Hardware Synthesis in ForSyDe

HENDRIK WOIDT
Abstract

There have been numerous efforts in the development of functional hardware description languages over the past years. In this thesis project the design space for embedded domain specific languages for hardware synthesis in Haskell is explored by comparing the approaches of two different language implementations.

This report contains an introduction to the fundamental concepts for modeling hardware in a functional style and the core concepts of implementing deep embedded languages. Based on this, the architectures of ForSyDe.Deep and Cλash are examined in order to find their strengths and weaknesses. The results are applied to the implementation of translation of data-parallel higher order functions in ForSyDe.Deep.

The implementation of higher order functions has shown that the lack of type information available for the translation of process functions in the current implementation of ForSyDe.Deep is the limiting factor for achieving a higher level of abstraction within process functions through polymorphism or higher order functions. This does not diminish the approach of ForSyDe though as the real power lies in the abstraction provided by the process network.
# Contents

1 Introduction ........................................ 6
   1.1 Problem Statement ................................. 7
   1.2 Overview ......................................... 7
   1.3 Task ............................................... 8

2 Background ........................................... 10
   2.1 Domain Specific Languages ....................... 10
   2.2 Embedded Domain Specific Languages ........... 13
   2.3 Domain Specific Languages in Haskell .......... 15
      2.3.1 Example: A simple deep embedded language 15
      2.3.2 Template Haskell ............................. 20
      2.3.3 Compiler API ................................. 21
   2.4 Compiling Functional Languages .................. 22
      2.4.1 Transformation and Translation .............. 22
      2.4.2 Defunctionalization .......................... 23
      2.4.3 Specialization ............................... 23
   2.5 The ForSyDe Methodology .......................... 24
      2.5.1 Introduction ................................ 24
      2.5.2 Process Networks ............................. 24
      2.5.3 Models of Computation ....................... 25
      2.5.4 Available Implementations ................... 26
   2.6 Expressing Parallelism ............................ 27
      2.6.1 Skeletons and Higher Order Functions ....... 28
      2.6.2 Forms of Data-Parallelism ................... 29
      2.6.3 Raising the Abstraction on Mapping ......... 31

3 Functional Hardware Description Languages .......... 33
   3.1 The ForSyDe.Deep Language ...................... 33
      3.1.1 Language Example ........................... 33
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADT</td>
<td>Algebraic Data Type</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CT</td>
<td>Continuous Time</td>
</tr>
<tr>
<td>DDG</td>
<td>Data Dependency Graph</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>EDSL</td>
<td>embedded Domain Specific Language</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FLI</td>
<td>Foreign Language Interface</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>ForSyDe</td>
<td>Formal System Design</td>
</tr>
<tr>
<td>GHC</td>
<td>Glasgow Haskell Compiler</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose GPU</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>INF</td>
<td>Intended Normal Form</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>KTH</td>
<td>Royal Institute of Technology</td>
</tr>
<tr>
<td>MoC</td>
<td>Model of Computation</td>
</tr>
<tr>
<td>QML</td>
<td>Qt Markup Language</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>STG</td>
<td>Spineless-Tagless-G</td>
</tr>
<tr>
<td>SYB</td>
<td>Scrap Your Boilerplate</td>
</tr>
<tr>
<td>TH-AST</td>
<td>Template Haskell Abstract Syntax Tree</td>
</tr>
<tr>
<td>TH</td>
<td>Template Haskell</td>
</tr>
<tr>
<td>VHDL-IP</td>
<td>VHDL Intellectual Property</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHPI</td>
<td>VHDL Procedural Interface</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The development of modern technology leads to an ever growing need for systems which satisfy the demand for smaller, faster and more power efficient computing. In modern cars, this is realized by deploying hundreds of embedded processing units controlling a multitude of environmental functions.

Consequentially, each new generation of cyber-physical and embedded systems solves tougher problems, is more powerful, more intertwined with it’s environment and therefore much more complex than it’s predecessor. Countering this complexity an approximately constant workforce of engineers tries to cope with the ever increasing difficulty of assuring correctness of these systems. Verification costs are rising sharply, taking up the largest part of the budget in most projects.

One approach to tackling these problems is to significantly increase the level of abstraction at which the systems can be designed and verified. The first executable high level model of an application is used to capture the functional requirements without needing to deal with lower level implementation details. Through a process of subsequential refinement design and implementation decisions are applied, which narrow down the set of possible solutions while trying to ensure functional correctness, keeping the design constraints and still optimizing performance and cost.

The implementation of a suitably abstract model is difficult without appropriate tools. Domain specific languages can be defined to describe such a model in an analyzable way to facilitate the automation of transformations. The implementation and translation of this kind of system design language provides the background for the work presented in this thesis.

One approach to the design of embedded systems using domain specific languages is the Formal System Design (ForSyDe) methodology [24]. It provides a formal framework for the definition of high level system models and is comprised of a suite of tool
flows for describing and transforming system models according to its theoretical principles. One particular tool flow is the ForSyDe.Deep implementation which provides a language for describing networks of concurrent processes and synthesizing digital circuits from them [2]. It is embedded into the general purpose functional programming language Haskell [30].

There have been additional efforts in the area of hardware synthesis from functional languages. In the context of Haskell as an implementation platform, the most notable are the Lava family of languages namely Lava [7], Chalmers Lava [9], Xilinx Lava [33] and Kansas Lava [13] as well as the Cλash language [4] [18].

1.1 Problem Statement

The languages given above follow different fundamental approaches for the implementation of domain specific languages in Haskell. This thesis presents an overview and analysis of the different approaches and specifically compares ForSyDe.Deep and Cλash. Building on these findings, ForSyDe.Deep is extended to improve the description of data parallel operations within processes.

1.2 Overview

The thesis is divided into five chapters. The current chapter gives a brief introduction to the problem area and clearly defines the goals of the thesis project.

The second chapter provides the theoretical knowledge obtained and needed for this project. In particular a taxonomy of domain specific languages is given, especially focusing on the aspect of embedding such languages into a general purpose host language. Building on this, a thorough introduction to embedded languages in Haskell is presented. The ForSyDe methodology is introduced in more detail followed by a discussion of the different forms of parallelism found in the process networks modeled in ForSyDe.Deep.

The third chapter analyzes functional hardware description languages. Special focus is laid on the architectural approaches of their implementations. The chapter ends in an in-depth comparison of ForSyDe.Deep and Cλash.

The fourth chapter discusses the implementation work that was performed within the context of this thesis project. It outlines the solution architecture for the implementation of translation of higher order functions in ForSyDe.Deep, followed by the evaluation and discussion of the results obtained.
Chapter 1

1.3 Task

This thesis project explores the design space of functional languages for synthesis and modeling of embedded systems and more specifically digital synchronous circuits. The goal is to explore the future direction of the ForSyDe.Deep project. As such the main focus is placed on Hardware Description Languages (HDLs) implemented in the programming language Haskell, particularly Cλash.

Additionally the feature set of the ForSyDe.Deep implementation is to be extended by adding support for process-internal data-parallelism (see Section 2.6.2 for more details). More specifically, the goal is to implement translation rules for the set of higher order functions on fixed-size-vectors (e.g. mapVector). The implementation is to be compared with the existing process-external parallelism that is given through the zipWithX family of functions.

As a side product of the thesis, maintenance work on the compiler implementation of ForSyDe.Deep is to be conducted, resulting in a proper release on Haskell’s package database HackageDB [14], as well as improved user and developer documentation.

In summary the mandatory goals and result artifacts are:

1. Implementation of process-internal data-parallelism in ForSyDe.Deep.

2. Thesis Report
   - Overview of the problems and design space for synthesizable system modeling languages.
   - In-depth comparison of ForSyDe.Deep and Cλash.
   - Discussion and evaluation of implementation work done within this project.
   - Recommendation about future direction of the ForSyDe.Deep project.

On top of that, the optional results are:

1. New release of the ForSyDe.Deep compiler package on HackageDB supporting the latest Glasgow Haskell Compiler (GHC).

2. Maintenance on ForSyDe.Deep
   - Implementation of continuous integration testing
1.3. TASK

- Porting to newest version of the Haskell compiler
- Bug-fixes as needed
- Clean-up and refactoring
- Improved developer documentation
- Improved user documentation

3. New features for ForSyDe.Deep

- GHDL simulation support
- Xilinx toolchain support
- Extensible Markup Language (XML) backend for interoperability among tools within the ForSyDe framework

Optional results are to be completed as time permits, whereas mandatory results are required for the successful completion of the thesis project.
Chapter 2

Background

This chapter introduces the concepts and knowledge required for the analysis of functional hardware description languages and the subsequent implementation work. The concept of embedded and domain specific languages is introduced thoroughly, also discussing possible implementation approaches of embedding a language in Haskell. Additionally the background and fundamental concepts of the ForSyDe methodology are presented as well as a discussion of different forms of data parallelism found in process networks.

An introduction to the Haskell programming language would be in order, which is out of the scope of this thesis report. Fortunately there is excellent literature available for learning Haskell. The recommended book for the first steps in the language is [20]. The first eight chapters form a good basis for understanding most of the Haskell specifics in this report. Additionally chapter twelve on the abstraction of monads is an important read for understanding some more advanced concepts.

2.1 Domain Specific Languages

There is a vast field of computer languages, most of them being Turing complete which means that they can be used to describe any computable algorithm. Nevertheless there are significant differences in how this is done. Different languages are optimized for different purposes. For example languages like C/C++ are specifically aimed at systems programming, which comprises operating systems and highly optimized software libraries. Java and C# on the other hand are more specialized in application programming i.e. implementing user-facing software.

There are countless examples of applications written in C++ and also an operating system written in Java which shows that there is no clear-cut distinction in the area of
2.1. DOMAIN SPECIFIC LANGUAGES

application. However there is a significant difference in the ease of use for the different tasks. Memory management is of little concern for most Java developers as this is taken care of by the automatic garbage collector. In contrast the developer of an operating system critically depends on being able to control every aspect of this low level behaviour.

The languages mentioned above belong to the class of general purpose languages, which are usable for any software development purpose. Sometimes this generality is not needed though and the syntax and mechanisms of abstraction provided by these languages are not optimal for solving the problem at hand in a clear and concise way. Instead, using or even implementing a domain specific language can be beneficial. This language can be very limited in terms of features and complexity because there is a very narrow field of problems targeted.

One popular example for a category of tasks that lends itself to a Domain Specific Language (DSL) is string parsing. Writing a small parser state machine or dissecting a string using general purpose string manipulation functions is tedious and brittle, opening the door for whole classes of bugs to be introduced. Instead the established way of solving this problem is to use a regular expression engine, that takes a very concise description of the pattern (called the “regex” short for “regular expression”) and provides an optimized implementation for parsing strings according to this pattern.

Figure 2.1 shows a comparison between different ways of implementing a function which parses a string, checking whether it starts with an “a”, followed by arbitrarily many occurrences of the substring “bc”. A regular expression is equivalent to a finite state machine. Implementing such an automaton in C++ is quite verbose and the original intent of the function is not immediately obvious from the code. However, stating the problem as a regular expression, takes only six characters and directly conveys the full intention for everyone familiar with this domain specific language.

In this form of declarative programming, the programmer only specifies the problem and leaves finding a suitable solution to an automated solver engine or a compiler. This is particularly suitable for DSL because the compiler is not artificially constrained by the semantics of the general purpose programming language, while the programmer does not need to care about low level implementation details. Other examples for declarative domain specific languages are Structured Query Language (SQL) for database queries or Qt Markup Language (QML) for the declarative description of graphical user interfaces.

This automatic solving of a declarative problem is especially useful in a domain specific context because only there it is actually feasible to implement such an intel-
bool fsm_match(char *in) {
    const int Err = 0;
    int state = 1;
    for (; *in != 0; in++) {
        switch (state) {
        case 1:
            state = (*in == 'a' ? 2 : Err);
            break;
        case 2:
            state = (*in == 'b' ? 3 : Err);
            break;
        case 3:
            state = (*in == 'c' ? 2 : Err);
            break;
        case Err:
            return false;
        }
    }
    return (state == 2);
}
Listing (2.1) Parser statemachine in C++

Figure 2.1: String parsing in different implementations

When implementing a DSL there are several design options, which can be used to classify them further. Starting from scratch by implementing the complete compiler toolchain, gives the greatest flexibility to the language designer. There are no limits to the syntax and there is complete control over all aspects of the language. However this incurs significant implementation overhead, as there is no existing infrastructure to take advantage of. An established general purpose language provides the compiler as well as integrated development environments, helper tools, software packaging and distribution channels as well as a helpful community already in place. Reusing these established components gives a head start for the implementation of a new DSL. This gives rise to the class of embedded domain specific languages.
2.2 Embedded Domain Specific Languages

An embedded domain specific language lives within another general purpose language. This can be done in several ways but ultimately relies on the compiler and runtime system of the host language. The most important restriction is that it relies on and needs to be compatible with the syntax of the host language. Some host languages provide meta programming facilities that allow to introduce limited variations of new syntax. Examples for these are macros in lisp and the preprocessor in C. Another option, supported by many languages is to overload existing operators with new domain specific functionality. Many languages also support the introduction of new operators, one example being Haskell. This can be used to establish a domain specific subset of the host language with a special interpretation and semantic.

Shallow Embedded Language

Since the embedded Domain Specific Language (EDSL) is a valid subset of its host language, it can be directly executed. In order to further classify this type of language, one can examine the result of the execution. The basic case is that the execution will directly yield the desired result. In case of a linear algebra implementation that overloads the multiplication operator of its matrix class, this would be an object containing the result of said multiplication. Another example is the simulation of a system described within the EDSL. Executing the code would result in simulation and computation of the system response to a set of given inputs. Languages that follow this pattern are called “shallow embedded”.

Deep Embedded Language

Directly executing the program is not desired, when the goal of the language is to optimize, transform or translate it in any way. For these activities, a representation of the program as a data structure, such as an Abstract Syntax Tree (AST), is needed. Language implementations in need of this structure are called “deep embedded” because they perform a deeper analysis of the program, operating on a higher level of abstraction, than their “shallow” counterparts. Obtaining the syntax tree can be done in several ways, each having their own peculiarities.

Stand-Alone Deep Embedded Language

One approach follows directly from the shallow embedded language. Instead of yielding the result of a computation, the language operations return objects that describe
themselves. Reiterating the linear algebra example from above, the multiplication operator would now yield an expression object that keeps the type (multiplication) of operation, as well as a reference to the argument expressions. Execution of such an EDSL program yields a complete tree of all the operations encountered. This form of implementation does not depend on any external compiler and completely lives within the host language. As such it is not tied to any specific implementation of the host language. It is especially important to note that only the operations that are actually encountered during execution are included in the resulting tree. Conditional statements of the host language can not be used to model choice within the EDSL as they are executed during compile time and only decide which operations to include for a given condition. Modeling choice within the EDSL needs specially defined operators or functions that will return a corresponding expression object. The same applies to other language constructs like loops or function definitions. This makes it possible to use the host language as a powerful meta programming language, to programmatically shape the structure of the resulting EDSL description.

**Compiler-Assisted Deep Embedded Language**

A deep EDSL that wants to fully represent the expressivity of the host language needs to take a different route. In order to not depart from the embedded approach, a preexisting compiler for the host language can be employed to extract an intermediate representation. Depending on the amount of information desired, the compilation process can be intercepted at different stages throughout the compilation pipeline.

The first stage is to extract the AST, without any further processing. This only alleviates the need for implementing a special parser but allows to freely reinterpret the program in any way. For syntactically simple languages (e.g. Python) or when only a small subset of the host language intended to be supported, this is a good way. Complex languages like C++ and Haskell, on the other hand, will require a lot of work to implement a reasonably complete interpretation of the AST.

In those cases it is often beneficial to leave more work to the original compiler and intercept the intermediate representation after desugaring. As the name suggest, desugaring removes the so-called syntactic sugar, which is the subset of language constructs that exists purely for convenience and can be expressed through other more basic syntax elements. This will remove redundant language constructs, thus giving a more complete interpretation of the host language for the same amount of work. In Haskell the desugaring process of GHC results in the Core language representation, which is described more closely in Section 2.3.3. This also has the benefit of the first
round of semantic checks which most compilers apply during desugaring. As long as one wants to stay within the semantic framework of the host language, this takes care of most consistency checks on the language level. In Haskell this step already includes the full type-checking and type inference process.

Depending on the purpose of the EDSL, it might also be of use to intercept the program at a later stage of compilation e.g. in the code generation backend. This is beneficial when the DSL targets the same computer architecture as the host language implementation, although the line gets blurry for deciding whether that would be a EDSL or just an extension of the host language.

### 2.3 Domain Specific Languages in Haskell

This section gives an introduction to the tools and approaches for building DSL in Haskell and more specifically GHC.

#### 2.3.1 Example: A simple deep embedded language

This example is meant for giving an overview and intuitive understanding of the structure of types and functions involved in the representation and transformation of languages and programs therein. According to the classification presented in Section 2.2, the language defined herein is a stand-alone deep embedded language. This means, it is fully embedded within the host-language Haskell and does not use any additional features of the host compiler.

The basis for any language implementation is the AST data type for representing the structure of a program. This commonly starts with a type for an expression which contains constructors for representing literal values, variables and operators. An example AST type for a basic language is defined as follows:

```haskell
data Expr = Lit Int | Plus Expr Expr | Mult Expr Expr

deriving Show, Eq
```

This language can represent arbitrarily nested addition and multiplication operations on integer literals. A program is written by directly using the constructors to instantiate the tree structure:

```haskell>
Plus (Mult (Lit 5) (Lit 6)) (Lit 1)
```

```
Plus (Mult (Lit 5) (Lit 6)) (Lit 1)
```
In order to make this an actual embedded domain specific language, functions for constructing values of the `Expr` type can be introduced. By defining add and multiply operator functions and introducing a type class for deriving literal expressions from native Haskell types the language becomes more intuitive:

```haskell
import qualified Prelude as P

class Ex t where
    toExpr :: t -> Expr
instance Ex Int where
    toExpr i = Lit i
instance Ex Expr where
    toExpr e = e

(+) :: (Ex a, Ex b) => a -> b -> Expr
x+y = Plus (toExpr x) (toExpr y)

(*) :: (Ex a, Ex b) => a -> b -> Expr
x*y = Mult (toExpr x) (toExpr y)
```

Note that it is needed to import the Haskell standard library `Prelude` with the `qualified` attribute, since the newly defined operator functions would interfere with the already existing operators from the `Num` class. A better way of implementing this would be to create an instance of `Num` for `Expr`. This is outside of the scope of this example.

The new feature simplifies the syntax as it allows to write natural expressions, which automatically result in the corresponding AST. The AST given above can now be described using a more intuitive syntax:

```haskell
> (5*6)+1
Plus (Mult (Lit 5) (Lit 6)) (Lit 1)
```

So far the language is of limited utility. To actually make this a proper DSL, functions and variables are needed. The first option is to rely on the host language to provide these features. Haskell’s syntax for defining variables and functions can be used to compose subtrees. In the following listing, the `let`-binding in the first statement defines a variable, while the second one defines a function `f`. The last statement is built, using Haskell’s lambda syntax for defining a function (equivalent to `f` from the second statement) and immediately applies it to the given expression.
On the one hand, this is an extremely powerful meta programming approach, because the full power of the host language can be used to create and compose subtrees of the AST. On the other hand, it discards important information about the structure of the program. The subexpression-tree bound to a variable is duplicated for every occurrence of the variable. This can become a serious resource issue when complex subexpressions are reused often.

The alternative is to introduce variables and assignments into the language itself. This can be done by adding the corresponding constructors to the \texttt{Expr} type. Variables are identified by a string which is abstracted through the type alias \texttt{Name}. A \texttt{Let} binding serves the purpose of assigning an expression to a variable name within the given \texttt{Expr} subtree. In order to make variable references as convenient as the use of integer literals, an instance of the \texttt{Ex} class is defined as well. Additionally a convenience function is introduced to bind several variables at once.

```haskell
import Data.Functor

-- The alternative approach to implementing variables

-- Type declaration

data Assignment = Assignment String Expr

-- Data declaration

data Expr = Lit Integer | Var String | Plus Expr Expr | Mult Expr Expr | Let Assignment Expr

-- Instance declaration

instance Show Expr where
  show (Lit x) = "Lit " ++ show x
  show (Var x) = "Var " ++ x
  show (Plus e1 e2) = "Plus " ++ show e1 ++ show e2
  show (Mult e1 e2) = "Mult " ++ show e1 ++ show e2
  show (Let a e) = "Let " ++ a ++ " = " ++ show e

-- Convenience function

letS :: [Assignment] -> Expr -> Expr
letS [] e = e
letS (a:as) e = Let a (letS as e)
```

Using these extensions, the let binding example from above can now be written as
The language defined so far is of limited complexity but already shows some important features of programming languages, namely literals, operators and variables. For extending the expressivity of the language, there are still missing features like additional operators, conditional statements, functions or lambda abstractions and mutually recursive let bindings. These features do not fit into the scope of this example and are left as an exercise to the reader. Some of these features can be found in the definitions of GHC-Core and Clash-Core which are discussed in Section 2.3.3 and Section 3.2.2 respectively.

There are interesting operations to be performed on the syntax tree. First of all, expressions can be evaluated to retrieve the numerical result of applying all operators. This is only possible, if all variable references are substituted for their actual values. As mentioned before, inlining all variable assignments is not a suitable strategy as it would cause duplication of possibly large subtrees. Instead, the values for variables should first be simplified if possible so that the duplicated terms are smaller.

This example uses the powerful generic programming framework Scrap Your Boilerplate (SYB) [19] for applying a substitution function to every node in the tree, traversing it in bottom-up order. The guard function ensures that local variable bindings are not overwritten by the variable currently being substituted.

Before evaluating the AST it is beneficial to simplify it. If the apply function would be used on every binding, nothing would be gained over the earlier meta-programming approach. An expression would be duplicated before being evaluated. Instead the simplification step reduces constantly known expressions as well as applications of plain literals and variables as they do not increase the size of the tree. Note that after
substituting a literal for a variable, the simplification of the subtree is repeated, as there could be new constantly known terms. An interesting aspect of this function, which can be seen below, is the conciseness of pattern matching for defining the substitution rules. Note that this uses the original addition and multiplication operators defined for integers in the *Prelude*. The prefix `P.` is used to access the qualified operators.

```haskell
simplify :: Expr -> Expr
simplify exp = everywhere (mkT reduce) exp
  where
    reduce (Plus (Lit a) (Lit b)) = Lit (a P.+ b)
    reduce (Mult (Lit a) (Lit b)) = Lit (a P.* b)
    reduce Let (name:= val@ (Var _)) exp = apply name val exp
    reduce Let (name:= val@ (Lit _)) exp = simplify (apply name val exp)
    reduce e = e
```

After simplifying, the expression can be evaluated for obtaining its numerical value. The corresponding function is given below. Let bindings are replaced in top-down order, while operators are traversed in bottom-up order. Since there might still be unbound variables, an exception is thrown when such an expression is encountered. The user facing `eval` function first applies a simplification to the expression after which it is actually evaluated using the recursive worker function `doEval`.

```haskell
eval :: Expr -> Int
eval e = doEval (simplify e)

doEval :: Expr -> Int
doEval Lit a = a
doEval Var n = error ("Undefined variable: " ++ (show n))
doEval Plus a b = (doEval a) P.+ (doEval b)
doEval Mult a b = (doEval a) P.* (doEval b)
doEval Let (name:= val) exp = doEval (apply name val exp)
```

In this example the simplification pass is not strictly needed but it provides a good example of generic traversals for transforming a syntax tree without translating it. When extending this language with functions using expression constructors for lambda abstraction and function application, the simplification step could be used to transform function application into let bindings. This would alleviate the `eval` function from needing to handle the new feature. Such a simplification is called desugaring, as it removes the so-called syntactic sugar from the language.
The use of the SYB generic programming framework, although convenient, has its limitations, as it needs multiple traversals for applying all variables. A more efficient approach would be to perform a single stateful traversal of the tree, carrying along a symbol table with the values of all the variables bound so far.

This example gives an overview of the techniques for defining, instantiating, traversing and evaluating an embedded language in Haskell. It shows the power of generic traversals along with pattern matching to concisely specify transformation rules for simplifying and evaluating an embedded program.

### 2.3.2 Template Haskell

GHC provides a way of acquiring a syntax tree for a specially marked portion of a Haskell program. It also provides an Application Programming Interface (API) for querying the compiler’s name and type information for the surrounding program. This capability comes in the form of the Template Haskell language extension and library.

The module `Language.Haskell.TH` introduces the types for a complete representation of Haskell programs as an AST. The core types are `Exp`, `Dec` and `Type` which define forms for expressions, declaration statements and types respectively. The `Q`-Monad encapsulates the compiler context and provides operations for querying and modifying the variable and type tables. Additionally the module includes a lot of helper functions and useful type synonyms.

The language extension defines a set of complementary brackets which lift the enclosed expressions from code to AST and vice versa. These are summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Bracket</th>
<th>Effect</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\ldots)$</td>
<td>Splice the given AST into the program</td>
<td>Code (Depends on given AST)</td>
</tr>
<tr>
<td>[\ldots</td>
<td>\ldots</td>
<td>\ldots]</td>
</tr>
<tr>
<td>[e</td>
<td>\ldots</td>
<td>\ldots]</td>
</tr>
<tr>
<td>[t</td>
<td>\ldots</td>
<td>\ldots]</td>
</tr>
<tr>
<td>[d</td>
<td>\ldots</td>
<td>\ldots]</td>
</tr>
<tr>
<td>[p</td>
<td>\ldots</td>
<td>\ldots]</td>
</tr>
</tbody>
</table>

Table 2.1: Template Haskell Syntax for quotation and splicing

When GHC encounters a quoted expression, it is parsed and type-checked. Likewise, a spliced AST is type-checked during compilation. This makes the language
extension safe to use, as only code that matches its type environment can be generated.

2.3.3 Compiler API

While Template Haskell is a language extension geared towards GHC, there is another interface for obtaining a Haskell AST from the compiler: GHC provides its internal API for extensions to the compilation process itself. This makes it possible to intercept the AST at any stage in the compilation pipeline.

The GHC compilation pipeline can be split up into three general phases, with their respective internal representation. The first phase is the front end, which parses the Haskell syntax into an abstract syntax tree of the full language. It does the first analysis, including name resolution and type-checking [29]. Following that, the Haskell AST is desugared into the much simpler Core language ("CoreSyn" in GHC terms, "GHC-Core" for the purpose of this thesis). GHC-Core only contains a reduced set of possible expressions, that is much easier to work with, while still retaining the important parts of the program structure. It is an explicitly typed second order lambda calculus. Polymorphic types are represented as normal lambda abstractions, i.e. there is no syntactic difference between a type abstraction and a value abstraction. The complete definition of GHC-Core’s expression type can be seen in Listing 2.3.

Some examples for desugaring from Haskell are patterns and guards in function definitions, which are all represented by the case expression in GHC-Core. Another example are where and let clauses which both result in a recursive let-binding.

GHC further optimizes the program, applying a large number of successive transformations, until it is finally translated into the Spineless-Tagless-G (STG) Machine language (StgSyn in GHC terms). The STG machine is an abstract machine used to precisely define the operational semantics of the STG Language itself [17]. The language contains no more information than strictly needed for execution of the program. While it is a functional language with denotational semantics, it is designed in such a way, as to enable an efficient mapping to executable machine code for modern hardware. This makes it interesting for the interface between the compiler and its code generation backend for a target language or platform.

The GHC does all its internal optimization on GHC-Core. There are several passes, the first of which is applying small local optimizations. Additionally there are larger ones like strictness analysis (identifying non-lazy subprograms) and the elimination of common subexpressions. Following that, the code is cleaned and tidied up for either bundling it into a Haskell-interface file or translating it into STG. During optimization,
2.4. Compiling Functional Languages

The compilation of functional programming languages is a wide ranging field. Even though most of the work in this area has been done in the field of software compilation, most techniques are applicable to hardware as well.

2.4.1 Transformation and Translation

In the context of this thesis, transformation is the application of a function on the AST that will yield an AST of the same type. Conversely translation is the application of a function which returns an AST of a different type.

Listing 2.3: GHC-Core Term definition [28]

```
type CoreExpr = Expr Var

data Expr b  -- "b" for the type of binders ,
    = Var   Id
    | Lit    Literal
    | App (Expr b) (Arg b)
    | Lam b (Expr b)
    | Let (Bind b) (Expr b)
    | Case (Expr b) b Type [Alt b]
    | Cast (Expr b) Coercion
    | Tick (Tickish Id) (Expr b)
    | Type Type

type Arg b = Expr b
type Alt b = (AltCon, [b], Expr b)

data AltCon = DataAlt DataCon | LitAlt Literal | DEFAULT

data Bind b = NonRec b (Expr b) | Rec [(b, (Expr b))]
```

a lot of the structure of the program is mutated and rewritten, sometimes changing the high-level structure of the program beyond recognition.

The compilation of functional programming languages is a wide ranging field. Even though most of the work in this area has been done in the field of software compilation, most techniques are applicable to hardware as well.
2.4.2 Defunctionalization

Defunctionalization is a transformation for removing higher order functions from a program. It is typically used for implementing functional languages in a software context. The transformation replaces all occurrences of functions in value positions with an identifier for the corresponding function. Additionally all calls to such values are replaced by a call to a global `apply` function, passing in the identifier alongside the arguments. `apply` then consults a table which maps the identifiers to their corresponding function and executes the call [11].

One issue is that all functions used in value positions need to be in the global scope for this to work. Additionally, this approach does not map well to hardware synthesis. Every call to the apply function would instantiate every function in the design which matches the type signature. In order to make this feasible, great care would need to be taken in order to encode the identifiers in such a way, that the synthesis tool can infer which functions are possibly called and remove the rest through dead-code-elimination. Alternatively this analysis could also be performed during defunctionalization.

2.4.3 Specialization

Specialization is another technique for removing higher order features from a program. Given a higher order function, the specializing transformation creates a copy of the function for every occurrence of a call to this function. This copy is modified such, that the higher order arguments which have a function type are removed. Instead, all calls to the argument function within the body of the copy are replaced by the actual function that was given at the corresponding call-site. This removes the need for representing functions as values and makes it possible to execute the program or synthesize hardware from it.
An important aspect of this transformation is the correct scope of the resulting copied function. The scope needs to contain the function passed in as an argument, while also containing the whole scope of the original higher order function. This can be mitigated by making sure that the higher order function has no free variables, i.e. only depends on its arguments and inserting it at the same hierarchy level and the same scope as the call-site. In Haskell this would be the where-clause of the function containing the call-site.

One downside of this approach is, that there are potentially many call-sites which could increase the program size considerably. For hardware synthesis, this is not much of an issue, since this is only a compile time cost, which is not relevant for the resulting circuit. [18].

2.5 The ForSyDe Methodology

2.5.1 Introduction

Formal System Design (ForSyDe) is a methodology developed at Royal Institute of Technology (KTH) [24] for the design of embedded systems. The fundamental idea is to first describe the system on a highly abstract level and then refine the system model using a sequence of semantic preserving transformations, eventually arriving at a solution which is "Correct by Construction" [26, 24]. Within this scheme, the first model is used for capturing all the functional requirements, and subsequentially explore the solution space towards an implementation that optimizes a cost function while fulfilling all constraints.

Special focus has been laid on the fact that the model allows formal, equational reasoning for example to prove that transformations are in fact semantic preserving. Another use of a formal basis is the analytical derivation of certain system properties, e.g. buffer sizes or execution time.

2.5.2 Process Networks

A "model" within the ForSyDe framework is a collection of processes which can communicate through "signals".

**Signal** A signal is a potentially infinite sequence of events where each event is composed from a value and a tag. The value represents the data that is communicated and can be a symbol of arbitrary type. The tags are used to define a meaning of time
for the system, by giving a partial order on events. Building on this concept, state can be represented by shifting the tags with respect to the signal values they originally belong to, thus effectively looking back in time. The precise meaning of tags and how they are affected by computation and communication is defined by the Model of Computation (MoC) [16].

Simulating systems with functions on infinite sequences may seem counterintuitive at first, but lazily evaluating the signals only as far as needed provides a suitable avenue for implementation.

**Process** Processes are pure functions on signals. This means they take a number of (possibly infinite) sequences and return another sequence as its reaction to the input signals.

In ForSyDe processes are built from higher order process constructors. A process constructor instantiates a process given the set of input and output signals as well as zero or more process functions which describe the actual computation to be done.

Processes are stateless which means that all state in the system is only kept in the signals. This makes analysis easier as there can only be local influences as opposed to global state, which can influence the whole system at once.

**Hierarchy** Process networks can be encapsulated into a system definition. A system has a defined set of input and output signals which are internally connected to the processes which make up the system. A system can be instantiated within another process network, to provide the aggregated functionality of its enclosed processes. In this way, a hierarchy of subsystems can be built, which allows to reuse components and manage the complexity, by hiding parts of the implementation. Formally, a subsystem can be expressed as a pure function on its input and output signals. Thus the use of hierarchical processes is consistent with the basic definitions.

### 2.5.3 Models of Computation

In order to model systems, it is not only important to describe what needs to be computed and in what order, but also when and where this computation happens and what triggers it. Models of Computation put this information on a formal basis by providing a semantic [16].

ForSyDe is built on the abstraction of MoCs, making it possible to describe systems in various domains of a wide range of abstraction levels.
2.5.4 Available Implementations

Currently there are four implementations of the ForSyDe methodology:

1. ForSyDe.Shallow
2. ForSyDe.Deep
3. ForSyDe SystemC
4. F2CC (CUDA)

**ForSyDe.Shallow** The earliest implementation was ForSyDe.Shallow which is a shallow EDSL in Haskell. It implements the most models of computation and can be used to quickly prototype a system for simulation. It currently implements six models of computation for simulation [25]:

1. Stochastic
2. Synchronous (SY)
3. Continuous Time (CT)
4. Untimed
5. Dataflow
6. Synchronous Data Flow (SDF)

**ForSyDe.Deep** The next implementation ForSyDe.Deep, also implemented in Haskell, provides a way to derive a hardware description from the system model. It is a deep EDSL which uses Haskells own introspection library Template Haskell as well as observable sharing [10], to extract the system structure and generate a netlist description. This netlist is then serialized as VHSIC Hardware Description Language (VHDL). ForSyDe.Deep only implements the synchronous MoC. It can be readily mapped to a synchronous digital circuit description. This thesis project is mainly concerned with this implementation. See Section 3.1 for a more in-depth discussion.

**ForSyDe SystemC** Following these experimental Haskell implementations, the design methodology was also ported to the industrially established SystemC language. SystemC is based on C++, implemented as a special library for describing, simulating and synthesizing systems. ForSyDe SystemC extends these capabilities, providing process constructors and a custom XML output format for interoperability between additional tools in the ForSyDe framework. ForSyDe SystemC models can be analyzed for extracting an XML representation of the system. Additionally the implementation supports distributed simulation through the distributed discrete event MoC [3]. The supported MoCs are:
2.6 Expressing Parallelism

One of the most important notion for describing the structure of hardware is parallelism. All functional units in a circuit are operating at the same time, performing computations in parallel. Since modern integrated circuit fabrication techniques reached the end of essentially free frequency scaling, it became even more important to harness parallel computation for further growth in performance. This applies equally to the description of programs running on parallel processors as well as circuits being implemented for modern process nodes.

The limiting factor for exploiting parallelism are data dependencies. Speculative execution aside, a computation can not be performed in parallel to another computation that produces its inputs. For modeling systems that are expected to work in a massively parallel manner, it is important to be able to express and control these data dependencies succinctly in order to expose the maximum amount of parallelism.

One way of modeling parallelism is through the description of intercommunicating processes, each performing continuous or sequential computations, while exchanging messages through explicit communication channels. This approach is taken by most hardware description languages, first and foremost by the mainstream industrial languages VHDL and Verilog. These languages also make the powerful assumption of synchronous system behaviour. This assumes that computation takes no time at all, meaning that the output of a process is immediately available when an input changes.

According to Skillicorn et. al. languages for parallel computations can be differentiated according to the layer of abstraction with respect to decomposition, mapping, communication and synchronization [26]. Decomposition is the process of splitting up...
an algorithm into concurrently executable parts. Mapping is the allocation of processes to computation resources. Communication is the exchange of data between processes. Finally, synchronization is the coordinative signalling for reaching a common state between processes, often associated with timed access to shared resources or the indication of completion of a computation.

Within Skillicorn’s framework the synchronous process network commonly used for hardware description can be classified as abstracting mapping and synchronisation which are not given by the system designer. Conversely, decomposition is made explicit through the division of the system into processes. The connection of processes using signals makes the communication aspect explicit.

### 2.6.1 Skeletons and Higher Order Functions

An important goal of EDSL for hardware is to raise the level of abstraction by making decomposition and communication implicit. One way in which the abstraction of communication can be achieved is the use of skeletons making the communication channels between processes implicit. A skeleton defines a communication network, which has a predefined shape that can be parameterized by the processes inside as well as the data type of the connections. One way of implementing these is in the form of higher order polymorphic functions. The arguments of function type specify the computational behaviour of the processes in the network, while polymorphism ensures, that the skeleton can be used for all desired functions, independent of their actual argument types [21].

The programmer only needs to implement the argument function as well as supplying the data to be processed and the rest is taken care of by polymorphism and the skeleton definition. There is no need to instantiate and wire up the entities that are implementing these functions.

The two most common skeletons for data parallel computations are map and fold. map applies a function to every element in a given vector, comparable to a for-each loop without data dependencies between its iterations in imperative languages. fold applies a function to every element, while carrying some state from one application to the next. This is equivalent to a loop with dependencies between its iterations. Typically the data dependency graph of a fold has a sequential shape. A special case occurs, when the argument function is associative. Then the order of applications does not matter, which allows to reorder the sequential shape into a tree form, thereby greatly reducing the longest path through the structure. See Figure 2.3 for the Data Dependency Graph (DDG) of these functions.
2.6. EXPRESSING PARALLELISM

2.6.2 Forms of Data-Parallelism

Data parallelism in process networks can be expressed in several ways and on different levels of abstraction. Recall that a process is the most basic unit of analyzability. For the purpose of this thesis, one can distinguish three different forms of expressing data parallelism within the ForSyDe framework. They are summarized in Figure 2.4, and are discussed below in more detail.

![Figure 2.3: Higher order functions map and fold](image)

![Figure 2.4: Comparison of internal, explicit and external parallelism](image)

**External Parallelism**

Functions for expressing external parallelism are process combinators on the host language level. This function instantiates and arranges a collection of processes to achieve the desired functionality. One example for this is the mapXSY constructor which ap-
plies a function to every element of a vector in the input signal. It is implemented as a normal Haskell function which instantiates a process with the given function for every vector element of the input signal. The processes are connected to the input signal through a unzipSY process which splits up the vector signal into a vector of signals. On the output side the signals are aggregated again, using a zipSY process. See Figure 2.5 for an illustration of this network.

![Figure 2.5: Process network for map with external parallelism.](image)

Since this form of parallelism is just an arrangement of processes it is easy to implement. On the other hand, it discards the available information about the structure of parallelism. An optimizing compiler would need to reestablish the fact, that e.g. all the processes involved in a parallel map are indeed identical before it could be used for optimization. Additionally, it is sometimes inconvenient from a programmers point of view, to split up a process function into multiple processes, due to a data-parallel operation inside.

**Explicit Parallelism**

In order to retain the information about the structure of parallelism, a new process constructor can be introduced. This makes all the information explicit and usable by an optimizing compiler. One example would be to try to exploit the fact, that a fold with an associative function can be represented as a tree instead of a sequence. The compiler would need to obtain or prove this property for the process function. It would be difficult to recover enough information from a fold expressed through external parallelism to even recognize the possibility of this optimization. Another example is the use of vector processing units in a modern Central Processing Unit (CPU) or Graphics Processing Unit (GPU), when compiling software. An explicit map already contains the information, that the operations are the same across the whole vector,
whereas for a loose collection of processes, such as it is produced by an externally parallel \textit{map}, this information would need to be expensively recalculated.

**Internal Parallelism**

The third and final form of parallelism that is distinguished in this context is internal parallelism. This is defined as data parallel operations which are directly applied within the process function. This implies that they are not visible in the process network, but hidden inside the opaque process function. The overall ForSyDe framework is not concerned with these functions, as only the compiler that translates process functions into synthesizable code needs to be aware of the parallelism. Conversely this feature provides better expressivity of the language since it makes it possible to actually process vectors reasonably within the body of a process function. When this feature for data parallelism is not available, the logical flow of a process function needs to be disrupted for passing a vector off to other processes for performing parallel operations. Alternatively the only option is to resort to a lower level of abstraction where the data flow is described manually by deconstructing the vector and enumerating all operations. Changing the level of abstraction within a process function, either through explicit processes or manual decomposition, hampers readability and understandability as well as increasing the difficulty of maintenance.

**2.6.3 Raising the Abstraction on Mapping**

Earlier it was mentioned, that HDLs already abstract the mapping of processes to hardware resources. Using modern techniques within an optimizing compiler, it is possible to raise the level of abstraction even further. One such possibility is the expansion of concurrency in the time domain instead of the space domain. This means that concurrent functions are not assigned to individual processing resources but are scheduled for execution on the same functional unit. The resulting trade-off is between hardware resources and time to completion. When using explicit parallelism, it is possible to parameterize the process constructor according to the amount of parallelism desired, or even leaving this decision up to the compiler [32]. Such a parameter could be exposed to a tool for automatic design space exploration [23] which could perform experiments to find a trade-off which fulfills all requirements while minimizing resource consumption.
Chapter 3

Functional Hardware Description Languages

3.1 The ForSyDe.Deep Language

ForSyDe.Deep is one of the modeling languages available within the ForSyDe framework. It is implemented in Haskell and provides a compiler to translate a synchronous system model into a sequential circuit description which is then written out as VHDL. An alternative backend generates a GraphML [8] description of the process network for visualization and documentation purposes.

3.1.1 Language Example

ForSyDe.Deep follows the ForSyDe convention of using process constructors to instantiate processes. A process constructor takes the parameters needed for specifying its behaviour as well as its input signals as arguments. Most process constructors are higher order functions. The function argument specifies the process's response to input signals. A definition of a process function is shown in Listing 3.1. The given example shows a function which calculates the sum of a vector of four integers. The ! operator represents the vector indexing function, which returns the value at the given index.

In order to be translated, the function needs to be made accessible to the ForSyDe compiler. For this, recall the principle of operation of Template Haskell from Section 2.3.2. Wrapping the declaration into \[d| \] quotes makes the AST available to be passed to the \texttt{newProcFun} function. It wraps the declaration into a data structure which contains the AST for translation alongside the actual function value for simulation.
3.1. THE FORSYDE.DEEP LANGUAGE

Functions may also contain additional declarations in their where clause. The following example given in Listing 3.2 implements an inner product for a fixed size vector. It reuses the definition of the \texttt{sum} function and adds a \texttt{mult} function, which takes two vectors and multiplies them element wise. This is done by obtaining all elements using the indexing operator \texttt{!} and concatenating the results of multiplication using the vector prepend operator \texttt{+>} which prepends the value of the left argument to the vector given as the right argument. An important restriction for declarations in where clauses is the need to supply a type signature, immediately preceding the actual declaration.

\begin{verbatim}
innerProduct = $(newProcFun [d] innerProduct :: FSVec D4 Int32 -> FSVec D4 Int32 -> Int32
   innerProduct a b = sum (mult a b)
   where
   sum :: FSVec D4 Int32 -> Int32
   sum v = v!0 + v!1 + v!2 + v!3
   mult :: FSVec D4 Int32 -> FSVec D4 Int32 -> FSVec D4 Int32
   mult x y = x!d0*y!d0 +> x!d1*y!d1 +> x!d2*y!d2 +> x!d3*y!d3 +> empty ||)
\end{verbatim}

Listing 3.2: Definition of a composite process function

For using this process function within a system, a process constructor is needed. One of the most basic constructors is \texttt{mapSY}. As the name suffix suggests, it belongs to the synchronous MoC which in fact is the only MoC supported in ForSyDe.DEEP. The process has one input signal and one output signal. Every cycle it applies the process function to the current signal value to compute the output signal. Similar to \texttt{mapSY} there exists a variant \texttt{zipWithSY} which takes two input signals and a process function of two arguments.

\begin{verbatim}
iprodProc inpA inpB = zipWithSY "iprod" innerProduct inpA inpB
\end{verbatim}

Listing 3.3: Process definition for applying an inner product

\texttt{mapSY} creates combinatorial processes, as the application of a process function
is stateless. ForSyDe.Deep also provides a number of sequential constructors which have state carrying internal signals. The basic building block for stateful circuits is the delaySY constructor, which shifts a signal in time by one time unit. Thus the output signal is the value of the input signal at the previous time step. In circuits it is implemented as a register. All other sequential process constructors internally use this for representing stateful computations. An example for a sequential circuit is given in Listing 3.4 as a shift register. It uses the scanlSY process constructor, which implements an FSM with an initial state and a process function for determining the next state from the input and the state. The output of the process is the new state. Thus it behaves as a mealy machine. The shift register functionality is implemented using the shiftr function for fixed size vectors which pushes in a new value from the right, while discarding the leftmost item.

```
shiftreg inp = scanlSY "shiftreg" next init inp
where
  init = copy D4 0
  next = $ (newProcFun
    [d| next_state :: FSVec D4 Int32 -> Int32 -> FSVec D4 Int32
    next_state state input = shiftr state input !!]
```

Listing 3.4: Process definition for a shift register

A basic building block of many signal processing circuits is a Finite Impulse Response (FIR) filter. This type of filter with input signal \( x(t) \) and output signal \( y(t) \) as well as \( N \) coefficients \( c_i \) operates according to the principle given in Equation 3.1.

\[
y(t) = \sum_{i=1}^{N} c_i \cdot x(t - i)
\]  

(3.1)

It needs a shift register for storing the last \( N \) signal values as well as a set of adders and multipliers for computing the sum of products. Reusing the processes defined so far, results in the system definition for the complete FIR filter given in Listing 3.5

```
fir :: Signal (FSVec D4 Int32) -> Signal Int32 -> Signal Int32
fir coefs input = iprodProc coefs (shiftreg input)
```

Listing 3.5: System definition for a FIR filter

This circuit description needs to be encapsulated in a system definition in order to be reusable and synthesizable. A system definition for the FIR system is given in
Listing 3.6. The system definition takes the function \texttt{fir} defining the circuit network as well as lists of identifiers for the input and output ports of the system. Additionally, a system identifier name needs to be given. These names are used for generating a readable system description during translation and synthesis.

\begin{verbatim}
  firSys = newSysDef fir "FIRfilter" ["c", "x"] ["y"]
\end{verbatim}

Listing 3.6: System definition for a FIR filter

### 3.1.2 Architecture

In ForSyDe.Deep the process functions and the processes themselves are translated separately. The structure of the system is derived from the applications of process constructors to signals. The result is a graph structure which connects the processes on the vertices using the signals on the edges. Each node has a number of tagged inputs and outputs which can be associated with a signal [1].

The system functionality is kept within process functions. A subset of the Haskell language is translated directly from the TH AST to a reduced VHDL AST. An important issue during this translation is the correct representation of types. Tuple types in Haskell are translated to record types in VHDL. Similarly, fixed-size vectors are instantiated as arrays in VHDL and the corresponding primitive functions for the unconstrained array type are generated. The resulting type definitions are then placed in a model-wide VHDL library [1].

All VHDL is generated as an AST which supports the needed subset of the HDL. This AST is then pretty-printed in order to obtain the actual VHDL code.

Figure 3.1 shows an overview of the compilation pipeline for ForSyDe.Deep. The system model written in Haskell is compiled and run, which provides the AST of all process functions as well as the netlist graph containing the signals and processes. By traversing the netlist, the structure of the system is translated by creating a VHDL block structure for every process and a VHDL entity for every hierarchical element of system definitions. Every process function encountered during this traversal is translated as well, performing a local traversal of the corresponding Template Haskell Abstract Syntax Tree (TH-AST). The resulting VHDL AST is then pretty printed to obtain the VHDL source code for further synthesis.
3.2 The Cλash Language

The Cλash project, developed at the University of Twente [18, 4], turns Haskell into a hardware description language, by providing an alternative interpretation of the basic functional language principles. As such, a function describes the structure of a circuit network, whereas a function application represents an instantiation of said circuit. Function arguments and variable bindings result in circuit network connections i.e. signals. Cλash employs the powerful type system of Haskell to derive the circuit level representation of signal values.

The Cλash language would significantly change the compilation process of the typical Haskell compiler. However it is still possible to share large parts of the language parsing and type-checking process with existing implementations. Because of this, the implementers chose to use the GHC by extracting its internal language representation, the GHC-Core language. This is a explicitly typed higher order lambda calculus. Haskell is desugared and further simplified to GHC-Core. Since GHC-Core is an internal format to GHC there are no guarantees of compatibility between versions. In order to protect the Cλash implementation from incompatible changes, Cλash has its own mostly compatible version, Cλash-Core of it. The Cλash compiler will then use Cλash-Core to do its own transformations and simplifications which will result in a program that is easily translatable to a VHDL netlist of the desired circuit.

3.2.1 Language Concepts

Cλash implements an alternative interpretation for Haskell, while keeping the semantics of the language. This means, that for combinatorial circuits, the straight forward evaluation of the program, using conventional execution, yields the same system response to individual constant stimuli as the circuit interpretation.
3.2. THE C\text{\LaTeX}ASH LANGUAGE

Sequential Circuits

Lifting this behaviour to the time domain and introducing state is encapsulated by the \texttt{Signal} type. As in most other functional hardware description languages, a synchronous signal is considered as an infinite sequence of values, each valid for one clock cycle. See Section 2.5.2 for a more detailed discussion of this type of signals.

Sequential functions in C\text{\LaTeX}ash take \texttt{Signal}s as input arguments and produce a \texttt{Signal} as output. Since numeric signals are so common, the \texttt{Signal} type has \texttt{Num}, \texttt{Ord} and \texttt{Frac} instances for those value-types that also support them, which allows to directly use the arithmetic operators on them. Other functions operating on the value-type can be lifted to the sequential domain using the \texttt{fmap} function defined in the \texttt{Functor} instance of \texttt{Signal}. Defining the function in this way, yields a sequential function but it still won’t have any state.

For adding state to a function, the primitive \texttt{register} function is used. Again it works the same as \texttt{delaySY} discussed in Section 3.1.1, by inserting an initial value and “shifting” the sequence of signal values, thereby making the previous value accessible to a function. There are several higher order functions for constructing state machines based on the \texttt{register} primitive. Given a transition function that determines the next state, an output function that defines the output for all states and the initial state, the \texttt{moore} function constructs a Finite State Machine (FSM). The \texttt{mealy} function works the same, while taking a combined transition and output function.

Translation

In order to translate a circuit description, C\text{\LaTeX}ash needs to know where to find the top-level component of the circuit. This is assumed to be the function with the magic name \texttt{topEntity}.

When the design is synthesized, it needs to conform to the surrounding environment. Because of this, it is possible to supply port names for all the top-level arguments as well as configuring clocks and reset settings through an annotation pragma.

Simulation

A circuit description in C\text{\LaTeX}ash can be simulated by simply evaluating the program with the set of inputs. During simulation \texttt{Signal}s are isomorphic to infinite lists. Using lazy evaluation it is possible to evaluate the system for a finite number of cycles. For this purpose C\text{\LaTeX}ash provides the \texttt{simulate} and \texttt{sampleN} functions which produce the unevaluated infinite output \texttt{Signal} and the first \texttt{N} values in the sequence respectively.
Complex interactions between the circuit and the environment are often impossible to reasonably implement through a fixed input sequence to the circuit. A good solution in this case is to implement a simple model of the environment that won’t be part of the circuit description itself. In Cλash this can be done by simply defining corresponding sequential functions which model the environment. Those will execute normally during simulation, but won’t be translated for synthesis because they won’t be part of the top-level function. However this type of environmental simulation is limited to the synchronous domain of the circuit and can not easily represent e.g. continuous signals without additional consideration and implementation of the sampling and similar issues.

3.2.2 Clash Implementation

Internal Representation

Cλash employs GHC’s front end, extracting the desugared but not simplified GHC-Core. The loss of high level structural information during simplification proved to be problematic for the implementation of Cλash, precluding the use of the simplified GHC-Core representation [18, p. 33].

```
--- | Term representation in the CoreHW language: System F + LetRec + Case

data Term
  = Var !Type !TmName
  | Data !DataCon
  | Literal !Literal
  | Prim !Text !Type
  | Lam !(Bind Id Term)
  | TyLam !(Bind TyVar Term)
  | App !Term !Term
  | TyApp !Term !Type
  | Letrec !(Bind (Rec [LetBinding]) Term)
  | Case !Term !Type [Bind Pat Term]

  deriving (Show, Generic, NFData)

--- | Term reference

type TmName  = Name Term

type LetBinding = (Id, Embed Term)

--- | Patterns in the LHS of a case-decomposition

data Pat
  = DataPat !(Embed DataCon) !(Rebind [TyVar] [Id])
  | LitPat !(Embed Literal)
```
Overall Cλash takes an approach similar to the GHC pipeline. The GHC frontend is used for parsing and type-checking. Further processing is done on the internal Cλash-Core which is derived from GHC-Core. Cλash-Core has some additional features compared to GHC-Core. The first of which is the distinction between type abstraction and value abstraction. While GHC-Core treats types and values the same in abstractions, Cλash-Core has separate lambda terms for type and value abstraction. Because of that a type does not need to be represented as an expression and can be handled separately. Finally, the option to add annotations (Tick marks in GHC-terminology) is not given in Cλash-Core. [6] Since GHC-Core is already type-checked and fully type-annotated by the GHC the types of all variables are known or explicitly abstracted.

Normalization and Optimization

Similar to the GHC pipeline, Cλash simplifies the program through a system of normalizing transformations. It finally arrives at the Intended Normal Form (INF). The INF is only loosely defined as the “result of applying all transformations until they reach a fixpoint and fail”.

It needs to fulfill the requirement of only containing representable types [5]. Representable types are those, that can be expressed as a bit-pattern of a certain length. This includes Algebraic Data Types (ADTs), and primitives as well as functions. Polymorphic types may not remain in INF, as they can not be represented. The transformation system needs to specialize these occurrences, which results in monomorphic instances.

One important difficulty is to ensure termination of the transformation system. Some transformations might never arrive at a fixpoint. One such example is the in-
lining of a recursive function, when done in top-down order. Some combinations of transformations will form cyclic substitutions when applied one after another. This happens, when one transformation effectively undoes the work of the other transformation. Removing these non-terminating transformations is difficult to impossible. There are some strategies that can applied though. In Clash a function may only be inlined once at every application site. This ensures termination for the former case. For the latter case an option can be to impose a strict ordering between problematic transformation rules, thereby breaking the cycle.

The transformations are driven by a “strategy” which is implemented using a set of custom combinator functions. These combinators take a transformation rule, and provide facilities for applying the rule recursively or repeatedly until it reaches a fixpoint. Each application of such a combinator yields another transformation rule which can be further combined with others.

A rewrite rule operates within a monadic context, which provides functionality for unique name generation, caching of performed specializations on functions as well as a number of counters for catching non-termination of the transformation system. The counters are used for keeping track of the number of applications of a specific rewriting rule, like e.g. inlining, in order to abort compilation in case of an excessive number of repetitions. As seen in Section 2.3.1, pattern matching is used effectively to concisely specify the substitution rules.

In summary, the term rewriting rule system provides a powerful, composable and extensible framework for implementing and supporting the reinterpretation and translation of an increasing number of features in the host language Haskell.

Translation

The final step of compilation is the translation of the normalized Clash-Core program into the final circuit description. The backends are implemented in a modular fashion, which allows a new backend to be implemented easily.

After normalization, there are only syntactical constructs left, which fulfill the constraints concerning representability of their arguments. They are translated into a language agnostic netlist of components.

Most hardware description languages have semantics that are sufficiently non-functional that a direct translation from Clash-Core to the target language would need to smooth out all the slight differences in interpretation of the basic syntactical constructs. The normalized Clash-Core is translated to a netlist format that can be easily described in all target languages because it matches their semantic model more
3.3 COMPARISON OF C\textsc{λ}ASH AND FORSYDE.DEEP

The netlist is built from components and their interconnections. Each component consists of a collection of ports and internal statements. A statement in turn can declare a new net, a (possibly conditional) assignment or another component instantiation. Starting from the top-level component, all subcomponents can be instantiated.

There are however some components that cannot be reasonably implemented in Haskell while being translatable. The primitive datatypes like integers and vectors as well as the basic operations on them are candidates for this. In case of integers, their implementation is given by the compiler and cannot be accessed for translation. The vector functions on the other hand are formulated recursively and also have some type-level arithmetic for type-checking their length, which interferes with automatic translation. Furthermore, these types typically map to special primitive types in the target language which makes a special case handling mandatory anyways. Since the declarations for these would be tedious to generate programmatically, C\textsc{λ}ash has a so-called “Black-Box” system. A black-box component is defined in a configuration file, which contains the Haskell name it should replace, and a suitable Haskell function for simulation, along with a textual code template for the target language. This template contains special markers (“holes” in C\textsc{λ}ash terminology), which can reference fields from the translation context of the component. When a black-box component is encountered during translation, the markers are expanded using the actual context present in the netlist, directly yielding the corresponding code in the target language.

3.3 Comparison of C\textsc{λ}ash and ForSyDe.Deep

In order to identify possible future directions for ForSyDe.Deep, it is compared with the C\textsc{λ}ash language. The first section summarizes the similarities, while the second one goes into more detail on the differences.

3.3.1 Similarities

Signals and State Both languages use a very similar semantic model for describing their circuits on the functional level. Evaluating a function with the value of a signal as an argument is considered equivalent to the computation of a combinatorial circuit. Sequential systems are modeled similarly as well, as both languages use the model of a possibly infinite sequence of values to represent a signal. Consequentially state is defined by looking back in time on the signal values. For this, ForSyDe provides the low-level delaySY constructor. C\textsc{λ}ash has an equivalent register function. Built
from this primitive, there are several helper functions of higher order that facilitate the description of FSMs. The \texttt{mealy} and \texttt{moore} functions in Cλash find their corresponding counterpart in the \texttt{mealySY} and \texttt{mooreSY} constructors in ForSyDe. As such the implementation of basic sequential circuits is not much different in both languages.

**Type-Level arithmetic for Vectors** Both languages provide an implementation of sized vectors, which carry their length within their type. While ForSyDe’s vector implementation depends on the \texttt{type-level} module which provides decimal arithmetic using functional dependencies on the type level, Cλash carries its own implementation which is based on type families. On top of that Cλash uses a GHC plugin which provides an advanced solver for type level arithmetic using the recently added natively supported type level natural number support in GHC [12].

**Testbench generation** Given a set of inputs, both languages are able to synthesize a testbench module which stimulates the unit under test and records the results for further processing in Haskell. This allows to automatically test the simulated VHDL model against the simulated Haskell model to verify the correctness of the compiler implementation. One notable technique in this context is to use the quickcheck [22] library for automatic, randomized generation of test input.

### 3.3.2 Differences

The most striking difference between the two languages is the modeling of circuit structure. ForSyDe chose an explicit description of every entity in the netlist, as a process, including its process function, while strictly differentiating between library functions and process functions. There is no way for a user to call their own functions from within a process function. Every user function needs to be explicitly wrapped into quotes in order to be translatable and needs to be introduced as an argument to a process constructor. While this enforces a certain amount of thought in the design of a system, it also hampers modularity because functionality that is needed within multiple process functions cannot easily be factored out.

**Syntax support** Generally ForSyDe only translates a limited set of Haskell’s language features. Among the missing features is the monadic do-syntax, user functions (as described above) and finally the full fidelity of pattern matching constructs for representing choice (Function patterns, guards and case statements). These restrictions can be attributed to the fact, that ForSyDe operates on the TH-AST. In order
to directly translate from Template Haskell (TH) to VHDL, every conversion step needs to consider the whole range of syntactical options at that point, thereby greatly increasing the complexity of the translation process. Cλash on the other hand, can leverage GHC’s desugaring machinery for condensing the system model into a simpler language. Cλash-Core only has one expression type for choice. All the pattern matching constructs mentioned above are simplified to this single type of case expression. From this basis it is much easier to cover a subset of Haskells syntax that is more natural and idiomatic.

**Type Awareness** The Cλash-Core AST is fully type annotated. This means that all information is available for transformation and translation. In contrast the TH-AST used for process functions in ForSyDe.Deep does not contain any type annotation apart from type signatures manually given by the user. Since VHDL requires the type of all signal and variable declarations as well as function arguments to be explicitly given, there is no way of introducing such constructs during translation without knowing the exact, translatable types for these expressions. Any sub-expression within the body of a process function is therefore required to have an equivalent counterpart in VHDL which can be written without the need of additional type information.

**Synthesis tools** ForSyDe provides helper functions for automatically synthesizing the model using the Altera Quartus toolchain. These allow to directly provide a pin-mapping and the target device, making it possible to describe the whole system purely in Haskell which is of great convenience to the user. Cλash only has support for top-level annotations, which allow to customize the final port interface in order to fit the design into a larger project. Driving the tools needs to be done separately.

**Interoperability** ForSyDe supports the output as GraphML which enables other tools to process the system model e.g. for performing automated transformations and optimizations or Design Space Exploration (DSE). Future versions of the framework may also provide compilers for different targets through this mechanism.

Cλash provides Interoperability on the VHDL side. This is done through the blackbox mechanism that is also used for primitive internal functions. It allows to transparently integrate existing VHDL Intellectual Property (VHDL-IP). This is an important precondition for achieving any industrial acceptance. However it makes any formal reasoning about the model dependant on the correct reimplementation of the Intellectual Property (IP) Core in Haskell.
Observable sharing  ForSyDe implements observable sharing according to [10]. This enables the use of circular data structures for representing the netlist. This introduces unsafety to the type system as the observable references depend on the unsafeCoerce function for dereferencing. This unsafety is encapsulated within the corresponding module though, so that the implementation can be verified.

Cλash does not have any special handling for sharing. Circular structures are built using binders in recursive-let blocks. During the normalization phase, great care is taken that no transformation creates copies of an expression but only references the corresponding binder.

Readability of generated Code  Cλash performs a large amount of transformations to the original design, which alter the structure considerably. Additionally the resulting VHDL code is split across a large number of files containing small entities which makes the original meaning unintelligible. On the one hand, the code is only meant for consumption by the automatic synthesis tool, so this is not a critical issue. On the other hand resource usage statistics for the synthesis process are generally given per hierarchy level which makes it difficult to track down irregularities in resource usage. ForSyDe handles hierarchy as a first class citizen, even having a separate compilation traversal for the process network which carries the system structure. This means the resulting design files closely resemble the original structure of the design.

3.3.3 Discussion

From a practical standpoint, the most important difference is the disparity in the subset of Haskell that is translatable. While Cλash is really close to idiomatic Haskell, models implemented in ForSyDe are relatively verbose, requiring manual type signatures or complex nested if-then-else constructs instead of an elegant pattern match.

An important point for future considerations is the limitation of type annotations within ForSyDe.Deep. As it is implemented, the translation process is little more than a direct syntactical mapping from Haskell syntax to VHDL syntax. It is extremely difficult to introduce concepts at a higher level of abstraction i.e. features that are not directly supported in VHDL, as these are often dependant on the knowledge of types involved. Some potential features that are directly affected by this are support for polymorphism and general higher order functions. But also dependently sized integer types are difficult to implement due to not knowing the size of an integer literal within an expression. This greatly reduces the expressivity of process functions. The expressivity of the language at large is not as limited though, as some of these
features are already supported at the process network level, where appropriate type information is available.
Chapter 4

Implementing Higher Order Functions in ForSyDe

4.1 Implementation

This section describes the implementation of higher order functions for process-internal parallelism in ForSyDe.Deep.

4.1.1 Goals

To make the description of hardware more powerful in ForSyDe.Deep and to extend the capabilities of the compiler, it should be possible to translate process functions which use the higher order vector functions like `map` and `fold` within the body of a process function.

4.1.2 Interfaces

An implementation of the desired functionality needs to be embedded into the ForSyDe.Deep compiler. The code will be interfacing with the compiler through the following interfaces.

Template Haskell AST  The process function that is to be translated is given as a TH-AST. It contains a function declaration which consists of the argument list, the function body and its accompanying declarations enclosed in the `where` clause. In the existing implementation, there are certain restrictions on the AST which are checked before translation. They are concerned with the type signatures, pattern matching as well as the types of expressions which are translatable. The most important restriction
is the availability of type signatures. Every declaration in the where clause needs to be
directly preceded by its explicitly spelled out type signature. Pattern matching is only
supported for function arguments, unpacking tuples and deconstruction of the AbsEx
type. Only a single clause per function is allowed, which disables the use of pattern
matching as a syntactical construct for choice. Expressions that are not supported are
lambda abstraction, the monadic do-syntax, list comprehensions and values as well as
record operations. Conditionals and case expressions are only supported when they
are found in the top-level position within a function body.

**VHDL-AST**  Traversal of the netlist and translation of all nodes, yields the imple-
mentation model as a VHDL syntax tree. The VHDL AST does not cover the whole
syntax of VHDL, as it is only made for generating the output by pretty-printing the
tree. As such it only covers the syntax which is needed for the current compiler feature
set.

**Translation**  The translation process takes the TH-AST and translates it into the
corresponding VHDL representation. This translation is done by a stateful recursive
traversal. The state is kept in the VHDLM monad, which carries tables of known func-
tions, algebraic datatypes and local declarations. In the TH-AST the application of
an operator, a function or a type constructor is represented by the same application
expression. Since these are rendered differently in VHDL, the symbol translation table
contains a function for every name, which translates a list of arguments to the correct
function call syntax in VHDL. The individual translation function only maps a list of
argument VHDL expressions to the corresponding function call syntax as a VHDL ex-
pression. It does not allow the introduction of auxiliary variables or functions through
the VHDLM monad and does not include the original TH-AST of the arguments.

**4.1.3 Design Considerations**

There are several options for the implementation, which are discussed in the following
paragraphs.

**Specialization vs. Defunctionalization**

VHDL does not support higher order functions directly, which means that they need
to be eliminated from the program in order to be translated. As discussed in Sec-
tion 2.4, there are two main options for removing this feature from the program,
namely defunctionalization and specialization. The arguments given in Section 2.4.3
make specialization a good choice for hardware synthesis. On closer examination, specializing a higher order function is in fact the degenerated case of defunctionalization with one separate apply function per call-site.

**Procedural vs. Functional VHDL Style**

Higher order functions for vectors which are targeted in this work, are typically implemented recursively, processing one vector element per recursive call. In VHDL, these structures are typically implemented using the sequential `for-loop` and the structural `for-generate` statements. Additionally VHDL supports recursion, as long as the recursion depth can be determined statically during synthesis, which is the case for fixed size vectors. Modern VHDL synthesizers should be able to map all these options to the same implementation, as they describe the same data dependency structure after elaboration.

**Transformation vs. Translation**

The existing implementation of ForSyDe.Deep is built around a single translation pass from the TH-AST to the VHDL-AST. Handling the translation of the higher order vector functions can be integrated into the existing translation flow.

Alternatively a transformational approach similar to the compilation strategy of C\(\lambda\)ash could be taken. This would perform a series of transformations on the TH-AST for simplifying it, before it is translated by the existing backend. A purely transformational implementation must fully eliminate the higher order functions from the AST and express them in terms of the existing feature set, before handing it off for translation.

Additionally a hybrid approach is possible where the higher order functions are transformed into a normal form, which is then translated.

**Permissible argument functions**

In order to be usable, a higher order function needs to accept all the functions one might need for describing data-parallel computations. Argument functions can be classified as follows:

1. Available in global ForSyDe library or module specific VHDL-Lib
2. Operator, translatable to VHDL expression
3. Local function in where clause
4. Local lambda abstraction
5. User defined function
A short discussion of each as well as the summary of estimated implementation complexity in Table 4.1 follows.

1. Available in ForSyDe or module VHDL-Lib The higher order function can be easily specialized for these argument functions, as the specialization only needs to insert a call to the corresponding VHDL function. There are no additional declarations or translations needed.

2. Operator The translation of an operator application to a VHDL expression is already defined. It can be reused, in order to use this expression in the loop body of the specialized function. There are no additional declarations or translations needed.

3. Local function in where clause A function within a where clause is translated as if it were a process function of itself. The resulting definition is placed in the local declaration area of the enclosing function. Thus the name of the function is sufficient for calling it.

4. Local lambda abstraction Lambda abstractions are not supported, but are useful for defining simple functions to be passed around. Support could be added by transforming the TH-AST in order to move the definition of the lambda function into a where clause, making it equivalent to item 3.

5. Partial operator application Haskell supports the partial application of infix operators to concisely define anonymous functions inline. This is similar to item 4. in that it defines an unnamed function locally. Support could be added by defining a named function within a where clause, containing the parameterized expression. Consequentially the implementation complexity is similar to item 4.

6. User defined function A global user defined function can only be translated if the TH-AST is available. As such it is impossible to translate a function that is imported or defined outside the scope of the process function. The user would need to register the function to the ForSyDe framework, using a special function constructor similar to the way process functions are defined. It is not clear, whether this would provide any benefit in terms of expressivity of the language and is consequentially considered impractical for the scope of this thesis project.
4.1. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Classification</th>
<th>estimated Implementation complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Available in module or ForSyDe VHDL-Lib</td>
<td>Low</td>
</tr>
<tr>
<td>2. Operator, translatable to VHDL expression</td>
<td>Low</td>
</tr>
<tr>
<td>3. Local function in where clause</td>
<td>Low</td>
</tr>
<tr>
<td>4. Local lambda abstraction</td>
<td>High</td>
</tr>
<tr>
<td>5. Partial operator application</td>
<td>High</td>
</tr>
<tr>
<td>6. User defined function</td>
<td>Impractical</td>
</tr>
</tbody>
</table>

Table 4.1: Classes of argument functions and their estimated implementation complexity

4.1.4 Solution Architecture

From the options discussed above, a specializing solution with a procedural translation target and a hybrid transformation approach are chosen. The hybrid approach is chosen, because it allows to explore whether the transformation system can also be used to increase the translatable subset of Haskell by applying desugaring transformations.

Normal Form

A call to a higher order function is transformed into a normal form. Since there is no procedural primitive for implementing loops in VHDL, the specialized form is assumed when the call to a higher order function is wrapped in a function which has identical arguments apart from the actual argument function. Thus the higher order argument is in a known position and can be treated specially during translation.

In normal form, the call is the sole expression in the function body. All arguments except for the argument function are passed through directly and are defined in the correct order. A type signature is mandatory as well. See Listing 4.1 for a collection of examples of function calls in specialized normal form. This ensures, that the types within the signature can be relied upon for translation purposes.

Listing 4.2 shows counterexamples containing functions which are not in normal form, although all are valid Haskell declarations.

In summary, higher order function calls are to be transformed into normal form, and translated. The normal form ensures that the type signature can be relied upon for all types involved in the translation of the higher order function.
map \_f \::\::\:\text{FSVec}\ D4\ \text{Int32} \rightarrow \text{FSVec}\ D4\ \text{Int16}
map \_f\ v = \text{map}\ f\ v

foldl\_f \::\::\:\text{Int32} \rightarrow \text{FSVec}\ D4\ \text{Bool} \rightarrow \text{Int32}
foldl\_f\ i\ v = \text{foldl}\ f\ i\ v

zipWith\_f \::\::\:\text{FSVec}\ D8\ \text{Int8} \rightarrow \text{FSVec}\ D8\ \text{Int8} \rightarrow \text{FSVec}\ D8\ \text{Int16}
zipWith\_f\ a\ b = \text{zipWith}\ f\ a\ b

Listing 4.1: Some specialized calls in normal form

map\_f = \text{map}\ f\ -- incomplete argument list
foldl\_f\ v\ i = \text{foldl}\ f\ i\ v\ -- order of arguments does not match
zipWith\_f\ v = \text{zipWith}\ f\ v\ v\ -- duplicated argument

Listing 4.2: Some higher order function calls which are not in normal form

Translation

The existing translation table for specially translated functions has translation functions of type \text{VHDL.Expr} \rightarrow \text{VHDL.Expr}. This is not sufficient for translation of the higher order functions, as there is the need for accessing the type signature and the scope of the currently compiled function. Because of this, a new translation table is introduced to contain translation functions of type \text{TH.Exp} \rightarrow \text{VHDLM (VHDL.Expr)}. This makes the \text{VHDLM} monad accessible for obtaining the current compilation state as well as sidestepping the need for translating the argument function to VHDL.

With these changes, translating the different higher order functions is a matter of defining the corresponding translation functions to produce the correct VHDL. As an example the VHDL code for the \text{foldl} implementation is given in Listing 4.3. This code is built as a subtree for the VHDL AST. The placeholders enclosed in <...> are replaced by the corresponding function types taken from the type signature. The resulting AST is included in the surrounding function’s auxiliary declaration area.

Up to this point the functionality works even without the transformation system. It is required though that the higher order functions are used in normal form.

Transformation

The transformation system is implemented as a desugaring transformation using SYB. It is the first step that is done within the \text{newProcFun}. The transformation has the type \text{[TH.Dec]} \rightarrow \text{Q [TH.Dec]}, which means that it operates on a list of declarations.
4.1. IMPLEMENTATION

Listing 4.3: Generated code for \texttt{foldl}

within the \texttt{Q} monad. This allows to access the compilation state of GHC from within the transformation function in order to generate unique names and query the symbol table.

The task of the transformation system is to transform all higher order features into the subset of Haskell which is supported by the translation system. Three transformations were implemented for specializing higher order functions as well as providing support for lambda function definition and partial operator application.

The transformations are implemented using a recursive traversal of the AST, visiting all function declarations. At each declaration, the function body is transformed by the transformation rule. The transformation may result in new function declarations, which are accumulated and appended to the declaration list of the current function declaration.

The transformation rule itself is applied to every node in the expression tree of a function declaration, by use of the SYB generic traversal mechanism. It matches a specific subtree using pattern matching and replaces it with the transformation result. Auxiliary declarations resulting from the transformation are first accumulated in a state variable and subsequentially included in the AST.

**Lambda definitions** The first transformation enables the use of lambda functions within the process function. It moves the lambda definition into a new function body and declaration. The original value is replaced by the name of the newly defined function, thus keeping the meaning of the code identical. Listing 4.4 shows an example of the transformation of lambda abstractions. Expressions similar to the expression above the line are translated into the expression below the line.
4.1. IMPLEMENTATION

Partial operator application The second transformation enables the translation of partially applied infix operators. A partially applied expression is moved into a new function body and the omitted operand is replaced by an explicit function argument. This transformation is similar to the previously defined one.

Normalization of higher order functions The most important transformation takes care of the normalization of higher order function calls. A higher order call that occurs within any expression is extracted and placed into a new function declaration while obeying the rules of the normal form described above. Listing 4.6 shows an example of this, where a call to `map` with an argument of `g` is moved into a new function `map_g` which satisfies the normal form.

The transformations described above produce valid Haskell code, which can be compiled and executed using GHC. They do not produce translatable code for ForSyDe.Deep though. For translation, type-signatures for all functions defined in where clauses are needed. This is due to the fact that VHDL strictly requires explicit type information.
for all function arguments, intermediate variables and return values. Internally, this 
information is needed for execution as well but GHC is able to infer the types of all 
subexpressions during type-checking. The type-inference run in GHC happens during 
type-checking which is done after splicing the AST into the program. The Q monad 
provides an API for querying the type of symbols known to the compiler. All sym-
bols encountered up to the current splice are accessible. ForSyDe.Deep captures the 
AST before splicing, which means that information about symbols defined within the 
spliced AST is not inferred yet when the translation to VHDL is performed.

As a workaround for this issue, all transformations require the use of explicit type 
annotations for all higher order expressions that are transformed. Adjusting the ex-
ample given in Listing 4.5 according to the requirement, yields the code given in 
Listing 4.7.

\[
\begin{align*}
  f &:: FSVec D4 Int32 \rightarrow FSVec D4 Int32 \\
  f \ vs &= \text{map} \ ((+1) :: Int32 \rightarrow Int32) \ vs \\
\end{align*}
\]

\[
\begin{align*}
  f &:: FSVec D4 Int32 \rightarrow FSVec D4 Int32 \\
  f \ vs &= \text{map} \ \text{operator}_0 \ vs \\
  \text{where} \quad \text{operator}_0 &:: Int32 \rightarrow Int32 \\
  \text{operator}_0 \ a &= a+1
\end{align*}
\]

Listing 4.7: Partial Operator Transformation with Type Signatures

For the transformation of higher order function calls, type-signatures are required 
as well. The corresponding example is given in Listing 4.8. In this case, the return 
value as well as all arguments, except the higher order argument function itself, need 
to be annotated for making the expression translatable.

Since the inline type signatures given by the programmer might be needed for 
further transformations, they are preserved throughout these transformations. Inline 
type signatures are not supported by the translation backend, which leads to a fourth 
transformation being defined. This transformation removes all inline type signatures 
from the AST. An example of this is given in Listing 4.9.

4.1.5 Results

The implementation of the translation system was performed according to the architec-
ture outlined above. The resulting system is verified to behave correctly by performing 
a set of tests containing different argument functions and setups which are now part
of the ForSyDe.Deep test suite. The systems were subjected to several runs of random test data generated by the Quickcheck [22] utility. The simulation results of the simulated system in Haskell are backed by the presumable correct implementation of higher order functions from the Prelude standard library. The translated system was simulated using the Ghdl simulator. The system response to the generated test data is compared between the different simulation environments, concluding with a passed result.

Additionally a slightly adjusted synthesized example FIR system derived from the one given in Section 3.1.1 is compared with another FIR implementation. The reference system is written in a structural style and does not use any higher order function within process functions. The adjusted example uses the higher order functions map and foldl1 instead of the manual specification of all operations. See Section B for the full code of the tested examples.

Both systems are configured as 9-tap 8bit with fixed-point multiplication instead of the standard multiplication (*) operation. The former provides the most significant word of the multiplication result instead of the conventional multiplication which yields the least significant word. The results of the synthesis using the parameters given in the code listing, using Altera Quartus 13.0.1 show, that both systems result in identical circuits.
4.2 Evaluation and Discussion

4.2.1 Language Features

The comparison of Cλash and ForSyDe.Deep given in Section 3.3 has shown that while the current implementation of Cλash is superior to ForSyDe.Deep in terms of features as a pure HDL, it lacks concepts for modeling heterogeneous systems. The ForSyDe framework has a broad and powerful conceptual base for describing these systems using different MoCs. Extending this part should be a priority in the future development of the language and framework.

4.2.2 Extension of ForSyDe.Deep

The extension of ForSyDe.Deep translation capabilities for data-parallel higher order functions has resulted in two subsystems which handle the normalizing transformation and the translation of the normalized result respectively. In principle, this architecture is promising as the transformation system is able to simplify complex Haskell language features and raises the level of abstraction possible within process functions. The translation functionality can be kept simple as it only needs to deal with the normalized base case.

However there are severe limitations for effectively implementing higher levels of abstraction within process functions. Any feature that is dependant on language constructs that result in the introduction of new functions or variables in the generated VHDL needs to have the full type information of the corresponding expression available. For typical Haskell programming, this is not an issue, as the types of all subexpressions are inferred during compilation. With the use of Template Haskell for extracting the AST, this information is not available. The only information available are type signature annotations which need to be explicitly given by the programmer.

Requiring type signatures is a workaround that introduces syntactical noise. This is especially cumbersome for lambda functions and partially applied operators, as these are intended as a lightweight way of defining functions. Adding full type signatures to such an expression increases the notational overhead, and typically reduces readability. In general, type signatures are a valuable tool for documenting the intent of code sections. This applies to the compiler for checking the code as well as to the programmer for reading and understanding the code. Especially for the latter role though, there is a fine line between improved readability and the addition of unnecessary noise.

One explanation for the difficulty of obtaining type information within Template
Haskell can be found in the intended use case of the language extension. It is originally made for template meta programming, meaning the composition of syntax trees for incorporation in Haskell programs. For that use case the whole machinery of type inference and type checking is available to compile the resulting Haskell program. This means that a transformation on the tree that provides higher levels of abstraction does not necessarily need to preserve or infer the type information as it can be recovered by GHC later on. For applying Template Haskell in the opposite direction to translate the AST into another language outside of the GHC framework this information needs to be available.

An interesting avenue of research would be the implementation of a type inference algorithm working on the TH-AST. The Template Haskell splices are evaluated during the rename and type-check phases. This means that the full type environment is not yet available when this inference would need to run. One question that remains is how incomplete the environment is in practice and whether the needed information for seeding the inference process can be made available unobtrusively e.g. through additional type signatures.

The true power of a higher level of abstraction in ForSyDe can be used on the level of process networks. Since this is a form of meta programming, the full power of type inference, higher order functions and polymorphism is already available. At this level, the importance of specifying an accurate system hierarchy and structure is important. In case of an extension of the compiler with new backends for modeling heterogeneous systems this is the most important aspect as it allows better reasoning for mapping the various processes of the system to implementation resources.
Chapter 5

Conclusion

5.1 Summary

This thesis has explored the design space for functional embedded domain specific languages for the description of synchronous circuits in Haskell.

The project has resulted in an in depth analysis and comparison of the two Haskell based embedded hardware description languages ForSyDe.Deep and Cλash. The fundamental concepts for building embedded languages in a functional host language like Haskell have been introduced and the basic building blocks and abstractions for describing synchronous circuits have been presented. Finally ForSyDe.Deep was extended to support the translation of data-parallel higher order functions in the style of process internal parallelism. The compiler was extended with a desugaring transformation system for extension of the supported subset of Haskell. Accompanying these efforts, the compiler was ported to the latest version of GHC, supporting both GHC-7.10 and GHC-8.0. A tool driver for the open source VHDL simulator GHDL was implemented and used for establishing continuous integration tests using the online build service TravisCI.

The major strengths and weaknesses of both languages have been shown. While Cλash provides powerful abstractions through the interpretation of idiomatic Haskell programs as circuits, the strengths of ForSyDe can be found in the modeling and structuring of heterogeneous systems. It has been shown that the support of higher order functions within process functions increases the expressive freedom for semantically structuring a model, while not causing measurable effects in the synthesis of hardware structures. Finally it has been shown that the architecture of ForSyDe.Deep is not well suited for providing a higher level of abstraction within process functions due to the lack of type information in Template Haskell.
These results show that a promising future direction of ForSyDe.Deep is to extend the modeling support for heterogeneous systems by adding additional backends. One such option could be the adoption of CAash itself as a backend compiler for process functions, thereby extending the translatable subset of Haskell.

5.2 Future Work

This thesis has shown some of the strengths and shortcomings of ForSyDe.Deep. The next steps are to build on the strengths in the area of modeling of heterogeneous systems. One avenue of research is that addition of further language backends either for interoperability with other tools from the framework or for the support of additional compilers to make more targets available. Following that, the support of a greater collection of MoCs could be a good addition.

Generally the following features are interesting for extending the compiler:

**Sized integer types**

For the efficient generation of Hardware, the number of bits per signal is an important choice. Currently, only 8, 16 and 32 bit wide integers are supported. An important feature is the support of signals with arbitrarily sized integers.

**Type inference engine for TH-AST**

The result of this work has shown, that the missing type information is blocking most interesting transformations from becoming freely usable. This could be alleviated by implementing a type-inference engine which operates on the TH-AST. This could be implemented according to the Hindley-Milner algorithm, which gives an efficient stateful traversal of the tree that determines the type of all named variables. Special consideration needs to be taken for the handling of type-level numerals used in the fixed size vector implementation, as this is heavily depending on GHC-specific type system extensions.

**Use GHC-Core instead of Template Haskell for the front-end**

Due to the limitations of the TH-AST, it might be more fruitful to directly use the GHC API for obtaining the AST of process functions. Since the AST types of template Haskell and the GHC API are not compatible, this change would have a severe impact on the whole translation system.
Use Cλash as a backend in ForSyDe

Another approach for the translation of process functions could be the use of Cλash as a compiler backend. Cλash could compile the process functions into a VHDL library of functions, that could be called by the individual process instantiations. The difficulty would be to find a way of specifying the functions to be translated to the Cλash API. This would vastly improve the supported subset of Haskell.

Improved support for pattern matching to express choice

ForSyDe.Deep currently supports a very limited way of pattern matching that only allows to decompose tuples and bind variables. Additionally case expression for enumeration types are supported. Since pattern matching is a very powerful way of concisely describing behaviour for different inputs, it would be a very good addition to the language for describing circuits.

ForSyDe-XML Backend

Other tools within the ForSyDe framework are using an XML format for passing process network definitions between different stages of the tool flow. In order to make ForSyDe.Deep a proper front-end to the framework a new backend is needed which supports the output of XML instead of VHDL. ForSyDe.Deep currently supports GraphML output, which is similar but not compatible to this XML format.

Xilinx Toolchain support

ForSyDe.Deep currently contains tool drivers for the Altera tools (Quartus and Modelsim) as well as for the open source GHDL simulator. For wider adoption, a tool driver for the Xilinx toolchain would be a good addition.

Allow integration with existing VHDL-IP components

For industrial acceptance it is of utmost importance to support the integration of existing IP into a design. Cλash supports this through the black-box mechanism. A major downside of this approach is the need for a reimplementation of the IP in Haskell. This makes the correctness of verification and simulation dependent on the faithful reimplementation of the IP. For large IP this is difficult to prove. An alternative would be a co-simulation approach with a VHDL simulator, as discussed in the next item.
Interactive co-simulation with VHDL simulators

Co-simulation is the operation of two or more simulators, which are interconnected for exchanging signal activations during simulation. ForSyDe.Deep currently supports co-simulation between a deep and a shallow embedded model. It also supports the simulation of a whole test-bench using a VHDL simulator. A good addition would be the co-simulation of a shallow model and a VHDL model running in an external simulator. This would allow to write much more expressive test cases, as the testing logic could be implemented as a shallow model and would not need to be translatable.

There exists a standard by name of VHDL Procedural Interface (VHPI) for VHDL co-simulation which is not widely supported. Most simulators implement their own co-simulation interface. The Modelsim API that implements this functionality is Foreign Language Interface (FLI).
Bibliography


Online References


List of Tables

2.1 Template Haskell Syntax for quotation and splicing . . . . . . . . . . . . 20

4.1 Classes of argument functions and their estimated implementation com-
plexity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
## List of Figures

2.1 String parsing in different implementations . . . . . . . . . . . . . . . . . . . 12  
2.2 GHC Compilation pipeline [27] . . . . . . . . . . . . . . . . . . . . . . . . . 23  
2.3 Higher order functions map and fold . . . . . . . . . . . . . . . . . . . . . . 29  
2.4 Comparison of internal, explicit and external parallelism . . . . . . . . . . 29  
2.5 Process network for map with external parallelism. . . . . . . . . . . . . . . 30  

3.1 Architecture of the ForSyDe.Deep Compiler . . . . . . . . . . . . . . . . . . 37  
3.2 Architecture of the Cλash Compiler [6] . . . . . . . . . . . . . . . . . . . . . 40
Listings

2.1 Parser statemachine in C++ ........................................ 12
2.2 Parser DSL: Regular Expression ................................. 12
2.3 GHC-Core Term definition [28] ................................. 22
3.1 Definition of a Process Function in ForSyDe.Deep .......... 34
3.2 Definition of a composite process function .................. 34
3.3 Process definition for applying an inner product .......... 34
3.4 Process definition for a shift register ....................... 35
3.5 System definition for a FIR filter ............................. 35
3.6 System definition for a FIR filter ............................. 36
3.7 Cλash-Core Term definition [6] ............................... 39
4.1 Some specialized calls in normal form ....................... 52
4.2 Some higher order function calls which are not in normal form ............................... 52
4.3 Generated code for foldl ........................................ 53
4.4 Lambda Transformation ......................................... 54
4.5 Partial Operator Transformation .............................. 54
4.6 Higher Order Function Normalization ........................ 54
4.7 Partial Operator Transformation with Type Signatures .... 55
4.8 Higher Order Function Normalization with Type Signatures ...... 56
4.9 Removal of Type Signatures .................................. 56
B.1 Example code for FIR filter with higher order functions 71
B.2 Example code for structural description of a FIR filter 72
Appendices
Appendix A

Development Infrastructure

A.1 Continuous Integration

Continuous Integration (CI) aims at making software development more robust and reliable, by requiring developers to integrate their work into the main product as often as possible. To facilitate this, automated infrastructure for running builds and tests of software is often implemented. On such platform is TravisCI [10].

A.1.1 TravisCI

TravisCI is an online service that provides free infrastructure for automated builds and tests for open source projects. It is integrated with github [3] to automatically trigger a build when changes are committed. The sequence of events is as follows:

1. Developer writes code and commits it to his local repository
2. Developer pushes to a public branch on github
3. Github notifies TravisCI through its notification API [4]
4. TravisCI generates the build matrix to determine the number and parameters for all builds
5. TravisCI enqueues the builds determined in the build matrix and waits until a machine is ready to run them
6. For each build:
   
   (a) TravisCI instantiates a clean Virtual Machine (VM) or docker image [1] with their standard configuration
(b) Build script downloads and installs all needed dependencies
(c) Build script builds the software package
(d) Build script runs the test suite of the package (optional)
(e) Build script pushes build-artifacts for release or deployment (optional)
(f) TravisCI stores the build log
(g) TravisCI discards the build environment

7. Github displays the up-to-date build status for all branches

Configuration    The build script is configured through the travis.yml file in the repository root directory [8]

Build matrix    TravisCI determines a build matrix from the configuration file. This is the cartesian product of all orthogonal configuration options given in the configuration. It can be used to build a package using different compiler versions and dependency libraries, in order to test all interesting combinations.

A.1.2 Travis and Haskell

TravisCI provides a default profile for Haskell projects [9] but since they update their build platform on a 6 month schedule, the installed GHC version is often not as recent as desired for the ForSyDe project. There is a project for providing installation packages for all available GHC (pre-)releases [7]. It generates a Travis configuration for a configurable build matrix.

A.1.3 Travis and ForSyDe

The ForSyDe repository has an appropriate configuration file for building with the latest supported GHC versions (starting from 7.10). Additionally the test suite is ran using the GHDL VHDL-simulator [2]. Running Modelsim or Quartus on TravisCI is infeasible as the multi gigabyte toolchain packages would need to be downloaded for each build. As an alternative, a tool driver backend for the GHDL VHDL-simulator was implemented. GHDL is comparatively lightweight and is packaged in the default Ubuntu repository which is available from TravisCI.
Future Work At some point Travis might implement some form of build environment caching, which could make the use of the Altera tool chain feasible. The caching mechanism available on the platform at the time of writing this thesis is neither intended nor practical for this usecase.

Another interesting feature could be to automatically push tagged builds (releases) to Hackage if the build succeeds.

The test suite should be expanded to provide more coverage of the features of ForSyDe.Deep. Additionally the build should include tools for checking Haskell code quality e.g. HLint [6] and test coverage e.g. HPC [5]
Bibliography


Appendix B

Complete Code examples

{-# LANGUAGE TemplateHaskell #-}

module FIR2 where

import ForSyDe.Deep
import qualified Data.Param.FSVec as V
import Data.Param.FSVec ((+>, FSVec, (!))
import Data.Int
import Data.TypeLevel.Num hiding ((*), (+))
import Data.TypeLevel.Num.Aliases
import Data.TypeLevel.Num.Sets

innerProduct = $(newProcFun [d| innerProduct :: FSVec D9 Int8 -> FSVec D9
                                  Int8 -> Int8
                                  innerProduct a b = sum 0 (mult a b)
                                  where
                                      sum :: Int8 -> FSVec D9 Int8 -> Int8
                                      mult :: FSVec D9 Int8 -> FSVec D9
                                         Int8 -> FSVec D9
                                         mult x y = V.zipWith fixmul8 x y ||])
iprodProc inpA inpB = zipWithSY "iprod" innerProduct inpA inpB

shiftreg inp = scanlSY "shiftreg" next init inp
  where
    init = V.copy d9 0
    next = $(newProcFun
             [d| next_state :: FSVec D9 Int8 -> Int8 -> FSVec D9 Int8
                next_state state input = V.shiftr state input ||])
fir_deep :: Signal (FSVec D9 Int8) -> Signal Int8 -> Signal Int8
fir_deep coefs input = iprodProc coefs (shiftreg input)

firSys = newSysDef fir_deep "FIR_thesis" ["c","in"] ["out"]

compileQuartus_sFIR :: IO ()
compileQuartus_sFIR = writeVHDLOps vhdlOps firSys
  where vhdlOps = defaultVHDLops
    quartusOps = QuartusOps
      { action=FullCompilation, 
        fMax=Just 50,  -- in MHz
        fpgaFamiliyDevice=Just ("CycloneII", 
          Just "EP2C35F672C6" ),
        -- Possibility for Pin Assignments
        pinAssigs=[] }

Listing B.1: Example code for FIR filter with higher order functions

{-# LANGUAGE TemplateHaskell , TypeOperators , FlexibleContexts #-}

module FIRdeep (dFIR , dFIRsys ) where

import Data . TypeLevel hiding ((+) , (*), div , (<))
import Data . Int
import Data . Typeable
import Data . Param . FSVec (FSVec , (!) , (+>) , empty)
import qualified Data . Param . FSVec as V
import ForSyDe . Deep hiding ( fir )

macFun :: ProcFun (Int8 -> Int8 -> Int8 -> Int8)
macFun = $(newProcFun [d] 
  macFun :: Int8 -> Int8 -> Int8 -> Int8 
  macFun a b c = a ‘fixmul8‘ b + c
    [] )
macProc :: Signal Int8 -> Signal Int8 -> Signal Int8
          -> Signal Int8
macProc = zipWith3SY "macProc" macFun
macSys :: SysDef (Signal Int8 -> Signal Int8
          -> Signal Int8 -> Signal Int8)
macSys = newSysDef macProc "macSys" ["in1","in2","in3"]
          ["out"]
sliceProc :: Signal (Int8,Int8) -> Signal Int8
          -> Signal (Int8,Int8)
sliceProc inS cf = zipSY "zipProc" dd macOut
where

```haskell```
macOut = instantiate "macIns" macSys dd cf a
(d, a) = unzipSY "unzipProc" inS
dd = delaySY "delProc" 0 d

sliceSys = newSysDef sliceProc "sliceSys" ["in", "cf"]
                  ["out"]

dFIR :: Signal (FSVec D9 Int8)
    -> Signal Int8 -> Signal Int8
dFIR coef inS = snd ( unzipSY "unzipOutput" a8 )

where
  coefS = unzipxSY "coefSig" coef
  mulF = $( newProcFun [d | mulF a b = fixmul8 a b |])
  a0_mul = zipWithSY "a0" mulF inS ( coefS ! d0 )
  a0 = zipSY "zipInput" inS a0_mul
  a1 = instantiate "slice1" sliceSys a0 ( coefS ! d1 )
  a2 = instantiate "slice2" sliceSys a1 ( coefS ! d2 )
  a3 = instantiate "slice3" sliceSys a2 ( coefS ! d3 )
  a4 = instantiate "slice4" sliceSys a3 ( coefS ! d4 )
  a5 = instantiate "slice5" sliceSys a4 ( coefS ! d5 )
  a6 = instantiate "slice6" sliceSys a5 ( coefS ! d6 )
  a7 = instantiate "slice7" sliceSys a6 ( coefS ! d7 )
  a8 = instantiate "slice8" sliceSys a7 ( coefS ! d8 )

dFIRsys = newSysDef dFIR "dFIRsys" ["coef", "in"] ["out"]
```

```haskell```
firSys :: Signal Int8 -> Signal Int8
firSys = dFIR $ constSY "source" $ (V. copy d3 0) V.++ 64 + (V. copy d5 0)

```haskell```
firSysDef :: SysDef (Signal Int8 -> Signal Int8)
firSysDef = newSysDef firSys "firsys" ["in"] ["out"]

```haskell```
adapFirSysDef :: SysDef (Signal (FSVec D9 Int8) -> Signal Int8)
adapFirSysDef = newSysDef dFIR "adapFirsys" ["cfs", "in"] ["out"]

```haskell```
compileQuartus_sFIR :: IO ()
compileQuartus_sFIR = writeVHDLOps vhdlOps adapFirSysDef
where vhdlOps = defaultVHDLOps{execQuartus=Just quartusOps}

  quartusOps = QuartusOps{action=FullCompilation,
                          fMax=Just 50, -- in MHz
                          fpgaFamilyDevice=Just ("CycloneII",
                                                  Just "EP2C35F672C6")}

```
```
pinAssigs=[]
}
writeAndModelsim_sFIR = writeAndModelsimVHDL Nothing firSysDef
[0,0,0,64,0,0,0,0]

main :: IO()
main = do
    putStrLn "$ simulate firSysDef[0,0,0,64,0,0,0,0]"
    putStrLn . show $ simulate firSysDef [0,0,0,64,0,0,0,0]
    putStrLn "$ simulate macSys[10,20,30][30,40,50][15,55,75]"
    putStrLn . show $ simulate macSys [10,20,30] [30,40,50] [15,55,75]
-- putStrLn "$ writeAndModelsim_sFIR"
--msRes <- writeAndModelsim_sFIR
--putStrLn . show $ msRes
    putStrLn "$ compileQuartus_sFIR"
    compileQuartus_sFIR

Listing B.2: Example code for structural description of a FIR filter