \textbf{let}\ x:\tau = \mathcal{N}[\![e_1]\!]\ \mathcal{N}[\![e_2]\!]\ \textbf{in}\ x \\
\mathcal{N}[\![e_1\ e_2]\!] & \hookrightarrow & \mathcal{N}[\![e_1]\!]\ \mathcal{N}[\![e_2]\!] \\
\mathcal{N}[\![\textbf{case}\ e_s\ \textbf{of}\ \overline{\rho \to e}]\!] & \hookrightarrow & \textbf{case}\ \mathcal{N}[\![e_s]\!]\ \textbf{of}\ \overline{\rho \to \mathcal{N}[\![e]\!]}
\end{array}
$$

Figure 3.4 – Naming rewrite rules $\mathcal{N}[\![i]\!]$

Serious applications $@\ e_1\ e_2$, that where introduced in §3.2.1, will be named in this rewrite step. As can be seen, names are only introduced for these serious applications. The names are introduced in the form of a let-expression with an unique variable x bound to a type τ. The type of the introduced variable is equal to the return type of the transformed function, because only the recursive function calls are transformed. Again, notice that these let-expressions are only used in the rewrite steps and are not part of the allowed input syntax.

Fibonacci example

Workings of the naming-step are illustrated by applying these rules to the Fibonacci example. The rewrite rules $\mathcal{N}[\![\;]\!]$ are recursively applied to the result of the previous transformation (where all serious applications are marked (3.2)). When this rewrite step is completed, all recursive function calls will be named in the form of let-expressions.

Equation (3.4) shows the result of applying this rewrite rule to the example. Notice that in this example changes in the expression only occur in the case-statement where the default pattern is matched. In other words: where n is not matched to 1 or 2. Therefore, only this pattern is depicted, and the rest is abbreviated with dots.

$$\mathcal{N}[\![e_{fib}]\!] \hookrightarrow \mathcal{N}[\![\cdots n \to ((+)\ (@fib\ (n-1)))\ (@fib\ (n-2))]\!] \tag{3.3}$$

$$\begin{aligned}\hookrightarrow \cdots n \to ((+)(\mathbf{let}\ (x_1 : U_{32}) = fib\ (n-1)\ \mathbf{in}\ x_1))\\ (\mathbf{let}\ (x_2 : U_{32}) = fib\ (n-2)\ \mathbf{in}\ x_2)\end{aligned} \tag{3.4}$$

As can be seen in the example, after applying the rewrite rule to the expression, each serious application is converted to a let-expression. In this case the unique names are x_1 and x_2. The binders $(x_1 : U_{32})$, and $(x_2 : U_{32})$ bind these unique variables to the return type of the function w_{ret}, which is in this case equal to an integer U_{32}.

3.2.3 Sequencing

In the rewrite step defined next, the previously defined let-expressions are *sequenced*. The presented rewrite rules are based on the 'sequentialize' rewrite rules defined in [12, p. 4]. These rewrite rules force the let-expressions to take a specific form. In this specific form, the following set of conditions is hold for all (sub-) expressions:

i let expressions do not occur in the bound expression e_1 of a let expression,

ii let expressions do not occur in applications,

iii the case scrutinee e_s does not contain let expressions.

Applying these conditions to the expressions yields a *sequence* of let-expressions, hence the name *sequencing-step*. A generic form of such a sequence is shown in (3.5).

$$\cdots \mathbf{let}\ (x_1 : \tau_1) = e_1\ \mathbf{in}\ \mathbf{let}\ (x_2 : \tau_2) = e_2\ \mathbf{in} \cdots \mathbf{let}\ (x_{n-1} : \tau_{n-1}) = e_{n-1}\ \mathbf{in}\ e_n \tag{3.5}$$

This sequence of let-expressions can be interpreted in terms of continuations in the CPS. Assume that the sequence of the let-expressions represent the execution order (from left to right) of each expression. If the first expression e_1 is executed and a result is returned returns, expression e_2 can be executed, therefore e_2 is the continuation of e_1. If e_2 then returns, e_3 should be executed. This procedure repeats itself until e_n is executed. By requiring previously defined conditions to hold, the expressions take the form as shown in (3.5), hence a CPS is found.

Figure 3.5 contains the rewrite rules for the *sequencing-step* $\mathcal{S}\,[\![\ ]\!]$. The let-bindings of let-expressions are collected recursively, in a bottom up traversal, in a list $\overline{v}$ until a lambda or case-expression is pattern matched. At these places, the collected list is converted into a sequence of let-expressions. The lambda and case-expressions act as a barrier for the let-expressions.

$$
\begin{array}{rcl}
\mathcal{S}\llbracket x \rrbracket & \hookrightarrow & (x,\, \varnothing) \\
\mathcal{S}\llbracket i \rrbracket & \hookrightarrow & (i,\, \varnothing) \\
\mathcal{S}\llbracket \lambda(x:\tau) \rightarrow e \rrbracket & \hookrightarrow & (\lambda(x:\tau) \rightarrow \overline{\textbf{let}\ v\ \textbf{in}}\ e',\, \varnothing) \\
 & \textit{where} & (e',\, \overline{v}) = \mathcal{S}\llbracket e \rrbracket \\
\mathcal{S}\llbracket \textbf{let}\ (x:\tau) = e_1\ \textbf{in}\ e_2 \rrbracket & \hookrightarrow & (e_2',\, \overline{v}_1 \mathbin{+\!\!+} \{(x:\tau) = e_1'\} \mathbin{+\!\!+} \overline{v}_2) \\
 & \textit{where} & (e_1',\, \overline{v}_1) = \mathcal{S}\llbracket e_1 \rrbracket \\
 & & (e_2',\, \overline{v}_2) = \mathcal{S}\llbracket e_2 \rrbracket \\
\mathcal{S}\llbracket e_1\ e_2 \rrbracket & \hookrightarrow & (e_1'\ e_2',\, \overline{v}_1 \mathbin{+\!\!+} \overline{v}_2) \\
 & \textit{where} & (e_1',\, \overline{v}_1) = \mathcal{S}\llbracket e_1 \rrbracket \\
 & & (e_2',\, \overline{v}_2) = \mathcal{S}\llbracket e_2 \rrbracket \\
\mathcal{S}\llbracket \textbf{case}\ e_s\ \textbf{of}\ \overline{\rho \rightarrow e} \rrbracket & \hookrightarrow & (\textbf{case}\ e_s'\ \textbf{of}\ \overline{\rho \rightarrow \overline{\textbf{let}\ v\ \textbf{in}}\ e'},\, \overline{v}_s) \\
 & \textit{where} & (e_s',\, \overline{v}_s) = \mathcal{S}\llbracket e_s \rrbracket \\
 & & \overline{(e',\, \overline{v}) = \mathcal{S}\llbracket e \rrbracket} \\
\mathcal{S}'\llbracket e \rrbracket & \hookrightarrow & e' \\
 & \textit{where} & (e',\, \varnothing) = \mathcal{S}\llbracket e_s \rrbracket
\end{array}
$$

FIGURE 3.5 – Sequentialize rewrite rules $\mathcal{S}\llbracket e \rrbracket$.

A specific notation is used to indicate the introduction of these sequences: $\overline{\textbf{let}\ v\ \textbf{in}}\ e'$. Each collected binding out of the list $\{(x_1 : \tau_1) = e_1\}, \cdots, \{(x_n : \tau_n) = e_n\} \in \overline{v}$, is surrounded with a let-expression, producing the desired let-sequence as in equation (3.5). Another notation is used to append two lists: $\mathbin{+\!\!+}$, which is common in Haskell.

In the case-statements, each pattern in $\overline{\rho \rightarrow e}$ introduces its own let-sequence. Let expressions are collected separately in a list v per pattern. The notation $\overline{\rho \rightarrow \overline{\textbf{let}\ v\ \textbf{in}}\ e'}$ denotes that a let-sequence is introduced for each pattern.

All conditions $i \cdots iii$ defined earlier in this section are satisfied when applying the rewrite rules in the *sequence-step* $\mathcal{S}\llbracket\ \rrbracket$. By construction, let-expressions only exist in the form of let-sequences after applying the rewrite rule, which satisfies condition i. Condition ii is satisfied because all let-expressions are lifted outside the applications by construction. Any let-expression inside the scrutinee of the case-statement is lifted out of the scrutinee. This satisfies the last condition iii.

Fibonacci example

The sequencing step $\mathcal{S}\llbracket\ \rrbracket$ can now be applied to the output of the naming step calculated in (3.4) in section §3.2.2. The results of applying these rules are shown in (3.6).

$$
\begin{aligned}
\mathcal{S}' \circ \mathcal{N}\llbracket e_{fib} \rrbracket \hookrightarrow \mathcal{S}\llbracket \cdots n \rightarrow ((+)(\textbf{let } (x_1 : U_{32}) = fib\ (n-1) \textbf{ in } x_1)) & \\
(\textbf{let } (x_2 : U_{32}) = fib\ (n-2) \textbf{ in } x_2) \rrbracket & \qquad (3.6a) \\
\hookrightarrow \cdots n \rightarrow \textbf{let } (x_1 : U_{32}) = fib\ (n-1) \textbf{ in} & \\
\textbf{let } (x_2 : U_{32}) = fib\ (n-2) \textbf{ in } ((+)\ x_1)\ x_2 & \qquad (3.6b)
\end{aligned}
$$

The result of the sequencing step applied to the example can be interpreted as follows: first $fib\ (n-1)$ is bound to $(x_1 : U_{32})$ in the first let expression. The result of the function can be accessed in the let expression via x_1. Next $fib\ (n-2)$ is then bound to $(x_2 : U_{32})$ and the result can be accessed via variable x_2. Lastly, we sum both x_1 and x_2 and this is the result of the function.

In terms of continuations, first $fib\ (n-1)$ is executed. When the result of $fib\ (n-1)$ is known, $fib\ (n-2)$ can be executed. So the continuation of executing $fib\ (n-1)$ is $fib\ (n-2)$. When the result of $fib\ (n-2)$ is known, one can sum both results and this is exactly the continuation of $fib\ (n-2)$: namely $((+)\ x_1)\ x_2$.

3.3 Hardware Generation

Using the rewrite rules of previous section each recursive call is transformed into a sequence of let-expressions as shown in equation (3.5) in §3.2.3. In the same section, this sequence of let-expressions was interpreted in terms of CPS. In this section these let-sequences and this interpretation of these sequences are used to generate hardware.

Recall that in CPS each function call is accompanied with a continuation. This will also be the case in the generated hardware later described in this section. The continuations will consist of hardware descriptions which describe what to do when the result of the called function is available. However, when executing a function, another function call can occur accompanied with another continuation. In order to keep track of the continuations, a stack is introduced. This stack stores a continuation until the function returns the result.

Recall the sequence of let-expressions as defined in equation (3.5). Such a sequence is copied in equation (3.7), and annotated with Roman numerals.

$$
\underbrace{\textbf{let } (x_1 : \tau_1) = e_1 \textbf{ in}}_{(i)}\ \underbrace{\textbf{let } (x_2 : \tau_2) = e_2 \textbf{ in}}_{(ii)}\ \cdots\ \underbrace{\textbf{let } (x_{n-1} : \tau_{n-1}) = e_{n-1} \textbf{ in}}_{(iii)}\ \underbrace{e_n}_{(iv)} \qquad (3.7)
$$

Expression e_1 is first executed, e_2 needs to be executed after e_1 is finished, thus (ii) is a continuation of e_1. This continuation (ii) belonging to e_2, is pushed on the stack, waiting for e_1 to return. If the function returns, the continuation is removed from the stack and the continuation e_2 is executed. However, the continuation e_2 can itself have a continuation, so when executing e_2, the continuation belonging to e_3 is pushed on the stack. This process repeats itself until the last continuation e_n is executed.

In the next subsection a stack architecture is introduced first. Then the results of previous sections are used in order to generate this stack architecture.

3.3.1 Stack Architecture

Figure 3.6 shows a generic version of the stack architecture used in this method. The stack architecture contains two registers which stores a call c and a continuation κ. The continuation register contains the top of the stack. The *next* function contains the logic to decide, given a call and a continuation, what to do next. This results in a stack instruction γ for updating the continuation stack and a follow up call c'

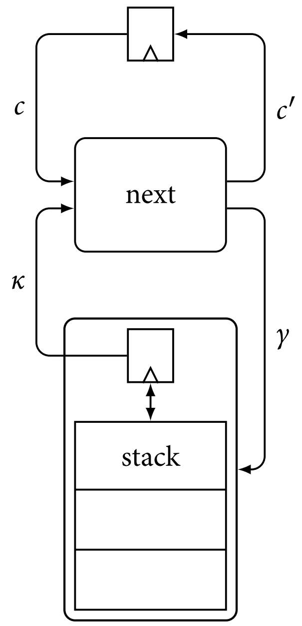

Figure 3.6 – Stack architecture

Continuations

Figure 3.7 contains the abstract representation of the continuations. A continuation describes what to do when the result of a (recursive) computation is completed. Often context dependent variables are needed when evaluating these continuations. The types of these variables

$\overline{\tau}$ are also present in each continuation κ. Each continuation is present in the form of a data type constructor. Each continuation κ is uniquely named.

Cont	::=	$\overline{\kappa\ \overline{\tau}}$	*Continuations, context* $\overline{\tau}$

FIGURE 3.7 – Abstract representation of the *Cont*

Call

As mentioned in the intro of this section, a function can either be called or the function returns a result, when interpreted in CPS. A definition of these calls is introduced in Figure 3.8. Types of the function call contains all the argument data types fetched from τ_{arg} in the function definition of the recursive function (see §3.1.3) definition. Return calls contain the return type of the original function definition w_{ret}.

Call	::=	$F\ \overline{w_{arg}}$	*Function call arguments* $\overline{\tau}$
	|	$R\ w_{ret}$	*Return call with return* τ

FIGURE 3.8 – Definition of *Call*

Stack Instructions

Another output of the *next* function is a stack instruction $\gamma \in \Gamma$. This instruction is used to update the continuation stack. Figure 3.9 describes Γ: the stack instructions.

Γ	::=	Push κ	*Push* $\kappa \in$ *Cont*
	|	Repl κ	*Replace* $\kappa \in$ *Cont*
	|	Pop	*Pop top from stack*
	|	Nop	*Do nothing*
	|	Done	*Finish and handle result*

FIGURE 3.9 – Stack instruction Γ definition

The *Push* instruction pushes a continuation κ on the stack while the *Pop* instruction removes the top instruction from the stack. *Repl* combines these two operations, resulting in a replacement of the top stack element. If nothing is to be done with the stack, the *Nop* instruction is used. Finally the *Done* instruction indicate the completion of a calculation.

3.3.2 Generating the Stack

With the generic description of the stack architecture presented in section §3.3.1, a more detailed description of the stack and how it is automatically generated can be provided.

Reconsider the general result of the sequencing step, as formulated in equation (3.5). After this step, the recursive function calls are in the form of a sequence of let-bindings. This sequence can be related to stack operations and data in the architecture.

Depending on the number of successive let-bindings in one sequence, different stack operations are executed. Equation (3.8) shows the relation between the stack instructions and the let-sequences. The continuations $\kappa \in \mathit{Cont}$ are denoted above the let-expressions. Notice these continuations exactly relate to single let-expressions in a sequence. A tuple below the sequence denote what is to be fed to the *next* function. The tuple consists of a call $c \in \mathit{Call}$ and a stack operation $\gamma \in \Gamma$.

$$\underbrace{\mathbf{let}\ (x_1 : \tau_1) = e_1\ \mathbf{in}}_{(c_1, \mathit{Push}\ \kappa_1)}\ \underbrace{\overbrace{\mathbf{let}\ (x_2 : \tau_2) = e_2\ \mathbf{in}}^{\kappa_1}}_{(c_2, \mathit{Repl}\ (\kappa_2))} \cdots \underbrace{\overbrace{\mathbf{let}\ (x_{n-1} : \tau_{n-1}) = e_{n-1}\ \mathbf{in}}^{\kappa_{n-2}}}_{(c_{n-1}, \mathit{Repl}\ (\kappa_{n-1}))}\ \underbrace{\overbrace{e_n}^{\kappa_{n-1}}}_{(c_n, \mathit{Pop})} \tag{3.8}$$

If the result of e_1 is known, e_2 needs to be executed. This behaviour is produced by pushing continuation κ_1 on the stack. If e_2 returns, the top of the stack is replaced with e_3. The replacements are repeated for each continuation in the sequence until the last one. If the last continuation is executed, a *Nop* instruction is sent to the stack ending the continuations.

There are some cases where continuations can be omitted, which lead to a more efficient way of executing the transformed algorithm. If $e_n = x_{n-1}$ then κ_{n-2} is the last continuation of this sequence, so a direct *Nop* instruction can used and κ_{n-1} can be discarded as a continuation. If a sequence contains only one let-expression, then no continuation need to be pushed on the stack so a *Nop* stack instruction will suffice.

Deriving next function

Previous results now can be combined in deriving the *next* function (as depicted in §3.3.1). Equation (3.9) contains a general outline of the *next* function. The purpose of the final rewrite step introduced in this section, is to fill in the unknowns and generate this *next* function.

$$\mathit{next}\ (c, \kappa) = \mathbf{case}\ c\ \mathbf{of}\ \begin{cases} F\ \text{args} \to e' \\ R\ r \to \mathbf{case}\ \kappa\ \mathbf{of}\ \overline{\alpha} \end{cases} \tag{3.9}$$

The call c can be either a *function call F* or *return call R*. The results of the *next* function is in the form of a tuple containing a next call c' and a stack instruction γ. Continuations are handled when a return call R is invoked, in the form of a case-expression. This case-expression will use data patterns (see §3.1.1) for each continuation. The result of each continuation will also be a tuple with a next call and stack instruction. So elements of $\overline{\alpha}$ will be in the form of $\kappa\ \overline{(x : \tau)} \to e_\kappa$.

Figure 3.10 contains the *derive next* rewrite rule $\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![\ ]\!]$ which collects e' and $\overline{\alpha}$ from an input expression e. This rewrite rule perform two tasks:

i All results of *next* the function must be in the form of a tuple containing a call and stack instruction.

ii Continuations are collected in the form of a data pattern for a case-expression which handles the continuations.

This rewrite rule is called with a parameter ϕ to indicate if the continuation is the first in a sequence. This parameter is used in the helper functions $\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![\ ]\!]$ and $\mathcal{DN}_{\mathcal{K}}\ \phi\ [\![\ ]\!]$ in order to determine which stack operation belongs to the current expression. The parameter is initially true $\top$. The result of the rewrite rule is a tuple $(e',\ \overline{\alpha})$ which are used in the *next* function (3.9).

$$
\begin{array}{rcl}
\mathcal{DN}\ \phi\ [\![x]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![x]\!],\ \varnothing) \\
\mathcal{DN}\ \phi\ [\![i]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![x]\!],\ \varnothing) \\
\mathcal{DN}\ \phi\ [\![e_1\ e_2]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![e_1\ e_2]\!],\ \varnothing) \\
\mathcal{DN}\ \phi\ [\![\mathbf{let}\ (x_1, \tau) = e_1\ \mathbf{in}\ x_2]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![e_1]\!],\ \varnothing) \\
 & \mathit{if} & x_1 = x_2 \\
\mathcal{DN}\ \phi\ [\![\mathbf{let}\ (x : \tau) = e_1\ \mathbf{in}\ e_2]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{K}}\ \phi\ [\![e_1]\!]\ \kappa,\ \kappa \to e'[x/r]; \overline{\alpha}) \\
 & \mathit{where} & (e',\ \overline{\alpha}) = \mathcal{DN}\ \bot\ [\![e_2]\!] \\
 & & \kappa = \kappa_{new}\ FV(\mathbf{let}\ (x : \tau) = e_1\ \mathbf{in}\ e_2) \\
\mathcal{DN}\ \phi\ [\![\lambda b \to e]\!] & \hookrightarrow & (\lambda b \to e',\ \overline{\alpha}) \\
 & \mathit{where} & (e',\ \overline{\alpha}) = \mathcal{DN}\ \phi\ [\![e]\!] \\
\mathcal{DN}\ \phi\ [\![\mathbf{case}\ e_s\ \mathbf{of}\ \overline{\rho \to e}]\!] & \hookrightarrow & (\mathbf{case}\ e_s\ \mathbf{of}\ \overline{\rho \to e'},\ \overline{\alpha}) \\
 & \mathit{where} & \overline{(e',\ \overline{\alpha}) = \mathcal{DN}\ \phi\ [\![e]\!]} \\
 & & \\
\mathcal{DN}_{\mathcal{R}}\ \top\ [\![e]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{C}}\ [\![e]\!], \mathit{Nop}) \\
\mathcal{DN}_{\mathcal{R}}\ \bot\ [\![e]\!] & \hookrightarrow & (\mathcal{DN}_{\mathcal{C}}\ [\![e]\!], \mathit{Pop}) \\
\mathcal{DN}_{\mathcal{K}}\ \top\ [\![e]\!]\ \kappa & \hookrightarrow & (\mathcal{DN}_{\mathcal{C}}\ [\![e]\!], \mathit{Push}\ \kappa) \\
\mathcal{DN}_{\mathcal{K}}\ \bot\ [\![e]\!]\ \kappa & \hookrightarrow & (\mathcal{DN}_{\mathcal{C}}\ [\![e]\!], \mathit{Repl}\ \kappa) \\
\mathcal{DN}_{\mathcal{C}}\ [\![e]\!] & \hookrightarrow & \begin{cases} e[f/F] & \mathit{if}\ f \in FV(e) \\ R\ e & \mathit{otherwise} \end{cases}
\end{array}
$$

FIGURE 3.10 – Derive next rewrite rules $\mathcal{DN}\ \phi\ [\![\ ]\!]$ for deriving next function in the stack architecture together with subroutines $\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![\ ]\!]$ and $\mathcal{DN}_{\mathcal{K}}\ \phi\ [\![\ ]\!]$, and $\mathcal{DN}_{\mathcal{C}}\ [\![\ ]\!]$.

The subroutines $\mathcal{DN}_{\mathcal{R}}\ \phi\ [\![\ ]\!]$ and $\mathcal{DN}_{\mathcal{K}}\ \phi\ [\![\ ]\!]\kappa$ add stack instructions to the expressions. These subroutines both use another subroutine $\mathcal{DN}_{\mathcal{C}}\ [\![\ ]\!]$ which makes a call instruction of the currently handled expression. This is accomplished by checking if the original function name f is in the free variables of the currently handled expression e. If this is true, the function name is simply substituted by the constructor name F. Otherwise the expression must be a return statement, so a return constructor name R is applied to the expression e.

Another task in deriving the *next* function is the collection of continuations. As can be seen in the definition of the rewrite rules, continuations are only introduced for each let-expression where $(x_1 \neq x_2)$. As already stated, the continuations are in the form of $\kappa\ \overline{(x : \tau)} \rightarrow e_\kappa$. The data constructor κ is named uniquely and will be of the form as presented in Figure 3.11. The first continuation will only contain the free variables used in the rest of sequence of continuations. Because the intermediate results of each successive continuation also can be used in the rest of the continuations, this value is added to the data constructor of the continuation when the result of the continuation is known.

Cont	::=	$\kappa_1\ \overline{\tau_{fv}}$	κ_1 *with free variables* $\overline{\tau_{fv}}$
	$\mid$	$\kappa_2\ \overline{\tau_{fv}}\ \tau_{x_1}$	
	$\mid$	$\kappa_3\ \overline{\tau_{fv}}\ \tau_{x_1}\ \tau_{x_2}$	

FIGURE 3.11 – Details of the *Cont* datatype

This completes the *derive-next* rewrite step as both task i and ii formulated earlier in this section are handled by this rewrite step. The remainder of this chapter will cover the application of the *derive-next* rewrite step to the Fibonacci example.

Fibonacci example

Returning to the Fibonacci example, the *next* description can be generated by applying the $\mathcal{DN}\ \top\ [\![\ ]\!]$ rewrite rule to the results found in equation (3.6). The *derive-next* rewrite rule collects the rewritten function description e', and the continuations $\overline{\alpha}$.

$$\begin{aligned}
\mathcal{DN}\ \top \circ \mathcal{S}' \circ \mathcal{N}[\![e_{fib}]\!] \hookrightarrow\ & \mathcal{DN}\ \top\ [\![\cdots n \rightarrow \textbf{let}\ (x_1 : U_{32})) = fib\ (n-1) \\
& \textbf{in let}\ (x_2 : U_{32}) = fib\ (n-2)\ \textbf{in}\ ((+)\ x_1)\ x_2]\!] && (3.10a) \\
& \hookrightarrow (e',\ \overline{\alpha}) && (3.10b) \\
& ,where \\
e' &= \textbf{case}\ n\ \textbf{of} \begin{cases} 1 \rightarrow (R\ 1,\ Nop) \\ 2 \rightarrow (R\ 1,\ Nop) \\ n \rightarrow (F\ (n-1),\ Push\ (\kappa_1\ n)) \end{cases} && (3.10c) \\
\overline{\alpha} &= \begin{cases} \kappa_1\ n \rightarrow (F\ (n-2), Repl\ (\kappa_2\ n\ r)) \\ \kappa_2\ n\ x_1 \rightarrow (R\ (x_1 + r), Pop) \end{cases} && (3.10d)
\end{aligned}$$

These results can now be plugged into the *next* description from (3.9). Equation (3.11) shows the resulting *next* function for the Fibonacci example.

$$next\ (c, \kappa) = \textbf{case}\ c\ \textbf{of} \begin{cases} F\ \text{n} \rightarrow \textbf{case}\ n\ \textbf{of} \begin{cases} 1 \rightarrow (R\ 1,\ Nop) \\ 2 \rightarrow (R\ 1,\ Nop) \\ n \rightarrow (F\ (n-1),\ Push\ (\kappa_1\ n)) \end{cases} \\ R\ r \rightarrow \textbf{case}\ \kappa\ \textbf{of} \begin{cases} \kappa_1\ n \rightarrow (F\ (n-2),\ Repl\ (\kappa_2\ n\ r)) \\ \kappa_2\ n\ x_1 \rightarrow (R\ (x_1 + r),\ Pop) \\ \kappa_0 \rightarrow (R\ r, Done) \end{cases} \end{cases} \tag{3.11}$$

This provides a *next* description for the for the stack architecture described in §3.3.1. This *next* description, together with a basic description for the stack architecture, can be fed to the CλaSH compiler to generate hardware.

Table 3.1 contains an evaluation of the *next* function, as defined in equation (3.11), with an input of F 3. Each successive application of the *next* function is numbered in this table. Each row consist of a *next* function applied applied to a call c and continuation κ. The resulting tuple (c', γ) is listed together with the stack after applying the stack instruction.

The result of the calculation of Fibonacci 3 is known after 6 successive applications of the *next* function. This concludes the example of transforming the original Fibonacci description to an implemented in a stack architecture.

	$next(c, \kappa)$	=	(c', γ)	Stack
				$[\kappa_0]$
1	$next\ (F\ 3, \kappa_0)$	=	$(F\ 2, Push\ (\kappa_1\ 3))$	$[\kappa_1\ 3, \kappa_0]$
2	$next\ (F\ 2, \kappa_1\ 3)$	=	$(R\ 1, Nop)$	$[\kappa_1\ 3, \kappa_0]$
3	$next\ (R\ 1, \kappa_1\ 3)$	=	$(F\ 1, Repl\ (\kappa_2\ 3\ 1))$	$[\kappa_2\ 3\ 1, \kappa_0]$
4	$next\ (F\ 1, \kappa_2\ 3\ 1)$	=	$(R\ 1, Nop)$	$[\kappa_2\ 3\ 1, \kappa_0]$
5	$next\ (R\ 1, \kappa_2\ 3\ 1)$	=	$(R\ 2, Pop)$	$[\kappa_0]$
6	$next\ (R\ 2, \kappa_0)$	=	$(R\ 2, Done)$	$[\kappa_0]$

Table 3.1 – Evaluation of *next* function in the case of Fibonacci

4

Implementation

In the previous chapter, a methodology is presented where rewrite rules are used to transform recursive descriptions into a stack architecture. An experimental rewrite program is developed as a proof of concept, implementing the presented rewrite rules. Furthermore, a specific hardware design is chosen as a template for the generated hardware. In this chapter, both the implementation details of the rewrite rules, and generated stack architecture will be elaborated.

4.1 Abstract syntax and rewrite rules

In the methodology covered in Chapter 3, an abstract syntax and rewrite rules are formally described. In the experimental rewrite program these formal descriptions are implemented in the Haskell language. Due to the similarities between the formal descriptions and the implementation of the rewrite program, no further implementation details have to be elaborated. Therefore, only the relations between the methodology and source code are covered in this section.

Table 4.1 shows an overview of the source code in the appendix linked to the method sections.

Subject		Section	Appendix
abstract syntax	e	§3.1	Appendix A.1
naming rewrite rule	$\mathcal{S}[\![\ ]\!]$	§3.2.2	Appendix A.2.1
sequentialize rewrite rule	$\mathcal{N}[\![\ ]\!]$	§3.2.3	Appendix A.2.2
deriving-next rewrite rule	$\mathcal{DN}\ \sigma\ [\![\ ]\!]$	§5.1.1	Appendix A.2.3
full transform	$\mathcal{DN}\ \top \circ \mathcal{S}' \circ \mathcal{N}[\![\ ]\!]$	—	Appendix A.2.4

Table 4.1 – Appendix source code references

4.2 Stack Architecture

As is elaborated in section §3.3.1, the stack architecture consists of more than only the *next* function. Only detailed descriptions of deriving the *next* function are presented until now. In this section a concrete hardware design of the stack architecture is elaborated. This stack architecture is written in the CλaSH language and is included in the appendix. However, before referring to the source code, an introduction to the hardware is made first.

A hardware implementation is proposed in this section and is described as a Mealy machine description, as discussed in section §2.1.1. This description contains the combinational logic and state necessary to: store and retrieve the continuations for the *next* descriptions, the *next* function itself, and a return value if a result is ready. The continuation stack is stored in a *Random Access Memory (RAM)* type of memory. In FPGAs it is common to use a *Block RAM* (BRAM) for medium sized memory which needs to be accessed fast. This Block RAM (BRAM) is used in the FPGA to store the continuation in a stack like manner.

The CλaSH prelude supports the use of this commonly used BRAM. However, a pitfalls is accompanied when using this BRAM. The BRAM instantiated by CλaSH is not of the type *pass-trough*, meaning that when writing and reading from the same address, the BRAM returns the old value instead of the newest value. The Mealy circuit must be handle cooping with this pitfall.

4.2.1 Abstract implementation of the stack architecture

Figure 4.1 depicts the stack architecture consisting of the Mealy description wired to the BRAM. An internal state of the Mealy machine stores a call $c \in Call$, a continuation $\kappa \in Cont$, and a stack pointer p. Output signals of the Mealy machine steer the BRAM, and present the result when ready. Read results of the BRAM are available as input signal of the stack update Mealy machine.

The `stackUpdate` function contains the generated *next* description, which determines the next call c and the stack instruction $\gamma \in \Gamma$ and the logic to control the BRAM. The next call is stored directly as internal state. The instruction γ determines all other states and the output.

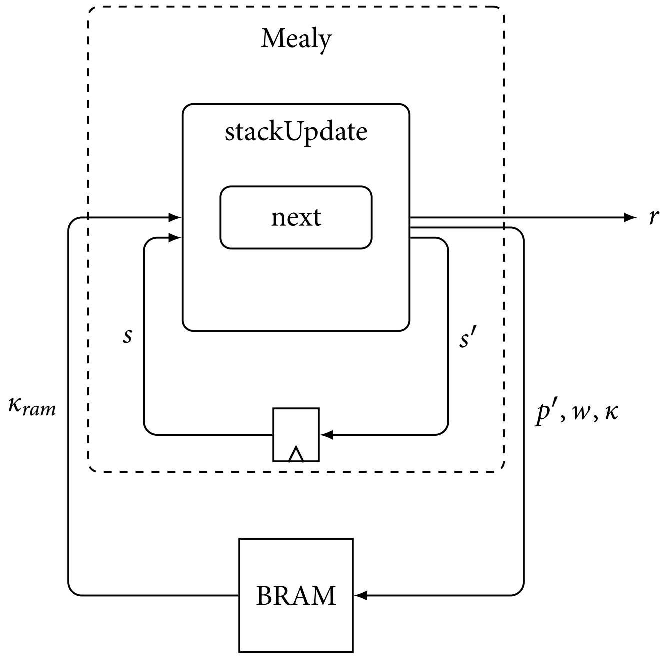

FIGURE 4.1 – Implementation of stack architecture using a Mealy machine with internal state $s = (c, \kappa, p)$ which controls a BRAM.

The BRAM pitfall is circumvented by storing the top of the stack internally in the Mealy machine. The BRAM contains the tail of the stack.

4.2.2 Implementation details of the stack architecture

Figure 4.2 contains a detailed view of the stack architecture. It contains the full circuit of the stack architecture. The pointer is updated according to the stack instruction, and is used in the BRAM to select both the read and write address. If the instruction is a push, the current continuation in the top register must be stored in the BRAM, thus the write bit is enabled. The new top is chosen according to the instruction; a push or replace instruction contains a continuation which is the new top. If a pop occurs, the new top is the continuation read from the BRAM. In case of a nop or done, the new continuation is simply the current continuation. Appendix B contains the CλaSH code of the stack architecture.

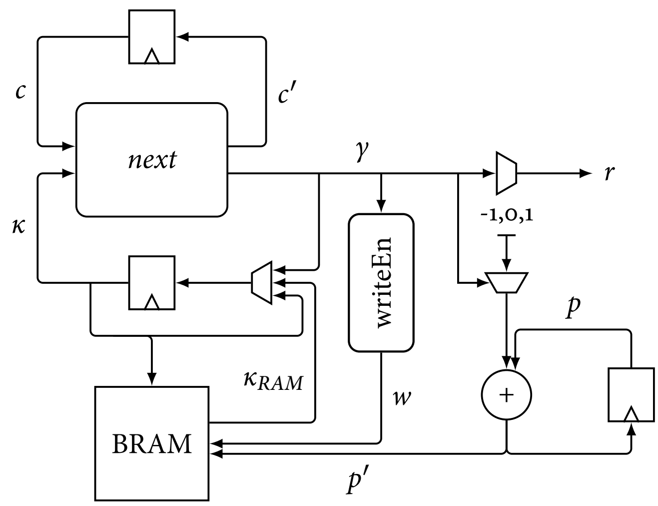

Figure 4.2 – Detailed stack architecture

5

Results

Using the combined results of the methodology described in Chapter 3 and the implementation of this methodology as elaborated in Chapter 4, hardware descriptions of stack architectures can be generated. In this chapter, more examples will be subjected to the developed rewrite rules. Furthermore, the results of the generated stack architectures will be synthesized and these results will also be evaluated.

Several recursive algorithms are used to test the presented rewrite rules and stack architecture. Figure 5.1 shows the path from recursive algorithm to FPGA. Several recursive algorithms are written in the abstract syntax (as defined in section §3.1), for testing purposes. Using the rewrite rules presented in section §3.2, a stack architecture is derived when combining the derived function with the CλaSH template defined in section §4.2.2. Using the CλaSH compiler, VHDL descriptions then be obtained. These VHDL descriptions are synthesized using the Altera Quartus tooling for a specific FPGA.

$$\textit{Abstract Syntax} \xrightarrow{\textit{rewrite rules}} \textit{StackArch} \xrightarrow{\textit{C}\lambda\textit{aSH}} \textit{VHDL} \xrightarrow{\textit{Quartus}} \textit{FPGA}$$

Figure 5.1 – From recursive descriptions in abstract syntax to FPGA

There are several points where the translation is tested. Using manual evaluation of the *next* function, descriptions produced by the rewrite rules are verified for their correctness. This validation is also performed with the Fibonacci example in Chapter 3. The produced stack architectures in CλaSH are verified in the interactive environment of the CλaSH compiler.

QuestaSim allows simulation of the VHDL descriptions produced by CλaSH. Visual inspection can also be used to verify synthesis of Quartus, which can be performed in the Register Transfer Level (RTL)-Viewer chip planner.

In the next section, other generated hardware descriptions, derived from recursive algorithms, are elaborated. Instead of the Fibonacci example in previous chapter, other algorithms will be subjected to the rewrite rules defined in section §3.2. The generated hardware descriptions will be synthesised for a specific FPGA as will be shown in section §5.2. In this section the results of the syntheses will be elaborated.

5.1 Rewriting other recursive algorithms

In Chapter 3, the Fibonacci example is used to support the explanation of the rewrite steps. The mathematical definition of this function is already elaborated a chapter earlier in section §2.2. In the same section two other mathematical definitions are presented: the *factorial* function and the *Ackermann* function. In this section, the results of previous chapters chapter will be used to generate stack architecture descriptions for these examples.

5.1.1 Factorial

The factorial function, as mathematically defined in equation (2.3) in section §2.2, can be expressed using the abstract syntax (see §3.1) which is introduced in the methodology. Using this abstract syntax, the factorial function is defined ass follows in (5.1).

$$\mathit{fact} : U_{32} \to U_{32} = \lambda n \to \textbf{case } n \textbf{ of } \begin{cases} 0 \to 1 \\ n \to ((*)\ n)\ (\mathit{fact}\ (n-1)) \end{cases} \tag{5.1}$$

Using the description defined in the abstract syntax, one can apply the rewrite rules as defined in Chapter 3 with the goal of deriving the `next` function (see section §5.1.1). First the *naming step* $\mathcal{N}[\![\ ]\!]$ is applied, then the *sequence step* $\mathcal{S}[\![\ ]\!]$, and finally the *derive next* $\mathcal{DN}\ \phi\ [\![\ ]\!]$ step.

The results of applying these subsequent steps to the factorial function is shown in equation (5.2). Notice the occurrence of a serious application in the input of the naming step. This is still a non automated step but is trivial to automate.

$$\begin{aligned}
\mathcal{N}[\![fact]\!] &= \mathcal{N}[\![\cdots n \to ((*)\ n)\ (@fact\ (n-1))]\!] \\
&= \cdots n \to ((*)\ n)\ (\mathbf{let}\ x_1 = fact\ (n-1)\ \mathbf{in}\ x_1) && (5.2a) \\
\mathcal{S}' \circ \mathcal{N}[\![fact]\!] &= \cdots n \to \mathbf{let}\ x_1 = fact\ (n-1)\ \mathbf{in}\ ((*)\ n)\ x_1 && (5.2b) \\
\mathcal{DN}\ \top \circ \mathcal{S}' \circ \mathcal{N}[\![fact]\!] &= (e', \overline{\alpha}) && (5.2c) \\
&\quad \mathbf{where} \\
&\quad e' = \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to (R\,1,\ Nop) \\ n \to (F\ (n-1),\ Push\ (\kappa_1\ n)) \end{cases} \\
&\quad \overline{\alpha} = \begin{cases} \kappa_1\ n \to (R\ ((*)\ n)\ r, Pop) \end{cases}
\end{aligned}$$

After applying all three steps, a description of e' and $\overline{\alpha}$ is known. This is then used in the `next` description as shown in equation (3.9) in section §5.1.1. This results in the following next description as shown in equation (5.3).

$$next\ (c, \kappa) = \mathbf{case}\ c\ \mathbf{of} \begin{cases} F\ n \to \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to (R\,1,\ Nop) \\ n \to (F\ (n-1),\ Push\ (\kappa_1\ n)) \end{cases} \\ R\ r \to \mathbf{case}\ \kappa\ \mathbf{of} \begin{cases} \kappa_1\ n \to (R\ ((*)\ n)\ r, Pop) \\ \kappa_0 \to (R\ r,\ Done) \end{cases} \end{cases} \tag{5.3}$$

Only one continuation is introduced: κ_1. This continuation multiplies the result of the recursive call $F\ (n-1)$ with the n stored in the continuation data type on the stack.

Evaluation of the factorial next function

In order to asses the behaviour of the next function, an evaluation of this function is performed while bookkeeping the results of applying the next function, and the stack manually. Just as is performed in the end of section of Chapter 3.

Table 5.1 shows the evaluation of the *next* generated from the factorial function applied to an input of $(F\ 3, \kappa_0)$. Each of the intermediate results is listed in a separate row. The state of the stack after applying the *next* function is displayed in a separate column.

After applying the next function 8 times successively, the result of the computation is known. In this case the result is 6 which is indeed true as $3! = 3 * 2 * 1 = 6$.

	$next(c,\kappa)$	=	(c',γ)	Stack
				$[\kappa_0]$
1	$next\ (F\ 3, \kappa_0)$	=	$(F\ 2, Push\ (\kappa_1\ 3))$	$[\kappa_1\ 3, \kappa_0]$
2	$next\ (F\ 2, \kappa_1\ 3)$	=	$(F\ 1, Push\ (\kappa_1\ 2))$	$[\kappa_1\ 2, \kappa_1\ 3, \kappa_0]$
3	$next\ (F\ 1, \kappa_1\ 2)$	=	$(F\ 0, Push\ (\kappa_1\ 1))$	$[\kappa_1\ 1, \kappa_1\ 2, \kappa_1\ 3, \kappa_0]$
4	$next\ (F\ 0, \kappa_1\ 1)$	=	$(R\ 1, Nop)$	$[\kappa_1\ 1, \kappa_1\ 2, \kappa_1\ 3, \kappa_0]$
5	$next\ (R\ 1, \kappa_1\ 1)$	=	$(R\ 1, Pop)$	$[\kappa_1\ 2, \kappa_1\ 3, \kappa_0]$
6	$next\ (R\ 1, \kappa_1\ 2)$	=	$(R\ 2, Pop)$	$[\kappa_1\ 3, \kappa_0]$
7	$next\ (R\ 2, \kappa_1\ 3)$	=	$(R\ 6, Pop)$	$[\kappa_0]$
8	$next\ (R\ 6, \kappa_0)$	=	$(R\ 6, Done)$	$[\kappa_0]$

TABLE 5.1 – Evaluation of *next* function in the case of Factorial

5.1.2 Ackermann

Another algorithm which is subjected to the rewrite rules is the Ackermann function. The same procedure as used in previous example is used for this algorithm. A mathematical description of the Ackermann function is provided in equation (2.5) in section §2.2. This function is an example of a nested recursive function. When expressed in the abstract syntax, the following recursive definition (5.4) is obtained.

$$
\begin{aligned}
acker : U_{32} \to U_{32} \to U_{32} = \lambda m \to \lambda n \to \mathbf{case}\ m\ \mathbf{of} \\
\begin{cases} 0 \to n + 1 \\ m \to \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to (acker\ (m-1))\ 1 \\ n \to (acker\ (m-1))\ ((acker\ m)\ (n-1)) \end{cases} \end{cases}
\end{aligned}
\tag{5.4}
$$

Notice that the Ackermann function has an arity of two, instead of previous algorithms. Therefore, two lambda abstractions are used for the function expression.

Using the defined abstract syntax definition of the Ackermann function, the rewrite rules of can be applied to again obtain a `next` function. Rewriting these expressions leads to the following results depicted in equations (5.5). Notice that, again, the serious applications are marked at the input of for the naming step.

$$\mathcal{N}[\![acker]\!] = \mathcal{N}[\![\cdots \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to @\ (acker\ (m-1))\ 1 \\ n \to @\ (acker\ (m-1)) \\ \qquad (@\ (acker\ m)\ (n-1)) \end{cases}]\!] \tag{5.5a}$$

$$= \cdots \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to \mathbf{let}\ x_1 = (acker\ (m-1))\ 1\ \mathbf{in}\ x_1 \\ n \to \mathbf{let}\ x_2 = (acker\ (m-1)) \\ \qquad (\mathbf{let}\ x_1 = acker\ m\ (n-1)\ \mathbf{in}\ x_1)\ \mathbf{in}\ x_2 \end{cases} \tag{5.5b}$$

$$\mathcal{S}' \circ \mathcal{N}[\![acker]\!] = \cdots \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to \mathbf{let}\ x_1 = acker\ (m-1)\ 1\ \mathbf{in}\ x_1 \\ n \to \mathbf{let}\ x_1 = acker\ m\ (n-1)\ \mathbf{in} \\ \mathbf{let}\ x_2 = acker\ (m-1)\ x_1\ \mathbf{in}\ x_2 \end{cases} \tag{5.5c}$$

$$\mathcal{DN}\top \circ \mathcal{S}' \circ \mathcal{N}[\![acker]\!] = (e', \overline{\alpha}) \tag{5.5d}$$

$$\mathbf{where}$$

$$e' = \cdots \mathbf{case}\ n\ \mathbf{of} \begin{cases} 0 \to (F\ (m-1)\ 1,\ Nop) \\ n \to (F\ m\ (n-1),\ Push\ (\kappa_1\ m\ n)) \end{cases}$$

$$\overline{\alpha} = \begin{cases} \kappa_1\ m\ n \to (F\ (m-1)\ r, Pop) \end{cases}$$

Using the obtained descriptions of the modified function expression e' and the continuations $\overline{\alpha}$, the `next` function template can be filled in. The result of this is displayed in equation (5.6).

$$next\ (c, \kappa) = \mathbf{case}\ c\ \mathbf{of} \begin{cases} F\ m\ n \to \mathbf{case}\ m\ \mathbf{of} \\ \qquad \begin{cases} 0 \to (R\ n+1,\ Nop) \\ m \to \mathbf{case}\ n\ \mathbf{of} \\ \qquad \begin{cases} 0 \to (F\ (m-1)\ 1,\ Nop) \\ n \to (F\ m\ (n-1),\ Push\ (\kappa_1\ m\ n)) \end{cases} \end{cases} \\ R\ r \to \mathbf{case}\ \kappa\ \mathbf{of} \begin{cases} \kappa_1\ m\ n \to (F\ (m-1)\ r, Pop) \\ \kappa_0 \to (R\ r,\ Done) \end{cases} \end{cases} \tag{5.6}$$

	$next(c, \kappa)$	=	(c', γ)	Stack
				$[\kappa_0]$
1	$next\ (F\ 1\ 2, \kappa_0)$	=	$(F\ 1\ 1, Push\ (\kappa_1\ 1\ 2))$	$[\kappa_1\ 1\ 2, \kappa_0]$
2	$next\ (F\ 1\ 1, \kappa_1\ 1\ 2)$	=	$(F\ 1\ 0, Push\ (\kappa_1\ 1\ 1))$	$[\kappa_1\ 1\ 1, \kappa_1\ 1\ 2, \kappa_0]$
3	$next\ (F\ 1\ 0, \kappa_1\ 1\ 1)$	=	$(F\ 0\ 1, Nop)$	$[\kappa_1\ 1\ 1, \kappa_1\ 1\ 2, \kappa_0]$
4	$next\ (F\ 0\ 1, \kappa_1\ 1\ 1)$	=	$(R\ 2, Nop)$	$[\kappa_1\ 1\ 1, \kappa_1\ 1\ 2, \kappa_0]$
5	$next\ (R\ 2, \kappa_1\ 1\ 2)$	=	$(F\ 0\ 2, Pop)$	$[\kappa_1\ 1\ 2, \kappa_0]$
6	$next\ (F\ 0\ 2, \kappa_1\ 1\ 2)$	=	$(R\ 3, Nop)$	$[\kappa_1\ 1\ 2, \kappa_0]$
7	$next\ (R\ 3, \kappa_1\ 1\ 2)$	=	$(F\ 0\ 3, Nop)$	$[\kappa_1\ 1\ 2, \kappa_0]$
8	$next\ (F\ 0\ 3, \kappa_1\ 1\ 2)$	=	$(R\ 4, Pop)$	$[\kappa_0]$
9	$next\ (R\ 4, \kappa_0)$	=	$(R\ 4, Done)$	$[\kappa_0]$

TABLE 5.2 – Evaluation of *next* function in the case of Ackermann

As can be seen in the `next` description, only one continuation is introduced. This continuation is pushed onto the stack when $m, n > 0$. This is the effect of omitting the introduction of continuations when a tail call occurs as discussed in §3.3.2.

Evaluation of the Ackermann next function

Table 5.2 shows the evaluation of the *next* generated from the Ackermann function applied to an input of $(F\ 1\ 2, \kappa_0)$. Each of the intermediate results is listed in a separate row. The state of the stack after applying the *next* function is displayed in a separate column.

The result of *acker* 1 2 is 4 as listed in the final row of the table. As can be seen: many recursive calls are made when calculating the Ackermann function. This is a property of the Ackermann algorithm.

5.2 Synthesis Results

In this section results of the synthesis of previously obtained hardware architectures will be elaborated. The stack architectures of the Fibonacci algorithm, the Factorial algorithm, and the Ackermann algorithm; together with other non-elaborated algorithms, will be used to obtain these synthesis results. Hardware descriptions as elaborated in section §4.2 from Chapter 4 are used to generate VHDL from the CλaSH compiler.

The hardware descriptions are synthesised using the Altera Quartus 15 tooling. A specific FPGA is chosen: the Cyclone IV `EP4CE22F17C6N` FPGA. This FPGA is used in a developer board called the `DE0`-nano board. It is relatively cheap and the FPGA is in the low-end range in terms of resources.

Algorithm	f_{max} in MHz	LEs	Registers	BRAM width
Fibonacci 32 bit	215.05	406	107	66
Factorial 32 bit	125.16	268	74	33
FactorialTail 32 bit	110.86	169	65	0
Ackermann 32 bit	184.98	424	117	65
GCD 32 bit	9.37	1,157	65	0

TABLE 5.3 – Results of the synthesis using Altera Quartus 15 tooling, targeting a Cyclone IV `EP4CE22F17C6N` FPGA.

Several post-synthesis attributes of the synthesis are used in this thesis to indicate the performance of the translated functions. These include:

maximum frequency f_{max} A maximum operational frequency f_{max}, while operating in 85° Celsius, is listed.

Logic Elements (LEs) The number of LEs (logic containing a Look Up Table (LUT)) are compared to indicate chip area consumption. These elements also contain registers; the number of LEs includes register counts.

Registers Separate counting of the registers usage used in the design. Registers are part of the LE blocks, but are counted here separately.

BRAM-width The width of the BRAM indicates memory consumption of the stack. Although the number of bits in the design may vary depending on the choice of the stack depth, the width of memory is static for each transformed algorithm. It is a measurement of the memory consumption.

In the Table 5.3 the synthesis results are compared. The frequency of the factorial is significantly slower than the other algorithms. This can be explained due to the multiplier, which generates a much larger propagation delay compared to the other algorithms which only uses equality tests and additions. The number of LEs are however greater in the Fibonacci and the Ackermann function. In the Fibonacci function, more continuations are introduced which leads to more control and logic and thus LEs. In the Ackermann however, the same number of continuations are present compared to the factorial function. However the Ackermann function has two input arguments of type *Unsigned32* and has more case data patterns then the factorial which leads to more LEs.

The FactorialTail and the GCD do not make use of the stack architecture, because these functions are both tail recursive. The GCD has a very slow frequency compared to the other transformed algorithms. This is the result of the use of the mod primitive. It causes a large

Algorithm	f_{max} in MHz		ALMs	
	CλaSH	[40]	CλaSH	[40]
Fibonacci	347.83	318	289	131
Ackermann	354.74	325	208	162

Table 5.4 – Comparison of the synthesis results between results produced by the CλaSH compiler and [40]

propagation delay. The implementation of the GCD also uses much LEs, which again can be linked to the use of this mod primitive.

5.2.1 Comparison with Edwards et al.

Another set of synthesis results is used to compare the work described in this thesis with the work of [40]. In this case, the transformed algorithms are synthesized for a different FPGA: the Stratix V `5SGXEA7H3F35C3` which is a more high-end FPGA.

In Table 5.4 a comparison is made between the synthesis results of both methods. The derived stack architectures of the Fibonacci and the Ackermann function are used in this comparison. In [40] the results are different for specific arguments; for example Fib (25) and Fib (30) differ in maximum frequency and Adaptive Logic Modules (ALMs), which are the modules that contain the LUTs and registers in the Stratix V FPGA. The stack architectures in this thesis are only synthesized for 32-bits unsigned integers, and produce the same circuitry for each argument. The results in [40], which produce the best results in terms of maximum frequency and ALMs, are compared with the results of the synthesis of the work presented in this thesis.

The synthesis of the stack architectures produced by the methods described in this thesis create obtain a higher frequency than [40]. However, more ALMS are introduced. The cause of this difference is hard to determine; different tooling (Altera Quartus 14.0.0 in [40] versus Altera Quartus 15.0.0 used in this thesis), different design choices, and CλaSH synthesis choices may all contribute in this difference.

The number of clock cycles it takes, for a computation to finish, is also compared with results in [40] and listed in Table 5.5. A simulation in the interactive CλaSH environment is used to obtain these results. As can be seen in the table, the number of clock-cycles, obtained by applying the methods described in this thesis, is structurally less then the results present in [40]. The precise cause of difference is again hard to determine as the procedure described in [40] skips implementation details. A probable cause of this difference may be the choice of architecture. The architecture presented in this thesis supports the *Repl* and *Nop* instructions. These instructions can sometimes be used in stead of successively executing a *Push* and *Pop*. If this replacement can occur, it saves an extra clock cycle.

Algorithm	Clock-cycles ($\times 10^3$)	
	CλaSH	[40]
Fib(20)	27	43
Fib(25)	300	486
Fib(30)	3328	5385
Ack(3,6)	258	344
Ack(3,7)	1040	1387
Ack(3,8)	4118	5571

Table 5.5 – Comparison of the number of clock cycles before a algorithm finishes. The methodology described in this thesis is compared to the results described in [40]

6 Conclusions and Recommendations

This chapter elaborates the findings of this thesis. First the findings of each of the chapter is discussed shortly. Then, an answer of the research question of this thesis is provided. Finally, recommendations are presented in the form of future work.

In Chapter 2, background and related work is elaborated. First CλaSH is introduced: an introduction to the CλaSH language is given, the global workings of the CλaSH compiler is elaborated, and the limitation of the current support of recursion in the CλaSH compiler is identified. Several properties of recursive functions are also highlighted in that chapter, which enabled us to identify the characteristics of these functions in the rest of the thesis. Relevant literature of the usage of recursion in reconfigurable hardware is discussed. Additionally, related work describing FHDL compilers is investigated. Finally, the CPS concept is elaborated, as it is an important concept in this thesis.

In Chapter 3, a methodology is developed, based on the literature and findings in the background chapter. First an abstract syntax is developed. Based on this syntax, formal rewrite rules are presented that implement the sketched rewrite rules of Edwards et al. [40]. These formal rewrite rules are inspired by the rewrite rules defined by Danvy et al. [12] that describe a generic CPS transform. This leads, eventually, to the derivation of a stack architecture.

In Chapter 4, implementation details of the rewrite rules, and the stack architecture are presented. The stack architecture is described in the CλaSH language, which enables the implementation of derived stack architectures in an FPGA. Several design aspects of implementing this stack architecture are covered, such as the usage of the BRAM.

In Chapter 5 results of the use of the developed methodology combined with the implementation techniques is assessed. This assessment is performed by deriving stack architecture descriptions of more example algorithms. The derived stack architectures descriptions are then synthesised for specified FPGAs. The results of the syntheses are elaborated and some results are compared to [40].

The presented work can be summarized in the form of answering the research question. As first posed in Chapter 1 section §1.1, this research question is:

» *How can data-dependent recursive function definitions be supported by the CλaSH compiler?*

In this thesis, the research question is answered by a presented methodology that derives hardware capable of handling data-dependent recursive functions. Research of Edwards et al. combined with other work, is used to develop formal rewrite rules for a simply typed lambda calculus. These rewrite rules transform the recursive function definition in a CPS form that can be executed on a stack architecture. The implementation details of creating such architectures is also elaborated in this thesis. Finally, several data-dependent recursive functions are transformed using the presented methodology and results are compared.

6.1 Recommendations and Future Work

Although the presented methodology is implemented in a proof of concept, which produces CλaSH circuit descriptions, there are certain aspects of this research that still need to be researched further. These aspects are presented in this section in the form of recommendations and future work. Before an actual implementation of the method should take place, one has to consider the following aspects.

6.1.1 Transforming more involved recursive functions

A set of simple theoretical recursive functions is transformed in this thesis in order to asses the correctness of the transformations. Future research may use the methodology presented in this thesis to transform other, more, involved recursive functions, such as: Divide and conquer algorithms, graph algorithms, etcetera. This will provide more insight into usability and practicality of this work.

6.1.2 Mutual recursive functions

In section §2.2.4 the concept of mutual recursion is elaborated. The presented rewrite rules in this thesis, do, however, not allow this form of indirect recursion — only direct recursive calls are allowed. As to future work, one may investigate the possibilities to enable this form of recursion. Edwards et al. [40] propose a solution for this which merges the dependent functions into one function, before transforming it.

Another possible solution may be to simply allow more than one function call F_n. If each function involved in the mutual recursive function corresponds to an unique function call, and each continuation is unique over all transformed function, mutual recursion can occur between the different functions. This should also enable mutual recursion, however this has to be researched.

6.1.3 Higher order functions

As mentioned §3.1.2, the abstract syntax used in this thesis, does not allow higher-order functions. The main reason for this limitation is because it simplified the analysis of the syntax. This restriction is however not desired when implementing the support of data-dependent recursion in CλaSH. It does support (some cases of) higher-order functions.

It should be possible to enable the support the use of higher-order functions 'between' the recursive function calls, for example, an operation mapped onto a vector may be defined as operation 'between' the recursive calls. Further research may provide answers to which extend higher-order functions can be combined in recursive function definitions.

6.1.4 Stack architecture

Although a stack architecture is proposed in chapter Chapter 4, a lot of variations can be made in the chosen stack architecture.

As mentioned in section §2.3.1, Sklyarov et al. [24, 32, 33] propose a methodology for manually implementing recursive algorithms in a HFSM. It would be interesting to determine if the rewrite rules can be altered to enable the derivation of such a HFSM. Future work may extend the rewrite rules and stack architecture to automatically generate such architectures.

The suggested stack architecture in this thesis may also be improved. For example: the stack frame currently holds all free variables. However, this can be reduced to only the free variables that are needed in the rest of the continuations. This will reduce the stack frame size in some cases. One may also detect if the stack is used at all. If the function is tail-recursive — as is the case in equation (2.6) — the continuation stack and control mechanisms can be removed, as it is not needed.

6.1.5 Space-time trade-offs

The generated hardware from our method, currently adds delays between each recursive call. All functions called between the recursive calls, are required to be combinational circuits. It may be interesting to investigate the effects of inlining the recursive calls, and create separate calls for these inlined functions. This may also work in the opposite direction, one can split up the function in multiple stages. This may involve marking more serious applications and handling the types correctly, but further research can investigate such trade-offs.

6.1.6 Interfacing surrounding hardware

The interface with the stack architecture is not yet investigated. One can choose for example to integrate the stack architecture within the data-flow support from CλaSH. Data-flow support in CλaSH has bidirectional synchronisation channels. One for asserting the validity of the data and the other for asserting circuit readiness. Because in data-dependent recursion, it is generally unknown how long the computations will take. One can synchronise using the validity channel when the computation is finished.

A

Abstract Syntax and Rewrite Rules

A.1 Abstract Syntax

```haskell
{-
Module : Expr
Description : Abstract Syntax
Copyright : (c) University of Twente 2015
License : BSD2
Maintainer : i.teraa@student.utwente.nl
Stability : experimental
-}
module Expr where

import Data.List (nub,intersperse,nubBy, intercalate)
import Control.Arrow (second)
import Text.PrettyPrint.HughesPJClass

data Expr r = Var String
            | App r (Expr r) (Expr r)
            | Lam Binder (Expr r)
            | Let Binder (Expr r) (Expr r)
            | Case (Expr r) [(AltCon, Expr r)]
            | Lit Int
  deriving (Show)

data AltCon = DefaultAlt Binder
            | LitAlt Int
            | DataAlt String [Binder]
  deriving (Show)

```

```
data Reynold = Trivial
          | Serious
  deriving (Show)

type ReynoldExpr = Expr Reynold
type CExpr = Expr ()

data Type = TyCon TyConId
        | TyVar TyVarId
        | TyApp Type Type
  deriving (Show)

type TyConId = String
type TyVarId = String
type Var = String
type Binder = (Var, Type)

data DataDef = DataDef String [(String,[Type])] deriving (Show)
data FunDef e = FunDef String [Type] Type e deriving (Show)

data Program = Program [DataDef] [FunDef CExpr] deriving (Show)

--------------------------------------------------------------------------------
-- pPrint instances
--------------------------------------------------------------------------------
instance (Pretty a) => Pretty (Expr a) where
  pPrint (Var x) = text x
  pPrint (Lit i) = int i
  pPrint (App r e1 e2) = (parens $ pPrint e1) <+> (parens $ pPrint e2)
  pPrint (Lam (v,_) e) = text "\\" <> text v ->> pPrint e
  pPrint (Let (s,_) e1 e2) = text "let" <+> text s <=> pPrint e1
                         <+> text "in" <+> pPrint e2
  pPrint (Case e alt) = text "case" <+> pPrint e <+> text "of"
                         <+> nest 2 (vcat $ map f alt) where
    f (ac, e) = pPrint ac ->> pPrint e

instance Pretty AltCon where
  pPrint (DefaultAlt (s,t)) = text s
  pPrint (LitAlt i) = int i
  pPrint (DataAlt s bs) = text s <+> (hsep $ map (text.fst) bs)

instance Pretty DataDef where
  pPrint (DataDef s cs) = text "data" <+> text s <+> cs' where
    cs' = hsep (punctuate (text "␣|") $ map f cs)
    f (s, dts) = text s <+> hsep (map pPrint dts)

instance (Pretty a) => Pretty (FunDef a) where
  pPrint (FunDef s tyArgs tRet e) = text s <+> text "::" <+> ty
                              $$ text s <=> pPrint e where
    ty = hsep $ punctuate (text "->")
                                $ map pPrint (tyArgs ++ [tRet])

instance Pretty Program where
  pPrint (Program ddef vdef) = vcat $ (map pPrint ddef) ++ (map pPrint vdef)

instance Pretty Type where
  pPrint (TyCon id) = text id
  pPrint (TyVar var) = text var
  pPrint (TyApp t1 t2) = pPrint t1 <+> pPrint t2
```

```haskell

--------------------------------------------------------------------------------
-- Helpers
--------------------------------------------------------------------------------
-- pPrint helpers
(<=>) :: Doc -> Doc -> Doc
a <=> b = a <+> text "=" <+> b

(->>) :: Doc -> Doc -> Doc
a ->> b = a <+> text "->" <+> b

-- make tuple
mkTuple e1 = App () (App () (Var "(,)") e1)

-- Create unique supply of binders
uniqueSupply :: String -> Type -> [Binder]
uniqueSupply s ty = zip (map ((s++).show) [0..]) (repeat ty)

-- Fetch free variables given a context
freeVars :: [Binder] -> CExpr -> [Binder]
freeVars bndrs expr = case expr of
  (Var v) -> lookupBinder v bndrs
  (Lit _) -> []
  (Lam b e) -> freeVars bndrs e
  (App () e1 e2) -> freeVars bndrs e1 ++ freeVars bndrs e2
  (Let b e1 e2) -> nubBy (\(a,_) (b,_)->a==b ) $(freeVars bndrs e1) ++ freeVars bndrs e2
  (Case e alts) -> concatMap (\(_,e')->freeVars bndrs e') alts

lookupBinder :: String -> [Binder] -> [Binder]
lookupBinder v bndrs = let fs = filter (\(id,ty)->id==v) bndrs in
                       if null fs then [] else [head fs]

-- Replace Variable in Expression
replaceVar :: Var -> Var -> CExpr -> CExpr
replaceVar v v' (Var s) | v == s = Var v'
                        | otherwise = Var s
replaceVar _ _ e@(Lit _) = e
replaceVar v v' (Lam v1 e) = Lam v1 (replaceVar v v' e)
replaceVar v v' (App () e1 e2) = App () e1' e2'
  where
    e1' = replaceVar v v' e1
    e2' = replaceVar v v' e2
replaceVar v v' (Case e alts) = Case (replaceVar v v' e) alts'
  where
    alts' = map (second (replaceVar v v')) alts
```

Listing A.1 – Expr.hs

A.2 Rewrite Rules

A.2.1 Naming

```haskell
{-
Module : Naming
Description : Naming rewrite rule
Copyright : (c) University of Twente 2015
```

```haskell
License : BSD2
Maintainer : i.teraa@student.utwente.nl
Stability : experimental
-}
module Naming (naming) where

import Expr
import Data.Traversable (mapAccumL)

-- | Renaming step of the CPS (continuation passing style) transform
-- for more information see [1]
naming :: FunDef ReynoldExpr -- ^ Original function definition
       -> FunDef CExpr -- ^ Rewritten function definition
naming (FunDef f argTy retTy e) = FunDef f argTy retTy e'
  where (e',_) = naming' (uniqueSupply "v" $ retTy) e

-- | Acutal naming rewrite rule, introduce let expression at Serious
-- applications.
naming' :: [Binder] -- ^ Unique supply for naming
        -> ReynoldExpr -- ^ Expressions with annotated
        -> (CExpr, [Binder]) -- ^ Tuple with transformed expression and rest of
                             -- unique names
naming' bs expr = case expr of
  Var x -> (Var x, bs)
  Lit i -> (Lit i, bs)
  Lam x e -> (Lam x e', bs')
    where (e',bs') = naming' bs e
  App Serious e1 e2 -> (Let b (App () e1' e2') (Var x), bs'')
    where
      (b@(x,_):bss) = bs
      (e1', bs') = naming' bss e1
      (e2', bs'') = naming' bs' e2
  App Trivial e1 e2 -> (App () e1' e2', bs'')
    where
      (e1', bs') = naming' bs e1
      (e2', bs'') = naming' bs' e2
  Case e alts -> (Case e' alts', bs'')
    where
      (e',bs') = naming' bs e
      (bs'', alts') = mapAccumL f bs' alts
      f acc (dc, e) = let (e'',acc' ) = naming' acc e in (acc, (dc, e''))
```

Listing A.2 – Naming.hs

A.2.2 Sequentialize

```haskell
{-
Module : Sequentialize
Description : Sequentialize rewrite step.
Copyright : (c) University of Twente 2015
License : BSD2
Maintainer : i.teraa@student.utwente.nl
Stability : experimental
-}
module Sequentialize (sequentialize) where

import Expr
```

```haskell
import Data.Traversable (mapAccumL)

-- | Sequentialize step of the CPS transform
sequentialize :: FunDef CExpr -> FunDef CExpr
sequentialize (FunDef f argTy retTy e) =
 let (e',_) = sequentialize' e in FunDef f argTy retTy e'

-- | Actual rewrite rule
sequentialize' :: CExpr -- ^ Input Expression
   -> (CExpr, [(Binder, CExpr)]) -- ^ Tuple of output Expression
sequentialize' (Var x) = (Var x, [])
sequentialize' (Lit x) = (Lit x, [])
sequentialize' (Lam b e) = (Lam b (lets $ sequentialize' e), [])
sequentialize' (Let b e1 e2) = (e2', nu1 ++ [(b,e1')] ++ nu2)
 where
   (e1', nu1) = sequentialize' e1
   (e2', nu2) = sequentialize' e2
sequentialize' (App () e1 e2) = (App () e1' e2', nu1 ++ nu2)
 where
   (e1', nu1) = sequentialize' e1
   (e2', nu2) = sequentialize' e2
sequentialize' (Case es alts) = (Case es' alts', nus)
 where
   (es', nus) = sequentialize' es
   alts' = map (fmap (lets . sequentialize')) alts

-- | Helper function for sequentialize' rewerite step
lets :: (CExpr, [(Binder, CExpr)]) -> CExpr
lets (e, nus) = foldr (\(b,e1) e2 -> Let b e1 e2) e nus
```

Listing A.3 – Sequentialize.hs

A.2.3 Generate stack architecture

```haskell
{-
Module : GenStackArch
Description : Generate Stack Arch module
Copyright : (c) University of Twente 2015
License : BSD2
Maintainer : i.teraa@student.utwente.nl
Stability : experimental
-}
module GenStackArch (stackArchGen) where

import Expr

import Data.List (nub, intersperse, deleteBy)
import Data.Traversable (mapAccumL)
import Data.Maybe (isNothing)

----------------------------------------------------------------------------
-- Stack Arch Introduction
----------------------------------------------------------------------------

-- | Generate stack architecture given a FunDef.
stackArchGen :: FunDef CExpr -- ^ A function that needs to be transformed
         -> Program -- ^ Resulting program
```

```
stackArchGen (FunDef f argTy retTy e) = Program [cont, call] [next]
 where
  cont = DataDef "Cont" (map (conDa2Ty.fst) (a:as))
  call = DataDef "Call" [("F",argTy),("R", [retTy])]
  next = FunDef "next" [(TyCon "(Call,Cont)")] (TyCon "(Cont,STDCmd)")
               (Case (Var "(c,k)") [
                 (DataAlt "F" bs, e''),
                 (DataAlt "R" [("r", retTy)], Case (Var "k") (a:as))])
  (bs, e') = firstLams [] e
  (e'', as) = deriveNext f (ks) bs True e'
  a = (DataAlt k [],
               mkTuple (App () (Var "R") (Var "r")) (Var "Done"))
  (k:ks) = map fst $ uniqueSupply "K" retTy

-- | Derive Next Function is used to collect an expression and continuations
-- for a description of the next function. The rewrite rule perform two tasks:
-- * All results of next the function must be in the form of a tuple containing
-- a call and stack instruction.
-- * Continuations are collected in the form of a data pattern for a case
-- expression which handles the continuations.
deriveNext :: Var -- ^ Function name
      -> [Var] -- ^ Unique supply
      -> [Binder] -- ^ Binders in context
      -> Bool -- ^ First continuation indicator
      -> CExpr -- ^ Rewrite expression
      -> (CExpr, [(AltCon,CExpr)]) -- ^ alternated expression and
                                   -- continuation case patterns.
deriveNext f (k:ks) bs phi expr = case expr of
 Var x -> (deriveNextR phi (Var x), [])
 Lit i -> (deriveNextR phi (Lit i), [])
 App _ e1 e2 -> (deriveNextR phi (App () e1 e2), [])
 (Let v e1 (Var v')) | v*=*v' -> (deriveNextR phi e1, [])
 Let b@(x,t) e1 e2 -> (deriveNextK phi e1 k', as')
  where
   (ce, as) = deriveNext f ks bs' False e2
   as' = ((DataAlt k fvs, replaceVar x "r" ce):as)
   fvs = freeVars bs expr
   k' = applyVars (Var k) fvs
   bs' = addBinder b bs
 Lam b e -> (Lam b e', as)
  where
   (e', as) = deriveNext f (k:ks) bs' phi e
   bs' = addBinder b bs
 Case es alts -> (Case es alts', as)
  where
   alts' = zip (map fst alts) (map fst xss)
   as = concatMap snd xss
   (_,xss) = mapAccumL fun (k:ks) (map snd alts)
   fun is e = (drop (length xs) is, (e',xs) )
    where
     (e', xs) = deriveNext f is bs phi e

-- | derive next helper for pushing and replacing continuations
deriveNextK :: Bool -- ^ First continuation indicator.
       -> CExpr -- ^ Original expression.
       -> CExpr -- ^ Continuation
       -> CExpr -- ^ result expression
deriveNextK True e k = mkTuple e (App () (Var "Push") k)
deriveNextK False e k = mkTuple e (App () (Var "Repl") k)
```

```

-- | Derive Next helper function for results
deriveNextR :: Bool -- ^ First continuation indicator
         -> CExpr -- ^ Original expression
         -> CExpr -- ^ Resulting expression
deriveNextR True e = mkTuple e (Var "Nop")
deriveNextR False e = mkTuple e (Var "Pop")

-- | Derive next helper for changing the result expression to Function
-- construct or Result construct
-- TODO: Add Return R case
deriveNextC :: Var -- ^ Function Name
         -> CExpr -- ^ Orignial expression
         -> CExpr -- ^ Alternated expression
deriveNextC f e = replaceVar f "F" e

--------------------------------------------------------------------------------
-- helpers
--------------------------------------------------------------------------------

conDa2Ty :: AltCon -> (String, [Type])
conDa2Ty (DataAlt id bs) = (id, map snd bs) where

firstLams :: [Binder] -> CExpr -> ([Binder], CExpr)
firstLams xs (Lam x e) = firstLams (x:xs) e
firstLams xs e = (reverse xs, e)

(*=*) :: Binder -> Var -> Bool
(*=*) b v = fst b == v

applyVars :: CExpr -> [Binder] -> CExpr
applyVars = foldl (\e b -> App () e (Var (fst b)))

addBinder :: Binder -> [Binder] -> [Binder]
addBinder b bs = b:bs' where
  bs' = deleteBy f b bs
  f (id,_) (id',_) = id==id'
```

Listing A.4 – GenStackArch.hs

A.2.4 Transform

```
{-
Module : Transform
Copyright : (c) University of Twente 2015
License : BSD2
Maintainer : i.teraa@student.utwente.nl
Stability : experimental

-}
module Transform (transform) where

import Expr
import Naming
import Sequentialize
import GenStackArch

```

```
transform :: FunDef ReynoldExpr -> Program
transform = (stackArchGen . sequentialize . naming)
```

Listing A.5 – Transform.hs

B

CλaSH Stack Architecture

```haskell
{-
Module : StackArch
Description : Stack Arch Description.
Copyright : (c) University of Twente 2015
License : BSD2
Maintainer : i.teraa@student.utwente.nl
Stability : experimental
-}
module StackArch where

import CLaSH.Prelude
import qualified Data.List as L
import Data.Maybe (catMaybes)
import Debug.Trace
import qualified Control.Exception.Base as E

-- Select description here
import Fibonacci
-- import Factorial
-- import Ackermann

type MemAddr = Unsigned 8
type StackArchState = (Call, MemAddr, Cont)
-- | the stack arch function ties the mealy description of stackUpdate together
-- with an instantiation of a block ram
stackArch :: Signal (Maybe ResultType) -- ^ currently unused.
       -> Signal (Maybe ResultType) -- ^ Resulting unsigned, when done.
stackArch _ = r
  where
    initialState = (start, 0, K0)
    (kappa, p, w, r) = unbundle $ mealy stackUpdate initialState ramKappa
    p_safe = assert "stack_overflow" (p .<=. 1000) (pure True) p
    ramKappa = blockRam (replicate d1000 K0) p_safe p_safe w kappa

```

```haskell
-- | Mealy description of stack update mechanism.
stackUpdate :: StackArchState -- ^ State of stack architecture
         -> Cont -- ^ blockRam read continuation
         -> (StackArchState, (Cont, MemAddr, Bool, Maybe ResultType))
stackUpdate (c,p,kappa) ramKappa = ((c',p',kappa'), (kappa,p',w,r))
  where
    (c', gamma) = next (c,kappa) -- next descriptions are externally defined.
    (kappa', p', w, r) = case gamma of
      Push newKappa -> (newKappa, p+1, True , Nothing)
      Pop -> (ramKappa, p-1, False, Nothing)
      Repl newKappa -> (newKappa, p , False, Nothing)
      Nop -> (kappa, p , False, Nothing)
      Done r -> (kappa, p , False, Just r )

topEntity = stackArch

sim = (L.head . catMaybes . sample . stackArch ) $ signal (Nothing)
```

Listing B.1 – StackArch.hs

Bibliography

[1] Ieee standard vhdl language reference manual. *ANSI/IEEE Std 1076-1993*, 1994.

[2] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[3] Industry Solutions Overview. `https://www.altera.com/solutions/industry.html`, November 2015.

[4] Transistor count. `https://en.wikipedia.org/wiki/Transistor_count`, November 2015.

[5] C. Baaij. CλaSH.Tutorial. `https://hackage.haskell.org/package/clash-prelude-0.10.3/docs/CLaSH-Tutorial.html`, 2014-2015. [Online; accessed November-2015].

[6] C. P. R. Baaij. *Digital circuits in CλaSH: functional specifications and type-directed synthesis*. PhD thesis, Enschede, January 2015.

[7] C. P. R. Baaij, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, Lille, France*, pages 714–721, USA, September 2010. IEEE Computer Society.

[8] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM.

[9] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM.

[10] J. Bos. Synthesizable Specification of a VLIW Processor in the Functional Hardware Description Language CλaSH, September 2014.

[11] M. DAM. Auditory processing using CλaSH, June 2015.

[12] O. Danvy. Three steps for the cps transformation. 1991.

[13] L. Gamut. *Logic, Language, and Meaning: Intensional logic and logical grammar*, volume 2. University of Chicago Press, 1991.

[14] M. E. T. Gerards, C. P. R. Baaij, J. Kuper, and M. Kooijman. Higher-Order Abstraction in Hardware Descriptions with CλaSH. In P. Kitsos, editor, *Proceedings of the 14th EUROMICRO Conference on Digital System Design, DSD 2011, Oulu, Finland*, pages 495–502, USA, August 2011. IEEE Computer Society.

[15] D. R. Ghica. Geometry of Synthesis: A Structured Approach to VLSI Design. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 363–375, New York, NY, USA, 2007. ACM.

[16] D. R. Ghica and A. Smith. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits . *Electronic Notes in Theoretical Computer Science*, 265(0):301 – 324, 2010. Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (MFPS 2010).

[17] D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource Management Through Type Inference. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 345–356, New York, NY, USA, 2011. ACM.

[18] D. R. Ghica, A. Smith, and S. Singh. Geometry of Synthesis IV: Compiling Affine Recursion into Static Hardware. *SIGPLAN Not.*, 46(9):221–233, September 2011.

[19] A. Gill. Type-safe observable sharing in haskell. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 117–128, New York, NY, USA, 2009. ACM.

[20] R. Harmsen. Specifying the WaveCore in CλaSH. `http://essay.utwente.nl/66896/`, March 2015.

[21] X. Jin. Implementation of the MUSIC Algorithm in CλaSH, June 2014.

[22] J. Kuper and R. Wester. N queens on an FPGA: mathematics, programming, or both? In *Communicating Processes Architectures 2014*, pages 181–203. Open Channel Publishing Ltd, 2014.

[23] R. Loader. Notes on simply typed lambda calculus. Technical report, 1998.

[24] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson. Hardware implementation of recursive algorithms. In *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pages 225–228. IEEE, 2010.

[25] A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Automata, Languages and Programming*, pages 37–48. Springer, 2000.

[26] A. Niedermeier. *A fine-grained parallel dataflow-inspired architecture for streaming applications*. PhD thesis, Enschede, August 2014.

[27] A. Niedermeier, R. Wester, K. Rovers, C. Baaij, J. Kuper, and G. Smit. Designing a dataflow processor using CλaSH. In *Proceedings of the 28th Norchip Conference, NORCHIP 2010*, page 69. IEEE Circuits & Systems Society, November 2010.

[28] J. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6(3-4):233–247, 1993.

[29] J. C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[30] M. Santarini. Issue 86: Xilinx Ships Industry's First 20-nm All Programmable Devices. `http://www.xilinx.com/publications/archives/xcell/Xcell86.pdf`, 2014.

[31] I. Skliarova and V. Sklyarov. Recursion in reconfigurable computing: A survey of implementation approaches. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 224–229, Aug 2009.

[32] V. Sklyarov. Hierarchical finite-state machines and their use for digital control. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(2):222–228, June 1999.

[33] V. Sklyarov. FPGA-based implementation of recursive algorithms . *Microprocessors and Microsystems*, 28(5–6):197 – 211, 2004. Special Issue on FPGAs: Applications and Designs.

[34] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[35] T. G. Team. The GHC Compiler. `http://haskell.org/ghc`, November 2015.

[36] F. Van Nee. To a new hardware design methodology: A case study of the cochlea model. `http://essay.utwente.nl/64835/`, March 2014.

[37] R. Wester. *A transformation-based approach to hardware design using higher-order functions*. PhD thesis, Enschede, July 2015.

[38] R. Wester and J. Kuper. Design space exploration of a particle filter using higher-order functions. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture notes in computer science*, pages 219–226. Springer, London, UK, 2014.

[39] R. Wester, D. Sarakiotis, E. Kooistra, and J. Kuper. Specification of APERTIF Polyphase Filter Bank in CλaSH. In *Communicating Process Architectures 2012*, pages 53–64. Open Channel Publishing, August 2012.

[40] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. Hardware synthesis from a recursive functional language. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*, pages 83–93. IEEE Press, 2015.

Acronyms

CλaSH CAES language for asynchronous hardware

CPS Continuation Passing Style

IR Intermediate Representation

HDL Hardware Description Language

HFSM Hierarchical Finite-State Machine

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

GHC Glasgow Haskell Compiler

LUT Look Up Table

VLIW Very Long Instruction Word

MUSIC MUltiple SIgnal Classification

SAFL Statically Allocated parallel Functional Language

DSL Domain Specific Language

FPGA Field Programmable Gate Array

MAC Multiply ACcumulate

FSM Finite-State Machine

RAM Random Access Memory

BRAM Block RAM

CPU Central Processing Unit

GPU Graphics Processing Unit

FHDL Functional HDL

RTL Register Transfer Level

ALM Adaptive Logic Module

LE Logic Element