Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Baseband Processing Using the Julia Language

Examensarbete utfört i Elektroteknik
vid Tekniska högskolan vid Linköpings universitet
av

Linus Mellberg

LiTH-ISY-EX--14/4812--SE

Linköping 2014

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Baseband Processing Using the Julia Language

Examensarbete utfört i Elektroteknik
vid Tekniska högskolan vid Linköpings universitet
av

Linus Mellberg

LiTH-ISY-EX--14/4812--SE

Handledare: **Antonios Pitarokoilis**
ISY, Linköping University
Niclas Wiberg
Ericsson
Diarmuid Corcoran
Ericsson

Examinator: **Mikael Olofsson**
ISY, Linköpings universitet

Linköping, 14 december 2014

Avdelning, Institution
Division, Department

Communication Systems
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2014-12-14

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
__

ISRN
LiTH-ISY-EX--14/4812--SE

Serietitel och serienummer
Title of series, numbering

ISSN
__

URL för elektronisk version
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-113284

Titel
Title

Baseband Processing Using the Julia Language

Författare
Author

Linus Mellberg

Sammanfattning
Abstract

Baseband processing is an important and computationally heavy part of modern mobile cellular systems. These systems use specialized hardware that has many digital signal processing cores and hardware accelerators. The algorithms that run on these systems are complex and needs to take advantage of this hardware. Developing software for these systems requires domain knowledge about baseband processing and low level programming on parallel real time systems. This thesis investigates if the programming language Julia can be used to implement algorithms for baseband processing in mobile telephony base stations. If it is possible to use a scientific language like Julia to directly implement programs for the special hardware in the base stations it can reduce lead times and costs.

In this thesis a uplink receiver is implemented in Julia. This implementation is written using a domain specific language. This makes it possible to specify a number of transformations that use the metaprogramming capabilities in Julia to transform the uplink receiver such that it is better suited to execute on the hardware described above. This is achieved by transforming the program such that it consists of functions that either can be executed on single digital signal processing cores or hardware accelerators.

It is concluded that Julia seems suited for prototyping baseband processing algorithms. Using metaprogramming to transform a baseband processing algorithm to be better suited for baseband processing hardware is also a feasible approach.

Nyckelord
Keywords

Julia baseband

Abstract

Baseband processing is an important and computationally heavy part of modern mobile cellular systems. These systems use specialized hardware that has many digital signal processing cores and hardware accelerators. The algorithms that run on these systems are complex and needs to take advantage of this hardware. Developing software for these systems requires domain knowledge about baseband processing and low level programming on parallel real time systems. This thesis investigates if the programming language Julia can be used to implement algorithms for baseband processing in mobile telephony base stations. If it is possible to use a scientific language like Julia to directly implement programs for the special hardware in the base stations it can reduce lead times and costs.

In this thesis a uplink receiver is implemented in Julia. This implementation is written using a domain specific language. This makes it possible to specify a number of transformations that use the metaprogramming capabilities in Julia to transform the uplink receiver such that it is better suited to execute on the hardware described above. This is achieved by transforming the program such that it consists of functions that either can be executed on single digital signal processing cores or hardware accelerators.

It is concluded that Julia seems suited for prototyping baseband processing algorithms. Using metaprogramming to transform a baseband processing algorithm to be better suited for baseband processing hardware is also a feasible approach.

Contents

Notation

Abbreviations

Abbreviation	Definition
3GPP	3rd Generation Partnership Project
AST	Abstract Syntax Tree
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DMRS	DeModulation Reference Signal
DSL	Domain-Specific Language
DSP	Digital Signal Processor
FDE	Frequency-Domain Equalization
FFT	Fast Fourier Transform
IDCT	Inverse Discrete Cosine Transform
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
LTE	Long Term Evolution
MRC	Maximal-Ratio Combining
OFDM	Orthogonal Frequency-Division Multiplexing
PRB	Physical Resource Block
SC-FDMA	Single-Carrier Frequency-Division Multiple Access
UE	User Equipment

1 Introduction

Baseband processing is a time critical and computationally heavy part of the modern mobile cellular systems that implement Long Term Evolution (LTE) of the 3rd Generation Partnership Project (3GPP) standard. For efficiency reasons baseband processing is executed on specialized hardware. This hardware is often heterogeneous and every unit has many digital signal processing cores. Each core can run a single job and many cores can operate in parallel and run process tasks simultaneously. The hardware also often has memory hierarchies that differ from normal PCs. It is also frequently equipped with special circuits that are used for certain tasks, for example Fast Fourier Transforms (FFT) or Inverse Fast Fourier Transform (IFFT). These circuits are referred to as (hardware) accelerators. Making the needed baseband processing algorithms run efficiently on these systems requires a lot of effort.

An often used workflow is to first conceive, then implement and finally, test baseband algorithms in a high-level technical language, such as Matlab. The main reason for using technical languages here is that they allow for fast development, it is easy to test changes and to evaluate the performance of the modified algorithm. Technical languages usually also have very good visualization capabilities, which helps when evaluating algorithms. When an algorithm that is good enough has been developed, it is prototyped on real hardware. Since technical languages cannot run directly on the specialized hardware, this usually means that the algorithm is rewritten in a low-level language, such as C. Since the low-level language and the target environment lack a lot of the features of the technical language the translation to C is generally not trivial. The prototype implementation also needs to conform to timing demands set for the algorithms. Common problems are parallelization, use of accelerators and memory management. This means that considerable work has to be done to change the algorithm so that it fits the

specialized hardware. The differences between the high-level and low-level environments are so large that the translation is usually done by different people than those implementing the high-level algorithm. This process makes development of baseband processing systems slow. There is a large time gap, between the algorithm development and the prototyping. This also creates the issue of communicating the algorithm between the high-level and the low-level designers.

The topic of this master thesis is to investigate whether the programming language Julia can be used for prototyping baseband processing algorithms or not and at the same time mitigate some of the issues mentioned above. Julia is a technical language with many features that are similar to those of Matlab. This indicates that it could be used a tool for the first step of the workflow described above. It should provide capability for implementing and evaluating baseband processing algorithms. One part of the thesis is to evaluate if this is true. This is investigated by translating an existing Matlab implementation of a baseband algorithm to Julia.

Another feature of Julia is that it supports reflection and metaprogramming. These features will also be investigated in the thesis. Reflection means that Julia source code can be represented as data structures in Julia. When Julia source code is parsed it is converted into a data structure. This data structure describes a tree of expressions and can be executed. Transforming source code into trees is done during compilation of almost every computer language. The tree representation is called an Abstract Syntax Tree (AST). In Julia the AST can be also be transformed back into source code. Metaprogramming is the process of modifying these data structures, to change a Julia program using Julia itself. Metaprogramming and reflection can be used to shorten the time that is used translating the high-level representation of an algorithm to the low-level representation. The approach is to develop a tool which can transform the code and produce a representation where parallelization and the use of hardware accelerators can be exploited. This tool should provide a number of operations that modify the high-level code so that the code expresses how it can be parallelized and how different parts of the code can be run on hardware accelerators.

2

Background

2.1 Julia

Julia is a high-level programming language for technical computing. It is similar to other languages such as Matlab and Python. It also aims to be fast at general purpose programming. This chapter will give an introduction to the language. It will try to explain strengths and weaknesses of the language focusing on how it can be used for baseband processing. It will also describe some of the differences from other similar languages.

2.1.1 Overview

Julia is designed as a general purpose programming language. This means that it has a wide range of uses and performs well in many different areas of application. The language has support for linear algebra, random number generation, signal processing and string processing. These features are implemented using external third-party libraries that have been in use for a long time and are very efficient. Julia also supports a number of features.

On the project home page [1] the following are given as examples.

- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Dynamic type system: types for documentation, optimization, and dispatch
- Good performance, approaching that of statically-compiled languages like C
- Built-in package manager

- Lisp-like macros and other metaprogramming facilities
- Call Python functions: use the PyCall package
- Call C functions directly: no wrappers or special APIs
- Powerful shell-like capabilities for managing other processes
- Designed for parallelism and distributed computation
- Coroutines: lightweight "green" threading
- User-defined types are as fast and compact as built-ins
- Automatic generation of efficient, specialized code for different argument types
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8
- MIT licensed: free and open source

Julia is influenced by languages used for numerical computing such as Matlab, R and Python. Syntax is similar to these languages and Julia can be used in a similar context. The strength of these languages is expressiveness, they are good for numerical computing because it is easy to write numerical computing code in them. These strengths are incorporated in Julia. It has support for many common numerical operations, such as vectorized expressions (eg. adding vectors and/or matrices) and common functions and operators (eg. fast Fourier transform and matrix multiplication).

Julia is a dynamically typed language, it infers types of variables at run-time, in contrast to a statically typed language where types are known at compile-time. The benefit from this is that it possible to avoid a lot of syntax concerning the types of variables. The drawback is that performance is generally worse than in statically typed languages. Clever use of a Just-In-Time Compiler and the type system (see Section 2.1.4) makes this performance degradation very small in Julia. This makes it possible for Julia to outperform many other dynamically typed languages and it can often come close to the performance of statically typed languages such as C, C++ or Fortran [1].

2.1.2 Syntax

This section will give a short introduction to the syntax used when writing Julia source code. It is not in any way complete but can help to understand later examples in the thesis. A more complete reference can be found in [5].

Variables

Variables are used to store values or objects. Assignment is always done by reference. The following example creates a variable `x`, assigns 1 to it and then assigns the value of `x` to `y`

```
x = 1
y = x
```

Julia is dynamically typed in the sense that variables do not have a type but the values they store have a type. Thus a variable can store values of an arbitrary type and the stored value can be replaced by another value of an another arbitrary type.

```
x = 1    # the variable x stores the value 1 which is an integer
x = 1.0 # x is changed to store 1.0 that is a floating-point number
```

Functions

There are two ways of defining functions.

```
function foo(x, y)
        x*y
end
```

or

```
foo(x,y) = x*y
```

Both of the above creates a function named `foo()` the function takes two arguments and returns the product of the arguments. The body of a function is the code before the closing **`end`** statement. Values can be returned by using the keyword **`return`** if the function executes until the **`end`** statement without encountering a **`return`** statement the functions returns the result of the last expression of the body. To call a function the following syntax is can be used.

```
foo(2,3)
```

In these examples no type has been given for the argument. If this is done a specialization will be created. This can be done as follows.

```
function foo(x, y)
        x*y
end
function foo(x::Integer, y)
    x + y
end
```

The best fitting specialization for the used set of arguments will be used when the function is called. In the case above the sum of the arguments will be returned if the first argument is an integer, in all other cases the product will be returned. This feature is called *Multiple Dispatch* and allows for a very flexible way of introducing new user defined types to the language. Types do not need to be put in a type hierarchy to be used. What is needed is the proper specializations of necessary functions.

Control of Flow

To control the flow of Julia programs the most important constructs are

- **if** – **else** statements
- **for** loops
- **while** loops

The **if** – **else** controls branching. It makes it possible to execute different blocks of code depending on a condition.

```
if condition1
    block1
elseif condition2
    block2
else
    block3
end
```

The above evaluates `condition1` and if it returns `true`, `block1` is evaluated. Otherwise `condition2` is evaluated and if it is `true`, `block2` is evaluated. If `condition2` is also `false` `block3` is evaluated. The **elseif** and **else** blocks can be omitted. There can be arbitrarily many **elseif** blocks, they will be evaluated in order and if all of them fail to execute the **else** block will be evaluated if it is present. The **if** – **else** statement returns the last expression of the evaluated block and thus if the statement only contains an **if** and an **else** it is equivalent to the more terse ternary operator.

```
condition?expression1:expression2
```

This operator evaluates `condition` and evaluates either `expression1` or `expression2` if it is `true` or `false` respectively. The value of the evaluated expression is returned. The ternary operator do not treat the expressions as blocks and thus only single expressions can be used.

The **for** loop is used to loop over a range or array. The syntax is

```
for i in range
    body
end
```

`range` can be an iterator or an array. `i` will be a local variable in the body of the loop, this variable will consecutively take the values of `range` and then the body will be evaluated for each of the values `i`.

while loops are used to run a loop until a condition evaluates to false. The syntax is

```
while condition
    body
end
```

The body of the **while** loop will be executed until `condition` evaluates to false.

Both in the body of **for** and **while** loops the two keywords **break** and **continue** can be used. **break** exits the body of the loop and continues execution right after

the loop. **continue** makes the program continue with the next iteration of the loop as if the end of the loop body was reached.

Iterators and Ranges

An iterator in Julia is any type that has the functions `start()`, `next()` and `done()` defined. Iterators can be used in the range specifications of **for** loops. The idea is to provide a way of iterating over many items without the need to create them in advance. Instead `start()` is called to get the first value, then `next()` produces new values until `done()` returns true. *Ranges* can be used to easily create iterators over a set if integers. They are created using `start:step:stop` where `start`, `step` and `stop` are integers. The step can be omitted and is then defaulted to 1 (`start:stop`). A range is an iterator which starts at `start` in each iteration it is increased by `step`, it stops before a value that is greater than `stop` is produced. As an example to print all numbers between 1 and 10 the following can be used.

```
for i in 1:10
    println(i)
end
```

The function `println()` prints its argument followed by a newline.

Scope of Variables

Variables are introduced either by declaring them using the keywords **local** or **global** or simply by assigning a value to them.

```
local x
global y
z = 1
```

All the above examples declare a variable, in the last example the integer 1 is also assigned to the variable. All variables have a scope. The scope of a variable decides where two variables with the same name will be referring to the same thing. Some constructs in the language introduces scope blocks, these are

- **function** bodies
- **while** loops
- **for** loops
- **try** blocks
- **catch** blocks
- **let** blocks
- **type** blocks

Scope blocks can be nested. When a variable is used scope blocks are searched for the variable. First the current scope block is searched then all enclosing scope A variable declared using the **global** keyword will belong to the global scope.

A variable declared using the **local** keyword will be local to the current scope block. An assignment will introduce a new local variable only if the variable is not found in the current scope, any enclosing scope or in the global scope. In the following example there is only one variable `x`, and `foo()` will return 5.

```
function foo()
    x = 1
    for i in 1:5
        x = i
    end
    x
end
```

In this example there will be two different `x` and `foo()` will return 1

```
function foo()
    x = 1
    for i in 1:5
        local x
        x = i
    end
    x
end
```

Function arguments will always be local to the function body and the same is true for the iteration variable in a **for** loop.

Arrays and Matrices

Arrays are used extensively in the thesis. They are essential to any signal processing application, and one of the most important reason for using numerical languages is that they make it easy to work with these data structures.

To create arrays and matrices brackets `[` and `]` are used. Arrays can have arbitrarily many dimensions and arbitrarily many entries in each dimension. Matrices are simply two-dimensional arrays. When using brackets to create arrays comma (`,`) means concatenation along the first dimension (vertical) and space means concatenation along the second dimension (horizontal).

```
array = [1, 2, 3, 4]
matrix = [[1 2], [3 4]]
```

The code above will create

$$\text{array} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$\text{matrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

The first expression creates a one-dimensional column vector containing 1, 2, 3 and 4 and the second creates a matrix with 1 and 2 in the first row and 3

and 4 in the second. To create arrays of higher dimensionality the function `cat(dim, A...)` can be used. This function concatenates along dimension `dim`. The following is equivalent to the previous example.

```
array = cat(1, 1, 2, 3, 4)
matrix = cat(1, [1 2], [3 4])
```

Another way of creating arrays is a *list comprehension*. It creates data in way that is similar to the mathematical set-builder notation. Below is an example. Assume that the variable `x` holds an array with numbers.

```
y = [t+1 for t in x]
```

The list comprehension creates a new array and assigns it to the variable `y`. The list comprehension consist of two parts, an expression and a range specification. These are separated by the keyword **`for`**. The expression is evaluated for every item in the range specification and the result is used to create a single element in the new array. In the example the new array is created by letting `t` successively take the value of every element in `x`, (**`for t in x`**) and evaluating the expression (`t+1`) to create the elements of the new array. Thus `y` will be a copy of `x` with 1 added to every element. The range specification `x` in the above example can be anything that is iterable and has a length (eg. an array).

Another useful operator in Julia is the splat operator (`...`). This operator is used to call a function with the values stored in an array.

```
a = [1, 2, 3]
foo(a...)
```

The above would be equivalent to

```
foo(1,2,3)
```

Arrays are indexed using brackets. `matrix[2,1]` would retrieve the first element of the second row of the two-dimensional array `matrix`. The colon operator (`:`) can be used to get an entire row or subarrray of a multidimensional array. Ranges can also be used, `matrix[:, 2:4]` would retrieve columns 2 through 4 in the matrix `matrix`.

Many arithmetic operators work as expected when working with arrays and matrices, especially if the programmer has worked with Matlab earlier. For noteworthy differences between Julia and other languages see [3]

2.1.3 Metaprogramming and Reflection

Metaprogramming is the process of writing programs that write or modify other programs. This can be done in many ways, for example a program might simply output source code. Templates in C++ is another example of metaprogramming. Julia has certain features that make it very useful for metaprogramming. The feature that is used in the thesis is called reflection. A language that supports reflection provides some way of representing the structure of a program written in

this language. It is also possible to modify this structure and change the behaviour or structure of the program.

A programming language is always parsed at some point. Compiled languages are parsed during compilation and interpreted languages are parsed at run-time. Some languages, including Julia, make it possible to use the language parser from a running program. When Julia source code is parsed it is converted into a data structure. The type of the data structure is `Expr`, this structure has a property `head` that usually describes an operation and a property `args` which is an array of arguments. An assignment `a = 1` would have the symbol `=` as head and the array `[:a,1]` as its argument. `:a` is the symbol `a` and `1` is the integer 1. The arguments of a `Expr` structure can be another `Expr` structure, in this case the structure forms a tree. `a=1+2` would be represented as before by an `Expr` with the head `=`, the argument would be another `Expr` with the head `+` and arguments `[1,2]`. Transforming source code into trees are done during compilation of almost every computer language, the tree representation is called an Abstract Syntax Tree (AST). In Julia the AST can also be transformed back into source code. This property is called homoiconicity.

Metaprogramming is the process of modifying these data structures, to change a Julia program using Julia itself. Metaprogramming and reflection can be used to shorten the time that is used translating the high-level representation of an algorithm to the low-level representation.

2.1.4 Just-In-Time Compiler

Julia uses the LLVM Compiler Infrastructure [2] to generated executable code. LLVM is a modular framework for producing compliers and compiler toolchains. More specifically Julia uses the Just-In-Time (JIT) compiler of LLVM. A JIT compiler compiles the code of a program at runtime i.e. while it is running.

In Julia the process can be described by the picture in figure 2.1. Source code is first parsed and converted into an internal representation, an AST. Then the JIT compiler is invoked, and it compiles the code at the top level scope of the code that is being executed. Since this is done at runtime it is possible to infer types of many of the variables that are used. This makes it possible for the compiler to generate statically typed machine code without the need for type checks. The compiled code is now executed until it hits a function call. At this time the *multiple dispatch* feature of the language is invoked.

Multiple dispatch means that different versions of a function are invoked depending on the type of the arguments to the function call. Since the compilation is done at runtime argument, types will be known and a version of the function with the types of the call can be compiled and then executed. This continues and each time a call is encountered two things can happen. Either the function with the current set of argument types has been compiled before and can be executed immediately, or the function is compiled and then executed.

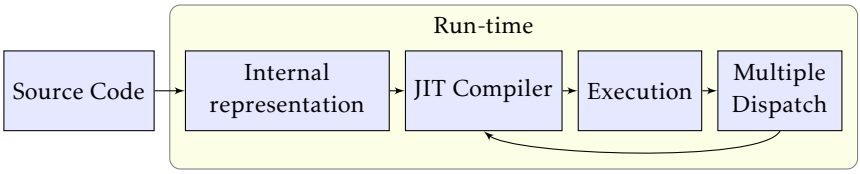


Figure 2.1: *Compilation model of the Julia language.*

2.2 Uplink Receiver Algorithm

The purpose of the uplink receiver algorithm is to provide an example of a baseband processing algorithm that is used in the telecommunications industry. The algorithm is used to demonstrate that program transformation can be used as a tool to ease the development and implementation of real baseband processing systems. This chapter will describe the implemented uplink receiver algorithm. It will not evaluate the performance of the algorithm.

2.2.1 LTE Uplink

The LTE uplink is using Single Carrier Frequency Division Multiple Access (SC-FDMA). This is a technique that is similar to Orthogonal Frequency Division Multiplexing (OFDM), but with better peak to average power ratio. With OFDM data is mapped onto complex frequency domain symbols and then the Inverse Discrete Fourier Transform (IDFT) is used to transform these symbols to the time-domain. A cyclic prefix is added to each symbol by copying the last part of the symbol and inserting it at the beginning. The symbols are then upconverted and sent on a carrier frequency. The frequency domain symbols or subcarriers are grouped into Physical Resource Blocks (PRBs) of 12 consecutive subcarriers. SC-FDMA uses the same technique but the data is modulated to a time-domain sequence that is transformed with a n-point DFT before it is mapped the to frequency domain symbols. The length of this DFT depends on the bandwidth that is scheduled for the transmission. After this OFDM is used to send the data. This extra step spreads data in the frequency domain and lowers peak-to-average power ratio. The process is shown in figure 2.2

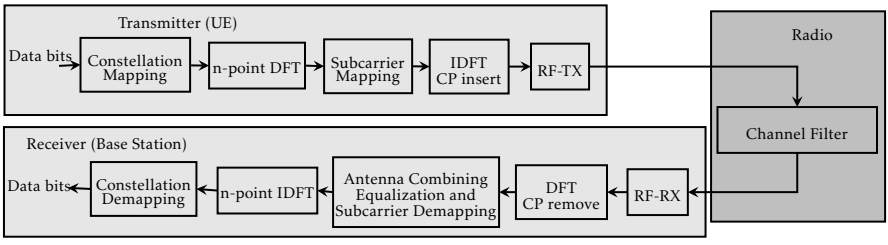


Figure 2.2: *SC-FDMA flowchart*

The LTE standard uses a frame structure to schedule transmission on the radio interface. Each frame is 10 ms long and is divided into 10 subframes each 1 ms long. Each subframe is further divided into two slots. Depending on the length of the cyclic prefix either 6 or 7 symbols are sent in each slot. Normally the shorter cyclic prefix is used and 7 symbols can be sent every slot. Uplink transmissions are scheduled both in time and frequency. User Equipment (UE) is scheduled to transmit on a subframe and a number of PRBs.

LTE also supports multiple antennas to achieve transmission diversity. Base stations use up to four receiving antennas and combine the received signals to reduce noise and channel effects.

2.2.2 Algorithm

The algorithm implements the processing that takes place after down-conversion and the initial cyclic prefix removal and DFT that extracts the frequency domain symbols. It performs channel and noise estimation, antenna combining and equalization, IDFT-despread and demapping. It then delivers a sequence of soft bits. This section describes the uplink receiver algorithm that is used to evaluate Julia for baseband processing. It processes a single user in a single subframe.

Channel estimation

The channel estimation is done using the Demodulation Reference Symbols (DMRS). The DMRS is a sequence defined in the LTE standard [4]. It is parametrized on different configuration parameters and can be calculated both by the UE and the base station. It is sent in symbol 4 in each slot during LTE uplink transmission. The DMRS is the pilot sequence that is used for estimating the channel. A matched filter is used to estimate a channel response for the channel, the channel response is then interpolated using a DFT to remove noise.

The DMRS denoted as Y is extracted from the received signal and the expected DMRS denoted as X is calculated. Both X and Y are column vectors with the lowest frequency subcarrier as the first element and the highest frequency subcarrier as the last element. X is used to create a matched filter that is applied to Y to get a raw frequency response of the channel.

The frequency response is then transformed by applying a DCT and windowed by zeroing all elements outside a window of length M. A IDCT is then applied to get the filtered frequency response. This will result in a smoothing of the channel frequency response.

$$\hat{H} = Y \cdot X^*$$

$$\hat{h} = \mathrm{DCT}[\hat{H}]$$

$$\hat{h}_{wind}(n) = \begin{cases} \hat{h}(n) & 0 \leq n < M \\ 0 & \text{otherwise} \end{cases}$$

$$\hat{H}_{\mathrm{smooth}} = \mathrm{IDCT}[\hat{h}_{\mathrm{wind}}]$$

This is done for all antennas and for both slots in the subframe that is being processed. $\hat{H}_{\text{smooth}}$ will now contain the smoothed estimated channel frequency response.

Noise estimation

The next step is to estimate noise. This is used to do antenna combining and Frequency Domain Equalisation (FDE).

A noise covariance matrix Q_l is estimated for groups of subcarriers. First the noise components of the received signal are estimated as

$$E(k,r) = Y(k,r) - \hat{H}_{\text{smooth}}(k,r)X(k)$$

Here E is a matrix containing noise estimates. $E(k,r)$ denotes the element of E in the kth row and rth column. $Y(k,r)$ and $\hat{H}_{\text{smooth}}(k,r)$ denotes the kth subcarrier from data received at antenna r from the received DMRS and the estimated frequency response, respectively. $X(k)$ denotes the kth subcarrier of the calculated DMRS. For each group of subcarriers l there is a smaller matrix $\tilde{E}_l$ which is constructed by picking elements from E in the following way.

$$\tilde{E}_l(i,r) = E(i + n_g, r)$$

where n_g is the first subcarrier of the group, $i = \{0, ..., N_g - 1\}$ and N_g is the number of subcarriers in the group. This means that E is split vertically into a number of smaller matrices. Q_l is then defined as

$$Q_l = \tilde{E}_l^* \cdot \tilde{E}_l$$

or each group. Q_l will be a square matrix with as many columns and rows as there are antennas.

Maximal-ratio combining (MRC) will be used for diversity combining. This technique only uses the noise energy for each antenna. Hence, only the diagonal elements of Q_l, which corresponds to the energy of the noise, are used. All off-diagonal elements are set to zero.

Antenna combining and channel equalization

This step is adding data from all antennas together using the noise estimates calculated earlier. This is done using Maximal-Ratio Combining (MRC). It also performs Frequency Domain Equalization (FDE) to equalize the channel impulse response. This produces a single array of equalized complex values for each symbol in the subframe.

User IDFT despread

To undo the DFT spread used in SC-FDMA IDFT is performed on the equalized data. The DFT is applied individually for each equalized symbol. Symbols that contain DMRS data are ignored.

Soft demapping

The equalized data can now be demapped to a stream of soft bits. Depending on the modulation order the received symbols are converted to floating point values between -128 and 127 representing the soft bits. The bits are also descrambled using a predefined scrambling sequence according to the LTE 4G 3GPP standard [4]. This is done by flipping the sign of the soft bits according to the scrambling sequence. Flipping a soft bit is done by multiplying it with -1.

2.3 Baseband Processing Platform

In this section a model of a baseband processing platform will be described. The model describes how a baseband processing algorithm is executed on real hardware and focuses on how parallelization is achieved. This model is used to motivate the transformation framework that is described in the thesis.

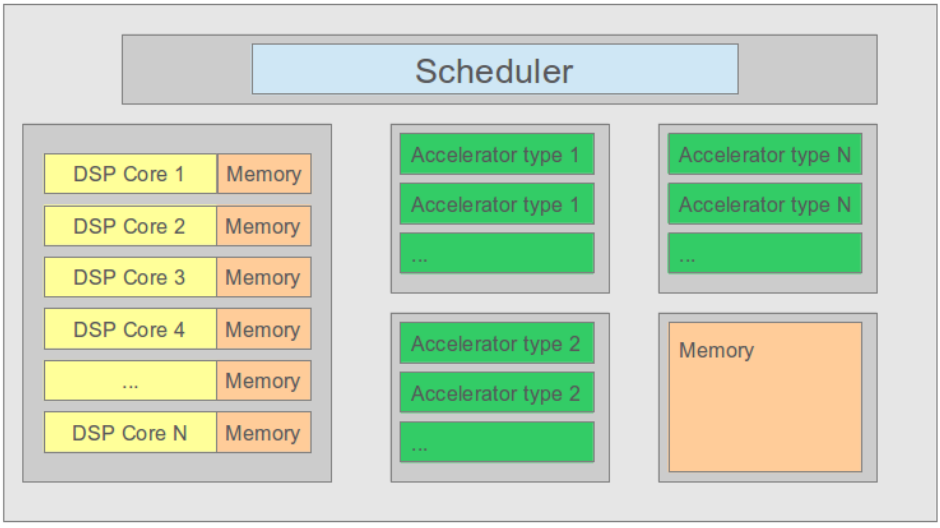


Figure 2.3: *Schematic picture of the baseband processing hardware.*

Figure 2.3 shows a schematic image of the hardware that is assumed in this thesis for baseband processing. The interesting feature of the platform is that it consist of a number of Digital Signal Processing (DSP) cores and a number of hardware accelerators. DSP cores are used for arbitrary calculations and are versatile, they can run different programs depending on what is needed at any time. The platform also has accelerators of different types. An accelerator of a certain type can only perform a certain operation (e.g. FFT), but it does it more efficiently than a DSP core. They are typically used for common and/or computationally intensive operations. A baseband processing algorithm that is executed on the platform needs to be divided into tasks. Each task is a specific job that is executed on either a DSP core or an accelerator. A task is always executed to completion. It is never

suspended and resumed later. Tasks are scheduled based on which other tasks have been completed. It is assumed that when a task starts all data that is needed for that task to run has already been computed.

To create a baseband processing algorithm that can run on this platform two things are needed. A number of tasks need to be specified and a schedule must be determined which describes when each task can be executed. The schedule consists of dependencies. When a task is dependent on another task it means that the dependent task can not run until the other task is completed. The model does not specify the number of DSP cores or accelerators, this number may vary depending on the specific platform and it is preferable to be able to easily adapt to changes in these numbers.

3

Problem formulation

The problem that is being investigated in this thesis is whether Julia can be used to incrase efficiency of baseband processing algorithm development or not. This is a very wide subject and needs to be significantly narrowed down. The focus will be on using metaprogramming in Julia to aid the development of baseband algorithms. The assumption is that there are many ways of writing a program that executes a given algorithm, but only some of these programs are easy to read and understand by algorithm developers. From experience at Ericsson, programs that implement baseband algorithm in a way that is suitable for the hardware described in section 2.3 are generally not easy to read.[6] The thesis investigates if it is possible that some of the easily readable programs can be *transformed* into programs adapted to the hardware platform. The method that will be considered for transforming a program will be to use tools developed in Julia (as part of the thesis work) that can take a program as input along with directions on how to transform it and output a new transformed program.

Figure 3.1 shows the approach. The arrow in the figure shows how a program is refactored and transformed. The large rectangle represents the set of all different Julia programs that implement a particular baseband processing algorithm. The left rectangle is a subset of all these Julia programs but it only contains those that are *well structured*. The property of a program being well structured is subjective and will depend on the developer that is reading the source code of the program. In this case the viewpoint is that of the algorithm developer. The program should be easy to understand and read for an algorithm developer who has little knowledge of how the target platform works. The right rectangle represents Julia programs that are adapted to the target platform, this means that the program specifies how it is divided into tasks and how these tasks are dependent on each other. These programs will generally not be considered to be *well*

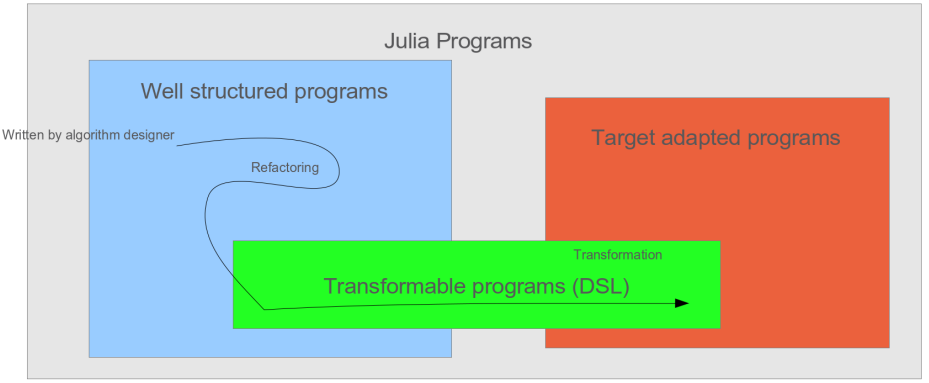


Figure 3.1: *Figure showing how to adapt a program to the hardware platform.*

structured or easy to read by a developer. The middle rectangle represents the set of Julia programs that are written in such a way that they can be transformed using the the tools developed during the thesis work. The programs that can be transformed will need to conform to a set of rules that are specified in section 4.2.

The reason for using the approach described above is to achieve the following way of working with baseband development. The baseband algorithm is first implemented in Julia by an algorithm developer. This is well structured in the sense that it is written by a developer with very few limitations, thus it is in a form that this developer thinks is suitable for the algorithm and *well structured*. When this has been done the program is refactored, which means that it is rewritten by a developer. This refactoring can be done incrementally with small changes at a time and the program can be well structured at all times. The goal of the refactoring is to make the program transformable. It is also possible that an algorithm developer writes programs that are transformable and no refactoring is needed. When a transformable program has been created the program can be transformed to a target-adapted form, that will describe how the algorithm should run on the target platform.

The problem that is investigated in this thesis is if the above can be achieved. To do this the following questions needs to be answered.

- Can Julia be used to implement baseband processing algorithms in a well structured way?
- How can a target-adapted program be described using Julia?
- Is it possible to transform a well structured program to the target-adapted form?
- Which transformations are needed to do this?

- What are the rules that are needed to be able to transform a program?

4

Julia on the Hardware Model and a Baseband Processing DSL

This chapter will describe a way to express how a Julia program is executed on the hardware model described in section 2.3. It describes how to structure a Julia program to show how it is divided into tasks and how it is parallelized. The program will also be an executable Julia program. The chapter will also describe a Domain Specific Language (DSL) that specifies which Julia programs are transformable by the transformations in chapter 6.

4.1 Julia on the Hardware Model

The hardware model described in section 2.3 is assumed to have a runtime environment. The runtime environment decides which tasks to run at a certain time. A task is a processing operation or a series of operations that are run on a single processing core or hardware accelerator. Tasks can be run in parallel as long as there is a processing core or accelerator that currently is not running another task.

The hardware is assumed to have an arbitrary number of cores. Each core can run one task at a time and tasks are always executed until completion, i.e. they can not be suspended. The hardware also has accelerators for running FFT and IFFT, hardware accelerators are used to efficiently compute these operations.

From the perspective of the runtime environment, an application in hardware-adapted form is a Julia function definition, but only with a very limited set of operations allowed. This function will be referred to as a top-level function. The top-level function should describe which tasks that should be run and in what order. In the thesis the only tasks that are run on an accelerator are fast Fourier transforms and inverse fast Fourier transforms. Running these tasks are represented by a call to `fft()` and `ifft()` respectively. All other tasks are run

on a signal processing core and are represented by a call to a function. Functions that are called must also have a function definition that defines what the task does. These function definitions are ordinary Julia function definitions.

To describe parallelization list comprehensions are used. The expression in a list comprehension in the top-level function should be function call that executes a task either on an accelerator or processing core. This task can be executed in parallel for each of the items in the range specification.

The top-level function also specifies in which order tasks can be executed. The order is specified by assignment of variables. A variable is assigned when the task or list comprehension on the right hand side of the assignment has finished. A task can be executed as soon as all variables that are used as arguments to the call have been assigned. To make sure that the top-level function also can be run in a regular Julia runtime environment it is also required that calls that use a variable, appear after they have been assigned.

Below is a simple example of a top-level function. `task1()` and `task2()` are assumed to be tasks that are defined elsewhere. These examples have an input variable `data`, the type of this variable is not specified but should be an array since the `fft()` task would fail otherwise. `task1()` and `fft()` can be run immediately since they only need the variable `data` to run. When both tasks are finished `tmp1` and `tmp2` have been assigned and `task2` can run.

```
function algorithm(data)
        tmp1 = task1(data)
        tmp2 = fft(data)
        result = task2(tmp1,tmp2)
end
```

The thesis focuses on transforming baseband algorithm to fit the parallel runtime environment better. An example of a baseband processing algorithm that can be transformed is the following.

```
function foo(data)
    tmp = [x * x for x in data]
    result = fft(tmp)
    result
end
function main(data)
    out = foo(data)
    out
end
```

The top-level function is `main()` it calls a single task `foo()`. `foo()` receives an array of data, then for each element in the array multiplies it with itself and creates a new array of the results. Then a call to `fft()` is used to calculate a discrete fourier transform of the new array. Finally the result from the call to `fft()` is returned from the function. The call to `fft()` appears inside a task and thus it can not be executed on an accelerator, only the top-level function can

execute tasks on accelerators. The calculation of `tmp` could be parallelized. The above could be transformed to the following.

```
function bar(data)
        tmp = x * x
        tmp
end
function main(data)
        tmp = [bar(x) for x in data]
        out = fft(tmp)
        out
end
```

The `foo()` task has been broken into two tasks, `bar()` and an `fft()` accelerator task. The `bar()` task has been parallelized so that each calculation of the **`for`**-loop the elements of `tmp` is in a separate task. How the transformation can be performed is described in detail in chapter 6.

This chapter describes a framework of simple transformations that can be applied to achieve the demonstrated transformation and other transformations that can be used to adapt a baseband processing algorithm to hardware. The framework is implemented as a tool written in Julia that can be used to transform baseband processing algorithms written in the specified DSL.

4.2 Baseband Processing DSL

The transformations described in this chapter require the code to be transformed to comply to a Domain Specific Language (DSL). This section will describe this DSL.

The runtime environment and the tasks of the baseband processing algorithm are described using an embedded Domain Specific Language (DSL). A DSL is a language that is used to efficiently solve problems in a specific domain. In this case it enables the description of the runtime environment of baseband processing hardware. It also allows to execute a number of transformations that make the execution more efficient on baseband processing hardware. Embedded means that the DSL is a subset of another language with no features that are not already present in that language. In this case an algorithm that is written using the DSL will be executable Julia source code, even though all features of Julia will not be allowed. DSLs are often used in similar situations where they can simplify operations in a specific problem domain.

The chosen DSL needs to be useful for the problem domain. This means that it needs to be able to express baseband processing algorithms. It also needs to be strict enough to make it possible to perform transformations of the code. This approach can also loosely be viewed as a kind of Model Based Design paradigm. The DSL provides the building blocks to model a baseband algorithm. When this model is built it can then be transformed to a form which is suited for hetero-

geneous hardware. This approach makes it considerably easier to address the problems with parallelization and the use of hardware accelerators.

Baseband processing consists of operations on data. Every operation takes some data performs its task on it and outputs new data. This leads to the decision that the algorithm is restricted to be written in a purely functional manner. Purely functional means that the code has two properties, variables are immutable and functions do not have side-effects or state. Immutable variables can only be written once, when they are defined. Functions that do not have state or side-effects will always yield the exact same result when called with the same arguments. This will create code that is in Single Static Assignment form, which is commonly used for internal representation of code in compilers. These restrictions make it easy to follow how data flows in the algorithm, from right to left in assignments. It is also possible to make assumptions on the ordering of operations. When this form is used with stateless functions, all expressions can be evaluated as soon as the variables in the expression are defined. To be able to perform transformations on code it needs to conform to the restrictions described above. Although code not directly affected by the transformations is allowed to use mutable variables. This means, for example, that functions that are called from transformed functions can use mutable variables.

These restrictions were chosen because they simplify the analysis of the code. When these rules are applied almost every line of code becomes an assignment where one or more variables are used in a function or operation and the result is saved in one or more new variables. This structure resembles how baseband processing algorithms are described. This also makes it natural to describe operations that are performed on lots of data with list comprehensions. This is also a convenient way to describe parallelized operations.

These rules have some important implications. A major one is that `for` loops become unusable. Since variables are immutable and functions do not have side-effects `for` loops can not do anything. Only loop local variables can be written and those values will not be visible outside the loop body. This also applies to values returned by functions, they can not be propagated to the outer scope. The same thing is true for `while` loops and other constructs that create a local scope. The exception is `if`–`else` statements and this is because they can return values in Julia. The natural replacement for looping constructs is the list comprehension. This might seem like a big limitation but there also are a lot of benefits. List comprehensions are expressive and it is often straight-forward to replace `for` loops with them. This is especially true when the operation is *data local*, that is the operation is a mapping from one array to another and the operation only uses one element of the input array to produce the corresponding element in the output array. A case where a limitation can occur is when input and output data have different sizes or when the relation between input and output elements is more complex, such as when some form of permutation is applied.

The choice of this DSL has two major benefits. First it makes the code easier to analyse. As described above a lot of assumptions can be made that are not possible

in arbitrary Julia code. It is easy to follow data flows and there is more freedom in the ordering of expressions. The other benefit is that it encourages the designer to write code that is parallelizable. List comprehensions are parallelizable and they make it easy to spot parallizable operations. Things that are difficult to express using list comprehensions are usually more difficult to parallelize. Encouraging list comprehensions will thus encourage the designer to write algorithms that are easy to parallelize.

5

Implementation of Baseband Processing Algorithm

A baseband processing algorithm has been implemented in Julia. An implementation written in Matlab was used as reference. The reference implementation is used internally at Ericsson. The algorithm that is implemented is described in section 2.2. The algorithm has been verified against the reference implementation and produces the same output on a set of testcases. This chapter will show some examples of what the baseband processing algorithm implementation looks like. It will focus on the part of the algorithm that estimates the channel impulse response. This is where most work with transformations has been done.

5.1 Design Considerations

The uplink receiver algorithm is implemented as a Julia module. The purpose of the module is to be a high-level implementation of a real baseband processing algorithm. It is written in a way that is natural for the Julia language and the problem domain. The algorithm is divided into functions that group the different part of the algorithm together. The purpose of these functions is to structure the algorithm in a way that makes it easy to read, maintain and develop. It is also written using pure functions, which means that functions do not have side effects or store state. This makes it easier to follow the different data flows that are present.

5.2 Implementation

The Julia module that implements the algorithm exposes a single function `PUSCH_receiver(rxPRBs, ueParams)`. The parameter `rxPRBs` should be a

3-dimensional array with the dimensions subcarrier, symbol and antenna. The data should be an entire received subframe. The initial FFT that is done when receiving LTE uplink data should already have been performed. The algorithm expects the array to contain samples for four antennas and 14 symbols. This means that for example `rxPRBs[1,2,3]` will contain data for the first subcarrier of the second symbol received on antenna number 3. The parameter `ueParams` contains configuration parameters for the received subframe, this includes for example which resource blocks to decode, the used coding scheme and the window length that will be used in the smoothing of the frequency response. The implementation of `PUSCH_receiver()` is shown in listing 1.

```
export PUSCH_receiver
function PUSCH_receiver(rxPRBs, ueParams)
  # Matched filter and smoothing
  frequency_response =
    estimate_frequency_response(rxPRBs,
                                ueParams)

  # Calculate noise covariances
  Q_tilde_inverse =
    calculate_noise_covariances(rxPRBs,
                                frequency_response,
                                ueParams)

  # Antenna combining and FDE
  Rx_data_combined, beta, lambda =
    antenna_combining_and_FDE(rxPRBs,
                              frequency_response,
                              Q_tilde_inverse,
                              ueParams)

  # Per-user IDFT despreading
  Rx_data_ifft = IDFT_despread(Rx_data_combined)

  # Soft-bits demapping
  Rx_soft_bits = soft_bits_demapping(Rx_data_ifft,
                                     beta,
                                     lambda,
                                     ueParams)

  Rx_soft_bits

end
```

Listing 1: *Implementation of PUSCH_receiver().*

The function is split into five functions. `estimate_frequency_response()` uses the DMRS of the received subframe to estimate a frequency response that is then smoothed to minimize noise. Then the received noise is estimated in

`calculate_noise_covariances()` these estimates is then used in `antenna_combining_and_FDE()` to merge the data from the four antennas into one signal and then equalize it. Next is the `IDFT_despread()` which performs an IDFT and last is `soft_bits_demapping()` which transforms the estimated symbols to soft bits. When all this has been performed a single one-dimensional array containing floating point values between -128 and 127 is returned, this corresponds to the transport block received in the subframe.

The function `estimate_frequency_response()` is implemented as in listing 2.

```
function estimate_frequency_response(rxPRBs, ueParams)
  # Get received DMRS
  rxDMRS = DMRS_extraction(rxPRBs, ueParams)

  # Calculate DMRS
  trueDMRS = calculate_DMRS(ueParams)

  # Get a raw frequency response
  raw_frequency_response =
    calculate_raw_frequency_response(rxDMRS, trueDMRS, ueParams)

  # Calculate h_hat using DCT
  h_hat = calculate_dct(raw_frequency_response)

  # Window h_hat
  h_hat_opt = window(h_hat, ueParams)

  # Calculate H_hat_opt using idct
  H_hat_opt = calculate_idct(h_hat_opt)

  H_hat_opt
end
```

Listing 2: *Implementation of estimate_frequency_response().*

This function consists of six functions. `DMRS_extraction()` uses information in `ueParams` to extract the DMRS symbols from `rxPRBs`. `calculate_DMRS()` calculate the expected DMRS, this is done according to 3GPP LTE specification [4]. Next a raw frequency response is calculated. This is simply a matched filter created by the expected DMRS and applied on the received DMRS. The matched filter is applied on all received DMRS symbols. It is implemented as shown in listing 3.

When the raw frequency response has been calculated it is smoothed. This is done by applying a DCT windowing and then applying an IDCT. The DCT is applied using `calculate_dct()` which applies the function `dct()` on each raw frequency response. `dct()` then does certain preprocessing and then uses the built-in function `fft()` to calculate the DFT. Then there is a postprocessing

```
function calculate_raw_frequency_response(rxDMRS, trueDMRS, ueParams)
  MF_DMRS_chan_est =
    [rxDMRS[subcarrier,slot,antenna]*conj(trueDMRS[subcarrier,slot])
      for subcarrier in 1:(ueParams.nPRBs*12),
          slot in 1:2,
          antenna in 1:GLOBAL_PAR.nRx]
  MF_DMRS_chan_est
end
```

Listing 3: *Implementation of calculate_raw_frequency_response().*

step to produce the DCT. These are implemented as shown in listing 4 and 5. In `calculate_dct()` the call to `dct` is done using a list comprehension as needed by the DSL described in 4.2. This is to be able to do transformations on this function.

```
function calculate_dct(H_hat)
  h_hat = cat(3, [cat(2, [dct(H_hat[:,slot,antenna])
    for slot in 1:2]...) for antenna in 1:GLOBAL_PAR.nRx]...)
  h_hat
end
```

Listing 4: *Implementation of calculate_dct().*

The windowing is then done in `window()`, the implementation can be found in listing 6. This function reads the parameter `windowlength` from `ueParams` and returns a copy of the input array `h_hat` where every element after the `windowlengt`th element is set to zero. This function is also written using a list comprehension, but this time with a helper function. This makes the function comply to the DSL defined in section 4.2.

After windowing is performed the IDCT is calculated using the function `calculate_IDCT()`. This is done like the calculation of the DCT, first preprocessing, then using the built-in `ifft()` and then postprocessing. It should be noted that the postprocessing step assumes that the window used in the previous step is shorter than half the length of the array. The postprocessing will not be correct unless the second half of the signal is zero. The implementation can be seen in listing 7 and 8.

```
## Calculates DCT for H_hat.
function dct(H_hat)
  v = dct_preproc(H_hat)

  # Calculate FFT of rearranged H_hat and extract the
  # DCT by applying correction factors.
  V_unscaled = fft(v)

  h_hat = dct_postproc(V_unscaled)
  h_hat
end

function dct_preproc(H_hat)
  N1 = size(H_hat,1)
  # Rearrange H_hat so that FFT can be used to calculate DCT.
  v = [Complex128[H_hat[i] for i in 1:2:N1],
       Complex128[H_hat[i] for i in N1:-2:2]]
  v
end

function dct_postproc(V_unscaled)
  # Compute correction factor R.
  N2 = size(V_unscaled,1)
  V = V_unscaled./sqrt(N2)
  R = [exp(-im*pi*i/(2*N2))/sqrt(2) for i in 0:N2-1]

  h_hat = [V[1], Complex128[V[i]*R[i] + V[N2-i+2]*conj(R[i])
                            for i in 2:N2]]
  h_hat
end
```

Listing 5: *Implementation of dct().*

```
function threshold(threshold, value, out)
    value <= threshold ? out : 0
end

function window(h_hat, ueParams)
    nPRBs = ueParams.nPRBs
    nSc = nPRBs*12
    windowlength = ueParams.windowlength

    L_opt = [windowlength for slot in 1:2]
    h_hat_opt =
      Complex128[threshold(L_opt[slot], sc, h_hat[sc, slot, antenna])
                 for sc in 1:nSc,
                     slot in 1:2,
                     antenna in 1:GLOBAL_PAR.nRx]
    h_hat_opt
end
```

Listing 6: *Implementation of window().*

```
function calculate_idct(h_hat)
    H_hat = cat(3, [cat(2, [idct(h_hat[:,slot,antenna])
      for slot in 1:2]...) for antenna in 1:GLOBAL_PAR.nRx]...)
    H_hat
end
```

Listing 7: *Implementation of calculate_idct().*

```
## Calculates IDCT.
#  Expects at least second half of DCT to be zero!
function idct(W)
  Y = idct_preproc(W)
  y_unscaled = ifft(Y)

  x = idct_postproc(y_unscaled)
  x
end

function idct_preproc(W)
  N1 = size(W ,1)
  R = [exp(-im*pi*i/(2*N1))/sqrt(2) for i in 0:div(N1,2)-1]

  # Rearrange and use IFFT for IDCT calculation
  Y = [W[1],
       Complex128[W[i]*conj(R[i]) for i in 2:div(N1,2)],
       W[div(N1,2)+1],
       Complex128[W[i]*R[i] for i in div(N1,2):-1:2]]
  Y
end

function idct_postproc(y_unscaled)
  N2 = size(y_unscaled ,1)
  y = y_unscaled*sqrt(N2)
  # Reorder IFFT to obtain correct IDCT
  x = Complex128[y[i%2 == 1 ? div(i,2)+1 : N2-div(i,2)+1]
    for i in 1:N2]
  x
end
```

Listing 8: *Implementation of idct().*

6

Code Transformations

Transforming code is an important issue in this thesis. Programs that run on signal processing hardware need to be adapted to the architecture of the hardware. The algorithms used for baseband processing are usually not straight forward to implement on hardware. There are two major concerns, firstly the algorithms need to run in parallel when possible and secondly it is required to use hardware accelerators for bottlenecks in the algorithms. This is a problem since code that handles these issues often is more complicated and harder to work with. In this chapter a framework for adapting Julia code to be able to handle these concerns will be presented. The strategy is to expose parallelism to the algorithm developer in an early stage in the development cycle and in a way that makes it possible to describe it in a high level language. This is done by specifying a DSL and providing a framework of code transformations that can be applied to code written in the DSL.

6.1 Transformations

This section will describe a number of transformations. These transformations provide a way to make changes to a program written in the DSL described in Section 4.1. The transformations make it possible to modify the program in such a way that parallelization is done in the top-level function. In the target runtime environments this means that the task can be parallelized and there will be better utilization of the available processing cores. The transformations also provide a way to split tasks into many small tasks and to join many tasks into a larger task. This can be used to alter the structure of the program and how it is divided into tasks. This is used to move operations that can be performed on a hardware accelerator e.g. FFT to the top-level function. The runtime environment can thus

schedule them to run them on accelerators.

6.1.1 Inline function

Inlining will remove a function call and replace it with equivalent code in place. Instead of calling the function all operations of the function will be performed in the calling function. This transformation is performed by copying the body of the called function. Then all references to the arguments of the function are replaced by the variables used in the call to the function. Then the new function body is put just before the previous call to the function. Lastly the call to the function is replaced by the return value of the function. Below is a minimal example.

Before transformation:

```
function toplevel(x,y)
    out = f(x,y)
end
function f(a, b)
    c = a + b
    c
end
```

After inlining at `z`

```
function toplevel(x,y)
    f_c = x + y
    out = f_c
end
```

To prevent any problem with naming of variables, local variables at the functions top scope will also be renamed by prepending the variable name with the name of the inlined function. The DSL specifies that functions do not have side effects, thus every function returns a value or is useless. To perform the inline transformation on the example above the following call could be used.

```
inline(m, :toplevel, :out, "f")
```

Here it is assumed that `m` contains the AST of the program before the transformation. `:toplevel` and `:out` points out which function-call to inline. It will look for an assignment of the variable `out` in the function `toplevel` and inline the call in the right hand side of this assignment. The string `"f"` will be prepended to all local variables in the inlined function. The call will return a transformed AST.

6.1.2 Extract function

This is the inverse operation of *inlining*. It takes a block of code and creates a function and a function call that is equivalent. To specify what to extract the input variables and the output variables need to be specified along with the target function where the extraction should be performed. A name of the generated function should also be supplied. To perform this transformation the target function is analysed to find all variables that both depend on at least one of the input

variables and has at least one of the output variables as a dependant. Here dependency means that if `b` is dependent on `a` then `a` needs to be assigned before `b`. Below is a minimal example.

Before transformation:

```
function toplevel(x,y)
    f_c = x + y
    out = f_c
    out
end
```

After extraction as `foo` from `x,y` to `out`

```
function toplevel(x,y)
    out = foo(x,y)
end
function foo(x,y)
    f_c = x + y
    out = f_c
    out
end
```

This transformation would be performed by the following call.

```
extractfunction(m, :toplevel, (:out,), (:x, :y), "foo")
```

As before `m` contains the AST and `:toplevel` points out which function to do the transformation in. `(:out)` is a list of the output- variables of the new function. `(:x, :y)` is a list of input-variables to the new function. The string `"foo"` will be the name of the new function.

6.1.3 Push Vectorization

This operation moves the level that a vectorization is performed on. A function that adds the absolute value of two numbers can be defined as follows.

```
function addabs(a, b)
        abs_a = abs(a)
        abs_b = abs(b)
        result = abs_a + abs_b
        result
end
```

If this function is used to add the absolute values for every element of two vectors we could do `[addabs(a[i],b[i]) for i in 1:length(a)]`. An alternative would be to use another function

```
function addabspushed(a, b, rng)
        abs_a = [abs(a[i]) for i in rng]
        abs_b = [abs(b[i]) for i in rng]
        result = [abs_a[i] + abs_b[i] for i in rng]
end
```

This function would be called with `addabspushed(a, b, 1:length(a))` instead of the list comprehension used earlier. Here the vectorization has been pushed down to the called function. This is what *push vectorization* does. It replaces a list comprehension which has a single call and iterates over a range, by a call to a new function. The new function is a copy of the function that was called in the original list comprehension but the operations are done in list comprehensions. This also increases the dimensionality of many variables. In the example all variables in `addabs()` are scalars, but in `addabspushed()` they are arrays. `addabspushed()` also has an additional variable `rng` which specifies the range that is being iterated over in each list comprehension in `addabspushed()`.

The benefit from this operation is that the *inline* transformation can be used to inline `addabspushed()`, but it can not be used to inline the call to `addabs()` in the *list comprehension*. To be able to inline a call in a *list comprehension* *push vectorization* is first used to make this possible. A *push vectorization* transformation will always be followed by an inline transformation and is used to handle inlining of more complicated expressions.

A simple example that describes how it works is the following

```
function toplevel(a)
    out = [foo(a[i]) for i in 1:10]
end
function foo(a)
    result = a*a
    result
end
```

Where pushing the first dimension of the target `out` creates

```
function toplevel(a)
    out = foopushed(a,1:10)
end
function foopushed(a,range1)
    result = cat(1,[a[i] * a[i] for i = range1])
    result
end
```

This example shows the functionality of the *push vectorization* transformation. The transformation removes the vectorized operation of calling `foo()` for all elements of the vector `a` and replaces this by a single function call to the new function `foopushed()`. This function takes two parameters, the vector `a` and a range to iterate over. The function then performs the same operation as the original function `foo()` for every element in `a`, it then returns the array that was produced. This is achieved by looking at the index variable `i` in this case. We can see that in the call to `foo() a` is dereferenced using `i`, this is saved as a *symbol that needs to be vectorized*. Then the body of the function `foo()` is copied into the new function `foopushed()`. The parameters of `foopushed()` is the same as for `foo()` but with the addition of a parameter to hold the range that was iterated over. Then the body of `foopushed()` is analysed. Every expression

that has a reference to a *symbol that needs to be vectorized* in it is enclosed in an appropriate *list comprehension* and the references to all vectorized symbols are replaced by a suitable array dereference of that symbol. Since the variable that has been assigned the new expression now will have higher dimensionality than before it will also be considered a *symbol that needs to be vectorized*. In the example this will be the variable `result` which is an array in `foopushed()` instead of a scalar as in `foo()`.

Push vectorization can also be used on list comprehensions that create multidimensional arrays. In this case either all dimensions can be pushed or just a subset of them. Pushing a subset of dimensions of a multidimensional array requires that the array is assembled in the correct way. To demonstrate consider the following example. `a` is assumed to be a 2-dimensional array with 10 elements in each dimension.

```
function toplevel(a)
    out = [foo(a[i,j]) for i in 1:10, j in 1:10]
end
function foo(a)
    result = a*a
    result
end
```

If the first dimension of the list comprehension is pushed the following would be produced.

```
function toplevel(a)
    out = cat(2,[foopushed(a[:,j],1:10,j) for j = 1:10]...)
end
function foopushed(a,range1,j)
    result = cat(1,[a[i] * a[i] for i = range1])
    result
end
```

Here there are two calls to `cat()` that need to be explained. They are used to keep the dimensionality of the array. Each call to `foopushed()` returns a 1-dimensional array of the data it has computed. The list comprehension in `toplevel()` creates an array of the items that are returned from `foopushed`, this means that the list comprehension will return an array of arrays. This is the reason for using `cat()`. The elements of the returned array of arrays is concatenated along dimension 2. This will create the expected 2-dimensional array that the list comprehension in the original `toplevel()` function did before the transformation.

This transformation would be applied by the following call

```
pushvectorization(m, :toplevel, :out, :foopushed, (1,), 2)
```

`m` should contain the AST to be transformed. `:toplevel` and `:out` are used to pick out the list comprehension to transform, just like in *inline*. `:foopushed` is the name of the new function. `(1,)` is a list of dimensions to push and `2` is the

total number of dimensions in the variable that is used in the list comprehension.

6.1.4 Pull vectorization

Pull vectorization supply the inverse functionality of *push vectorization*. It moves parallelization upwards in the function hierarchy of the algorithm. A similar example as used in *push vectorization* can be used to demonstrate.

```
function toplevel(a)
    out = foo(a,1:10)
end
function foo(a,range1)
    result = [a[i] * a[i] for i = range1]
    result
end
```

This example shows a top-level function that executes a task `foo()` with the an array `a`. This task calculates the square of each element in `a` and returns the result. A *pull vectorization* operation changes this into

```
function toplevel(a)
    out = cat(1,[foopulled(a[i],i) for i = 1:10])
end
function foopulled(a,i)
    result = a * a
    result
end
```

Here the task `foo()` is replaced by the task `foopulled()`. The difference is that before the transformation the task `foo()` was called once and did all calculations on a single core. After the transformation the `foopulled()` is called for every element in `a` since each call will create a task this can be run in parallel on many cores.

To apply this transformation the following would be used

```
pullvectorization(m, :toplevel, :out, :foopulled, (:a,), (1,), 1)
```

The arguments are almost the same as for *push vectorization*. `m` is the AST, `:toplevel` and `:out` picks the call to transform. The difference is `(:a)` which is a list of arrays that should be affected by the transformation. `(1,)` and `1` is the dimensions to be pulled and the number of dimensions in the affected arrays, respectively.

The *pull vectorization* transformation can also be used on a task that contains multiple expressions. The limitation here is that all expressions need to have the same range specification in each list comprehension that is affected by the transformation. It should also be noted that expressions that are not affected by the transformation, for example scalar operations, will be duplicated and executed in every call.

6.1.5 Split vectorization

The above transformation *pull vectorization* can be used to create a vectorized task for a certain operation. It will create one task for each element in the dimension that has been vectorized. This might sometimes create too many tasks. Since the number of DSP cores is limited and there is overhead when starting new tasks to many tasks is sometimes undesirable. The *split vectorization* transformation is used to control the number of tasks created by a vectorized call. This is achieved by replacing the vectorized tasks with (fewer) new tasks that each executes many of the vectorized operations.

The result of the example in section 6.1.4 can be used. This example will create ten tasks.

```
function toplevel(a)
    out = cat(1,[foopulled(a[i],i) for i = 1:10])
end
function foopulled(a,i)
    result = a * a
    result
end
```

This can be split into the following

```
function toplevel(a)
    out = cat(1,[foosplit(a,bundle,5,1,10) for bundle = 1:5]...)
end
function foopulled(a,i)
   result = a * a
   result
end
function foosplit(a,bundle,nBundles,start,stop)
    bundlesize = iceil(((stop - start) + 1) / nBundles)
    from = 1 + (bundle - 1) * bundlesize
    to = min(stop,bundle * bundlesize)
    result = cat(1,[foopulled(a[i],i) for i = from:to]...)
    result
end
```

Above the vectorized call to `foopulled()` has been moved to `foosplit()` which is vectorized instead. `foosplit()` is executed in five tasks and each task will call `foopulled()` two times. In this way the number of tasks has been reduced to half of what it was before.

To apply this transformation the following is used.

```
splitvectorization(m, :toplevel, :out, "foosplit", 5, 1)
```

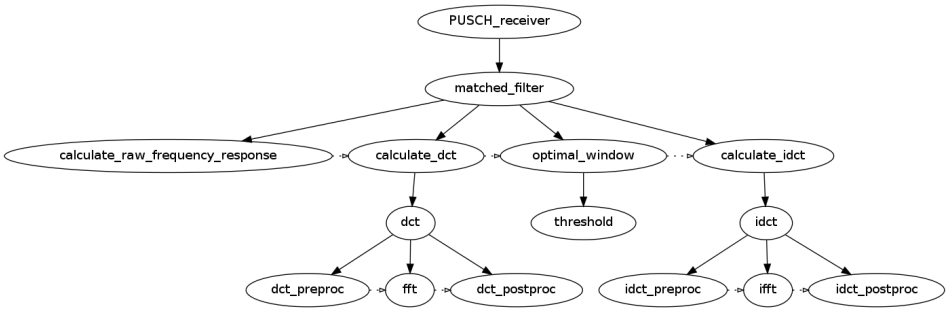
As before `m` is the AST. `:toplevel` and `:out` specifies where to perform the transformation. `"foosplit"` is the name of the new function, `5` specifies the number of tasks after the transformation and `1` is the dimension that should be split.

6.2 Using the transformations

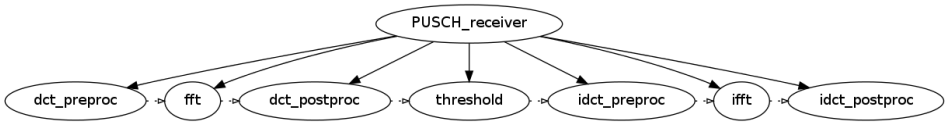
The purpose of using Julia, the DSL presented in chapter 4.2 and the transformations presented in the previous section is that it allows us to implement a baseband algorithm in a style that is easy to understand and read. This means that it should be free of explicit parallelization constructs. It should also be structured in a way that is logical to a algorithm developer. The *Uplink Receiver Algorithm* described in Chapter 2.2 is divided into the functions `matched_filter()`, `calculate_noise_covariances()`, `antenna_combining_and_FDE()`, `IDFT_despread()` and `soft_bits_demapping()`. The transformations should then be used to transform this into a form that explicitly specifies how the implementation is parallelized and how it is divided into tasks. The reason to do this is to make it easy to work with the implementation and still be able to quickly generate code that is parallelizable. This also makes it possible to see and reason about how the changes made by the transfomations affect parallelization.

Applying transformations to the baseband processing algorithm is semi-automatic in the sense that the order to do them, where to do them and which transformation to do is specified manually, but the transformation is automatic. Choosing the right transformation is not always straight forward, but there exists a good strategy. First *inlining* and *push vectorization* is used to create a flat structure of the program. This will create a top-level function that does all the work. This function is just a list of operations. When this flat structure has been created, operations that are parallelized in the same way should be identified and grouped into tasks by using *extract function*. Operations that can be performed on hardware accelerators should be left in the top-level function. When this is done the *pull vectorization* is used on the created functions to pull the identified parallelizations to the top-level function. The problem is that it is not always obvious how to group operations together.

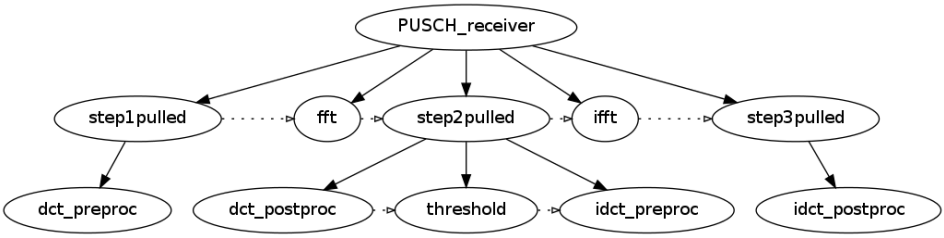
In Figure 6.1 the process can be seen. The figure shows a graph of calls that are being made from the top-level function of the implemented uplink receiver algorithm. Its a modified version that only calculates frequency response. Each ellipse is a function and a solid arrow means that the function at the start of the arrow calls the function at the end of the arrow. The small dotted arrows show the order that the function are called in. The top image shows the algorithm before any transformation has been made. The `PUSH_receiver()` top-level function calls `matched_filter()` which then performs a number of other operations. Notably the `fft()` and `ifft()` operations are at the fourth level in the graph and are thus nested deep in the function hierarchy. The middle graph shows the algorithm after a number of *inline* and *push vectorization* transformations has been performed. We note that almost all calls have vanished. Almost all functions that were previously called have been inlined an the functionality that they provided is done directly in `PUSCH_receiver()` instead. It is important to notice is that the `fft()` and `ifft()` operations have moved and are now called directly by the top-level function. This means that they can now be executed on accelerators. Now all the *inlined* code that currently is in `PUSCH_receiver()` is grouped



(a) Before transformations



(b) After inlining *and* push vectorization *transformations*



(c) After all transformations

Figure 6.1: *Callgraphs during transformation of the uplink receiver algorithm. (Some calls has been left out to make the graphs easier to read.)*

into three functions `step1()`, `step2()` and `step3()`. `step1()` will contain everything before the call to `fft()`, `step2()` will contain everything after the call to `fft()` and before the call to `ifft()` and `step3()` will contain everything after `ifft()`. Lastly a *pull vectorization* is performed on all the *extracted* functions to make them parallelized. This is shown in the bottom graph of Figure 6.1, the three parallelized tasks and the accelerator tasks are shown. In Listing 9 the top-level function and the `matched_filter()` function of the code before the transformation and the top-level function after the transformations can be seen. Here we see that the FFT and IFFT operations has been moved to the top-level function. We can also see that the DSP tasks (`step1pulled()`, `step2pulled()`, `step3pulled()`) are parallelized and can operate on multiple antennas and on both slots at in parallel.

The structure of a Julia program that is described in Section 4.1 provides structure that can be used to express hardware dependent aspects of a baseband process-

ing algorithm in a high-level language. It will create a Julia program that is self-contained and can execute on a normal PC while it still expresses the parallelization and task division that will be used in target hardware. This approach offers great freedom in the development process of a baseband processing algorithm. A first step in implementation of a baseband algorithm can be a prototype implementation in Julia. This implementation can either conform to the DSL described in section 4.2, only have parts that do, or not conform at all. When the prototype is implemented a refactoring step can be done that refactors the implementation to use the structure with a top-level function and conform to the DSL. The refactoring should also consider how the described transformations can be applied and make changes to the implementation to accomodate this. When this is done transformations can be applied to create a Julia program that explicitly describes the tasks and parallelizations that should be used in a target hardware implementation. This is an incremental approach where all steps can be more or less integrated with each other. This should be put in contrast to the often used workflow where the high-level and hardware implementation are separate and sequential.

Before transformations:

```
function PUSCH_receiver(rxPRBs,ueParams)
    frequency_response = matched_filter(rxPRBs,ueParams)
    frequency_response
end
function matched_filter(rxPRBs,ueParams)
    rxDMRS = DMRS_extraction(rxPRBs,ueParams)
    trueDMRS = calculate_DMRS(ueParams)
    raw_frequency_response =
      calculate_raw_frequency_response(rxDMRS,trueDMRS,ueParams)
    h_hat = calculate_dct(raw_frequency_response)
    h_hat_opt = optimal_window(h_hat,ueParams)
    H_hat_opt = calculate_idct(h_hat_opt)
    H_hat_opt
end
```

After transformations:

```
function PUSCH_receiver(rxPRBs,ueParams)
    matchfilt_trueDMRS = calculate_DMRS(ueParams)
    dct_v = cat(3,[cat(2,
      [step1pulled(rxPRBs[:,slot,antenna],
      ueParams,matchfilt_trueDMRS,antenna,slot)
      for slot = 1:2]...) for antenna = 1:GLOBAL_PAR.nRx]...)
    dct_V_unscaled = cat(3,[cat(2,
      [fft(dct_v[:,slot,antenna]) for slot = 1:2]...)
      for antenna = 1:GLOBAL_PAR.nRx]...)
    idct_Y = cat(3,[cat(2,
      [step2pulled(dct_V_unscaled[:,slot,antenna],
      ueParams,antenna,slot)
      for slot = 1:2]...) for antenna = 1:GLOBAL_PAR.nRx]...)
    idct_y_unscaled = cat(3,[cat(2,
      [ifft(idct_Y[:,slot,antenna]) for slot = 1:2]...)
      for antenna = 1:GLOBAL_PAR.nRx]...)
    frequency_response = cat(3,[cat(2,
      [step3pulled(idct_y_unscaled[:,slot,antenna],antenna,slot)
      for slot = 1:2]...) for antenna = 1:GLOBAL_PAR.nRx]...)
    frequency_response
end
```

Listing 9: *Source code showing application of transformations*

7

Conclusions

This chapter discusses the results and conclusions that can be drawn from the thesis. Both problems that have been solved and those that still remain will be discussed.

7.1 Implementing baseband algorithms in Julia

The work with implementing the uplink receiver algorithm gives indication that Julia is a language that is suitable for prototyping baseband processing algorithms. It has been an easy job to translate an existing Matlab implementation to Julia. This indicates that Julia is as expressive as Matlab when implementing baseband processing algorithms. There has not been any instance where the use of Julia has made it necessary to use significantly different constructs than those used in Matlab. The problem that exists with using Julia as a prototyping language for baseband processing is not the expressiveness of the language itself but rather a matter of the maturity of the language. Julia is still under development and there are problems with the stability of the language. Bugfixes and changes in the language are frequent and this can break implementations and make it necessary to put a lot of work in maintenance of existing code. If this becomes a big problem it could be solved by using a fixed release of Julia but this will also create problems. There is less development on old releases and there is very little work maintaining them. This will probably not change until a version 1.0 is released. Migrations to new releases will be probably require lots of effort since the language is changing so fast. This means that if Julia is to be used the following have to be taken into consideration, is it important to have a stable language or is it important with new features and improvements in the language. It is also important to consider how to handle release changes.

7.2 Using transformations to adapt Julia source code to target platform

The thesis investigates if using the metaprogramming features of Julia is a good approach for simplifying development baseband processing algorithms. During the work a model of a baseband processing platform has been established. The model can be used to express parallelization and the use of hardware accelerators in Julia. This gives an indication that with more work it might be possible to do all platform adaptation in Julia and then use a combination of code generation and more transformations to generate programs that can be executed on target. If this is achieved it is probable that it will be possible to test both different strategies for parallelization and different algorithms much more quickly than earlier. It will also be possible to adapt the implemented algorithms to many different hardware realizations. The model described in the thesis does not specify the number of processing cores or memory sizes. The adaptation to a particular hardware should be done with transformations and not during implementation of the algorithm. This will make it possible to easily adapt a single implementation of a baseband processing algorithm to many different hardware realizations, as long as they follow the platform model from section 2.3.

Using the metaprogramming features in Julia has proven to be a useful approach for implementing code transformations. Implementing transformations in the same language has a lot of benefits. There is only one language used which removes the need for other tools that perform transformations. This means that there are less languages or scripting languages that needs to be learned. This might help to close the gap between application developers and tool developers. Other tools that are used for code transformation often need complex implementations of parsers. Julia makes it possible to use Julia's own parser to construct an AST, which then can be used for code transformations. This approach has proven possible. An issue is to decide whether it is useful to do these transformations. The assumption that has been made in the thesis is that the transformed code should be able to efficiently use the assumed hardware model, including using FFT hardware accelerators and parallelizing the work over multiple DSP cores. The work described in the thesis shows that Julia can be used to do source code transformation on Julia programs in a way that achieves this.

7.3 Future Work

The transformations described in the thesis are useful for achieving desired structure of the program if used on the uplink receiver algorithm described in the thesis. To develop this into a general tool, that can be used for a wide variety of baseband processing algorithms, there is a need for more transformations. It has not been concluded if there is a set of transformations that are sufficient for all algorithms. Most likely this is not the case. There will be cases that can not be handled by a reasonable set of transformations. There will be a need to do

highly specialized transformation for certain problems. Another way to handle these cases is to simply rewrite them by hand when they arise. Common transformations can be implemented in Julia, while uncommon ones can be handled manually.

The transformations described in this thesis handle vectorization of operations where the input and output array are of the same size. This could be generalized to support output arrays that are both larger and smaller than the input array. An example would be to calculate the energy of a signal. Input would be an array of samples and the output a single value. It would also be useful to be able to produce more than one output array from a single input array.

Another improvement would be to check that a program conforms to the described DSL. This would make it less likely that errors occur when the transformations are applied. If the check could report where the algorithm is breaking the rules of the DSL it would reduce time spent on finding errors.

To make Julia useful for baseband processing the transformed code would need to be converted to run on the target hardware. This will involve converting the top-level function to a schedule that describes when tasks should be executed and how input and output data from the tasks are connected. It also needs to convert the Julia functions in the program to tasks that are executable on the target hardware. This will require some sort of memory management and to remove the reliance of the JIT-compiler in Julia.

Bibliography

[1] *julialang.org*, 2014. URL http://julialang.org. Cited on pages 3 and 4.

[2] *The LLVM Compiler Infrastructure*, July 2014. URL http://llvm.org. Cited on page 10.

[3] *Julia Language Documentation*, July 2014. URL http://julia.readthedocs.org/en/latest/manual/noteworthy-differences/. Cited on page 9.

[4] 3GPP 36.211. *Evolved Universal Terrestrial Radio Access (E-UTRA); Physical channels and modulation*. 2006. URL http://www.3gpp.org/DynaReport/36211.htm. Cited on pages 12, 14, and 29.

[5] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, and et al. *Julia Language Documentation*, July 2014. URL http://media.readthedocs.org/pdf/julia/latest/julia.pdf. Cited on page 4.

[6] Niclas Wiberg. Personal Communication, 2014. Cited on page 17.

Linköpings universitet

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida `http://www.ep.liu.se/`

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: `http://www.ep.liu.se/`

© Linus Mellberg