CHALMERS


# Raising Feldspar to a Higher Level

## Extending the Support for Digital Signal Processing Functions in Feldspar

*Master of Science Thesis in the Programme Foundations of Computing – Algorithms, Languages, and Logic*

KARIN KEIJZER

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2010

Raising Feldspar to a Higher Level
Extending the Support for Digital Signal Processing Functions in Feldspar

Karin Keijzer



Supervisor: Emil Axelsson
Examiner: Mary Sheeran

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2010

# Contents

# Chapter 1

# Introduction

Baseband telecommunication systems need more processing power every day, and single core systems have a hard time to deal with the processing requests from the users. Therefore, more and more systems are replaced by multi-core or multi-processor systems. These systems need adapted applications to optimize usage of the cores and processors. On top of the need for more processing power, most code for digital signal processing applications is currently hand written and optimized in low level C code. Writing this code for various hardware platforms is a time-consuming process.

Feldspar [1] is a domain specific language (DSL) for digital signal processing (DSP) initially designed with baseband telecommunication systems in mind. Feldspar is a strongly typed high level functional language which during compilation is translated to hardware specific C code via an intermediate core language. Because it is a high level functional language it raises the level of abstraction of the code for programmers. In short, digital signal processing systems perform operations, i.e. functions, on continuous data streams, which can often be represented as functions on finite-sized vectors or arrays[2]. Feldspar programs are able to use these vectors and in many cases the compiler can optimize the code written for these vectors. In this thesis I will investigate raising the level of abstraction for implementing digital signal processing functions. More specifically, I will design and implement general helper functions for filters and filter-like functions; and I will design and implement a matrix module for transforms like the Fourier transform.

The language Feldspar is a fairly young language; and I was one of the first to implement digital signal processing functions in it. As a result of Feldspar being a new language, it has changed several times during my thesis. For example one of the larger change included the vector module. Before there were two different kind of vectors, parallel and sequential, where there is currently only one kind of

vector. Also the vectors had type level sizes which means that the length of the vector was defined in the type. The changes to the current version of vectors made programming in Feldspar easier; however, it also meant that I had to rewrite my examples and other Feldspar code from time to time.

During this thesis I also contributed to a paper sent to MEMOCODE 2010, *Feldspar: A Domain Specific Language for Digital Signal Processing algorithms* [1]. It was very nice to get such an opportunity. For the paper I worked on examples and on the case study about matrices in Feldspar. This case study is also included in Chapter 6.2 of this thesis.

In this thesis some prior knowledge of functional programming, more specifically Haskell, is expected. A good introduction to Haskell is given by Hutton [3].

**Outline** Chapter 2 provides background information about domain specific languages, digital signal processing, and digital signal processing functions. Next, Feldspar is introduced briefly in Chapter 3. Chapter 4 gives a short summary about the case studies I have worked on during this thesis, after which Chapters 5, 6, and 7 describe them in more detail. After that, in Chapter 8, I will discuss some related work which has been done in the area of domain specific languages and digital signal processing. I will end this thesis with a conclusion in Chapter 9.

# Chapter 2

# Background

This chapter contains general background information on domain specific languages, digital signal processing, baseband processing, and the digital signal processing functions used in this thesis.

## 2.1 Domain Specific Languages

Domain specific languages (DSLs) are programming languages designed and optimized for a specific domain. As a result of being designed for a specific domain DSLs offer often more expressiveness and are easier to use in their domain than a general purpose programming language such as C, Java, or Haskell [4].

Examples of well known DSLs are SQL, a language to add, change, and retrieve information from databases; VHDL, a hardware description language; and LaTeX, a document markup language.

In this thesis I will use the domain specific language Feldspar which is designed for digital signal processing with baseband telecommunication systems in mind.

## 2.2 Digital Signal Processing

Digital signal processing (DSP) encompasses the analysis, representation, and modification of digital signals. The signals in DSP are represented as discrete-time signals which contain equally spaced discrete samples synchronous to the original analog signal [2]. We will not consider continuous signals in this thesis.

### 2.2.1 Digital Signal Processing Domains

Signals in digital signal processing can be evaluated in several domains[1]. Examples of these domains are the time domain, frequency domain, autocorrelation domain, and wavelet domain. Each domain has different characteristics, and depending on the domain of the signal it is possible to capture different aspects of the signal. In the next paragraphs a short overview of the time and frequency domains is given.

**Time domain** In the time domain, a signal is represented as a function over time. The three most basic operations which can be applied to signals in the time-domain are scaling, delay, and addition. Scaling multiplies the input signal with a constant value, a delay operation delays the input signal, and addition adds a constant value or another signal to the input signal [2]. Below is the Feldspar code of a scale, delay and add function shown. Note that this code could as well be Haskell code.

```
  -- Multiply the signal xs by a constant
scale c xs = map (* c) xs

  -- Delay the signal xs by c
delay c xs = replicate c 0 ++ xs

  -- Add a constant c to the signal xs
add c xs = map (+ c) xs
```

These basic operations are used to build more complex signal processing operations, like filters. Filters are used to capture specific characteristics of the signal. For example how a signal changes over time. Examples of different filters are finite impulse response and infinite impulse response filters. A more detailed discussion on filters is provided in Section 2.2.2.

**Frequency domain** The frequency domain shows the magnitude and the phase component of each frequency used in a signal. Furthermore, in this domain, filters can be used to capture specific information such as the frequency and phase of a signal. The Fourier transform transforms a signal from the time domain to a signal in the frequency domain [5]. The signal can be transformed back to the time domain using the inverse Fourier transform. A mathematical description of the discrete Fourier transform is given in section 2.4

[1] The word domain is here used in a different context from that in Section 2.1.

### 2.2.2 Filters

In digital signal processing a filter removes unwanted information from a signal. In this thesis only discrete time filters are used. Below two important types of filters are explained, namely the finite impulse response filter and the infinite impulse response filter.

The **finite impulse response filter** (FIR) is a type of digital filter which after an impulse converges to zero in a finite number of steps [5]. This is in contrast to infinite impulse response filter which does not converge to zero. Examples of FIR filters are Moving Average and lowpass filters [6].

The **infinite impulse response filter** (IIR) is a type of digital filter which, contrary to a FIR filter, does not converge to zero when applied to an impulse [5]. This is because an IIR filter uses the previous input and output values as feed-forward and feedback. In this thesis I will use the term feed-forward for reusing previous input values, thus when iterating over a vector and the current index is $n$ (and the next input is $n + 1$) with the term previous input values I mean the indices 0 to $n - 1$ of the input vector. Examples of IIR filters are Chebyshev filter, Butterworth filter, and the Bessel filter [5] [6].

## 2.3 Baseband Processing

Signal processing systems tend to have strict timing requirements, for example in real-time signal processing applications like Internet voice and video applications delays are undesirable. Telecom base-stations often have hardware and microprocessors specifically designed for DSP operations. Digital signal processors perform many mathematical operations in parallel and therefore have an instruction set optimized for these calculations [7]. In this section, I will give a brief introduction to baseband processing.

The radio frequency spectrum is a scarce resource; therefore researchers and engineers are trying to improve the spectral efficiency of modulation methods. This is done to increase the bandwidth such that more users simultaneously are able to use the same frequencies and more data could be sent on the same frequencies [8].

Modulating a data signal with a carrier signal is varying the frequency, amplitude, phase, and other components of the carrier signal such that the data is sent over the carrier signal.

Baseband processing encompasses the modulation of digital bit-streams to frequency signals before sending the data, and the recovery of the modulated signal after receipt. During the transmission of the signal they are subject to interference,

noise, impairments, etc. Consequently, error correction algorithms are needed to recover the original signal [8].

Improving the spectral efficiency also introduces a significant amount of complexity to baseband processing on top of the fundamental challenges and computations of radio wave propagation. Two of the problems which occur during the transmission are multi-path propagation and transmission errors. The first one occurs when a signal reaches the destination using several paths, for example when a signal reflects on objects between the sender and the receiver. There are many reasons transmission errors occur, one of them is noise. Noise is an unwanted signal combined with the data signal sent by the sender. Noise can, for example, come from lightning or from interference with other signals. After receiving a signal, several processing tasks need to be performed: channel coding including turbo and convolutional codes, but also synchronization, channel estimation and equalization. For more information on baseband processing, refer to [8].

## 2.4 Digital Signal Processing Functions

In this thesis, several common digital signal processing functions are used as examples and implemented in Feldspar. In the next paragraphs, I will give short descriptions of them.

### 2.4.1 Filters

**Moving average** (MA) is a type of finite impulse response filter which is used to analyze data points by creating a series of averages of the input. [5]

$$y[n] = \sum_{i=0}^{P} b_i \; x[n-i] \tag{2.1}$$

Where $x[n]$ is the input vector, $b_i$ are the filter coefficients, $P$ is the filter order, and $y[n]$ is the output vector. Figure 2.1 shows the Moving Average model [6]. Below is a short explanation of the symbols used in Figures 2.1, 2.2, and 2.3.

In the Figures 2.1 and 2.2, $x_k$ and $y_k$ represent the input and output vector of the filter. The values $a_n$ and $b_n$ are the filter coefficients, as shown in Equations 2.1 and 2.2. The symbols $\bigotimes$ and $\bigoplus$ represent multiplication and addition of the input values. As last, the symbol $\boxed{\mathbf{D}}$ represents a delay in the signal. In addition to the symbols explained above, the turbo encoder (Figure 2.3) also has a switch (shown as two crossing arrows, a solid and a dashed arrow). This switch connects

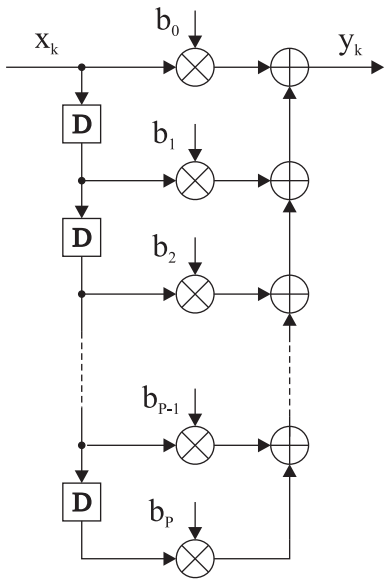


Figure 2.1: Moving average model.

the solid lines whenever there are new input values; however, when there are no more input values and there are still values in the delay components, the switch connects the dashed line to the system.

**Autoregressive model** (AR) is a type of infinite impulse response filter that attempts to predict the output values $y[n]$ based on the previous outputs. The output value depends on the current input value ($x[n]$) and previous output values ($y[n-1], y[n-2], \ldots, y[0]$). [5]

$$y[n] = x[n] - \sum_{i=1}^{Q} a_i \; y[n-i] \tag{2.2}$$

Where $x[n]$ is the input vector, $a_i$ are the filter coefficients, $Q$ is the filter order, and $y[n]$ is the output vector. Figure 2.2 shows the Autoregressive model [6].

**Autoregressive moving average model** (ARMA) is a combination of AR and MA, and therefore a type of infinite impulse response filter. [5]

$$y[n] = \sum_{i=0}^{P} b_i \; x[n-i] \; - \; \sum_{j=1}^{Q} a_j \; y[n-j] \tag{2.3}$$

Where $x[n]$ is the input vector, $b_i$ are the feed-forward filter coefficients, $P$ is the feed-forward filter order, $a_i$ are the feedback filter coefficients, $Q$ is the feedback filter order, and $y[n]$ is the output vector.

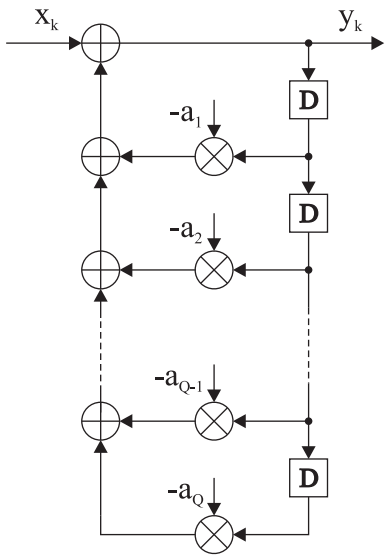


Figure 2.2: Autoregressive model.

**Turbo code** is not a filter but a high-performance error correction code used in 3G mobile telephone standards. [9]

Turbo coding consists of two parts, encoding of the signal before sending, and decoding of the received signal. In this report I will only use and implement the Turbo encoder.

The Turbo encoder, shown in Figure 2.3, receives one input vector and returns three output vectors. The first output vector $x_k$ is equal to the input vector. The second output vector $z_k$ is a combination of the input and intermediate values. The last output vector $z'_k$ is basically the same as the second output vector, except before computing it the input vector is interleaved [9]. Interleaving a vector is re-arranging the elements in a non-contiguous way such that two succeeding elements are not next to each other after interleaving the vector.

### 2.4.2 Transforms

The transforms considered in this thesis are as follows:

The **discrete Fourier transform** (DFT) transforms a signal from the time domain into the frequency domain [5]:

$$y[n] = \sum_{k=0}^{N-1} x[k]\ e^{-\frac{2\pi i}{N}kn} \qquad n = 0,\ \ldots\ , N-1 \tag{2.4}$$

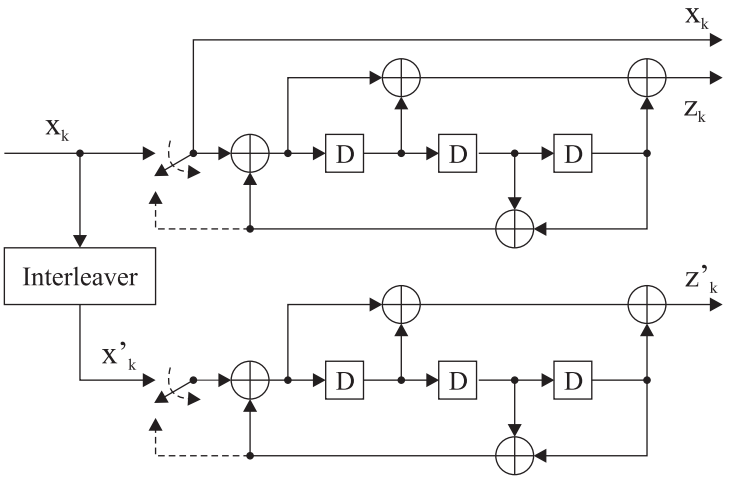


Figure 2.3: Turbo encoder.

The input $x[n]$ and output $y[n]$ are finite sequences where the input consists of real valued numbers and the output of complex valued numbers, and $N$ is the number of input values. The naïve implementation has a runtime complexity of $O(n^2)$.

The **fast Fourier transform** (FFT) is an efficient algorithm to compute the DFT which takes $O(n \log n)$ instead of $O(n^2)$ arithmetical operations. A well known algorithm is the Cooley-Tukey algorithm [10].

The **discrete cosine transform** (DCT) is a function similar to the DFT, but using only real numbers. The DCT is used in many signal and image processing applications. There are several variants of the DCT, the most common it the type-II DCT [11]:

$$y[n] = \sum_{k=0}^{N-1} x[k] \ \cos\left[\frac{\pi}{N}\left(k + \frac{1}{2}\right) n\right] \qquad n = 0, \ \ldots \ , N-1 \tag{2.5}$$

Where $y[n]$ are the output values and $x[n]$ are the input values.

The **Walsh-Hadamard transform** (WHT) is a generalized Fourier transform, using only real numbers [12]:

$$y[n] = \sum_{k=0}^{N-1} x[k] \ (-1)^{n \cdot k} \qquad n = 0, \ \ldots \ , N-1 \tag{2.6}$$

Where $n \cdot k$ is the bitwise dot product of $n$ and $k$, $x[n]$ are the input values, $y[n]$ are the output values.

# Chapter 3

# Feldspar

Feldspar [1, 13] is a domain specific language (DSL) for digital signal processing (DSP), designed with baseband telecommunication systems in mind. Feldspar is developed in a joint research project between Ericsson, Chalmers University of Technology (Göteborg, Sweden), and Eötvös Loránd University (Budapest, Hungary).

Feldspar is a strongly typed high level functional language which raises the level of abstraction for programmers. The aim is to provide the same style of programming as used in Haskell with the opportunity to allow compilation to efficient hardware specific C code. To make the compilation to efficient code possible, Feldspar is more restrictive than Haskell.

This chapter provides a short introduction to Feldspar. First the language Feldspar is described: how is it used, what does it look like, and what are the results. After that, an extension to Feldspar, the vector module, is discussed. This chapter ends with showing matrices in Feldspar.

## 3.1 Using Feldspar

Feldspar is a functional language which allows the user to use a data-flow style of programming, where the output of a computation is described as several transformations of the input. Programs can be built up from simple reusable components which leads to short and readable programs.

Feldspar can be found at [14].

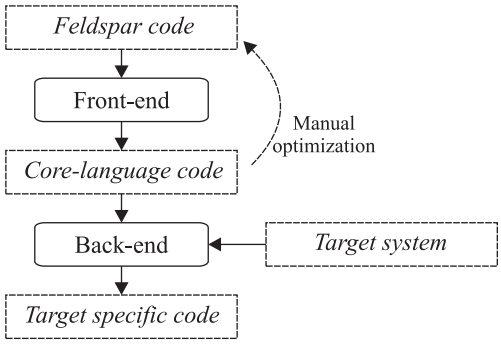


Figure 3.1: Compilation flow of Feldspar programs.

### 3.1.1 Compilation

The Feldspar compiler consists of two parts: the front end which transforms a Feldspar program into the *core language* code (Section 3.1.3), and the back end which compiles the core language code into code optimized for the desired target platform (Section 3.1.4). The front end is a collection of Feldspar modules implementing functions like `for` and `map`. For convenience I will call the front end also a compiler in this thesis since it transforms Feldspar code into core language code. A high-level overview of the Feldspar compilation flow is given in Figure 3.1.

### 3.1.2 Feldspar

Below is a simple example of a Feldspar program which computes the sum of squares of the numbers `1` to `n`, and next to it, on the right side, is the same program written in Haskell.

Feldspar:

```
sumSq :: Data Int -> Data Int
sumSq n = sum (map square (1...n))
  where
    square x = x*x
```

Haskell:

```
sumSq :: Int -> Int
sumSq n = sum (map square [1..n])
  where
    square x = x*x
```

The first line of code defines the type of the function, namely given a parameter of type integer it produces an integer as output. The type of the integers in Feldspar is not `Int` as in Haskell but `Data Int`. The type `Data a` is the Feldspar version of the type `a` in Haskell. Feldspar is embedded in Haskell, which means that you can still use Haskell types, functions and variables in the Feldspar code. However the Haskell code needs to be removed during the compilation to core language code.

In Chapter 6 a short example will be shown how to remove Haskell variables.

The function `(...)`, used in the code above, produces a list of numbers from `1` to `n`. After that the function `square` is applied to each element of the generated list using the function `map`. The function `square` returns the square of the input value. Finally the program adds up all the elements in the list of squares using the function `sum`. The Feldspar code can be evaluated using the GHCi prompt like Haskell:

```
*Main> eval (sumSq 10)
385
*Main> eval (sumSq 20)
2870
```

The next section will discuss the core language of Feldspar.

### 3.1.3 Core language

The core language code can be produced, from a program written in Feldspar, using the function `printCore`. The core language code can for example be used to analyze the efficiency of the generated code. In this section, some important core language constructs are shown and core language code, generated from Feldspar code, is explained.

An important constuct in the core language code is the `while`-loop. The `while`-loop in the core language takes as arguments two functions and an initial state. The first function, `cont`, is the continuation condition that takes the current state and returns a boolean indicating whether to continue or not. The second function, `body`, is the body of the loop. It takes the current state and returns the next state.

```
while :: Storable a =>
    (Data a -> Data Bool) -> (Data a -> Data a) -> Data a -> Data a
```

When the Feldspar program `sumSq`, from previous section, is compiled to core language code, the core language code contains a `while`-loop.

```
*Main> printCore sumSq
program v0 = v11_1
  where
    v2 = v0 - 1
    v3 = v2 + 1
    v4 = v3 - 1
    (v11_0,v11_1) = while cont body (0,0)
      where
```

```
cont (v1_0,v1_1) = v5
  where
    v5 = v1_0 <= v4
body (v6_0,v6_1) = (v7,v10)
  where
    v7 = v6_0 + 1
    v8 = v6_0 + 1
    v9 = v8 * v8
    v10 = v6_1 + v9
```

The core language code generated by the compiler is actually correct Haskell code and can be interpreted, given suitable helper functions. The syntax of the core language is easy to understand and reasoning about the efficiency is straightforward. The core language code can be translated relatively easily to C.

The generated code, from the `sumSq` program, contains several variable assignments and one C-style `while` loop. The `while` loop loops over the indices of the input vector. Inside the loop, variable `v5` contains the continuation condition, variable `v7` contains the index of the while loop, variable `v9` contains the value of the list at the current index squared, and variable `v10` contains the total sum of the squared values.

Next to the `while`-loop, there are two other important core language constructs: `ifThenElse` and `parallel`.

The conditional expression `ifThenElse` takes a boolean condition, a `then` branch, an `else` branch, and a value which is the input to the then or else branch (depending on the value of the condition).

```
ifThenElse :: (Storable a, Storable b) =>
    Data Bool -> (Data a -> Data b) -> (Data a -> Data b) -> Data a ->
                                                                Data b
```

The `parallel` construct produces an array. As first argument it takes the size of the final array and as second argument a function which, given the index of the array, computes the value of the element. Using the `parallel` loop construct, it is possible to compute each element of the array independently of the other elements, for example in parallel. Currently the parallel construct is still computed sequentially; however, this construct opens the possibility to be computed in parallel in the future.

```
parallel :: Storable a =>
    Data Int -> (Data Int -> Data a) -> Data [a]
```

### 3.1.4 C Code

The core language code from previous section can be compiled to platform dependent C code. Currently the only backend implemented for Feldspar transforms core language code to C99 code. Below is an example of the Feldspar program `sumSq` compiled to C99 using the `icompile'` function.

```
*Main> icompile' sumSq "sumSq" defaultOptions
#include "feldspar.h"

void sumSq(signed int var0, signed int * out)
{
    signed int var11_0;

    var11_0 = 0;
    (* out) = 0;
    {
        while((var11_0 <= (((var0 - 1) + 1) - 1)))
        {
            signed int var6_0;
            signed int var8;

            var6_0 = var11_0;
            var11_0 = (var6_0 + 1);
            var8 = (var6_0 + 1);
            (* out) = ((* out) + (var8 * var8));
        }
    }
}
```

In this thesis I will only use the generated core language code and not the generated C code. For more information about the generated C code and the Feldspar compiler refer [15].

## 3.2 Vectors

Digital signal processing functions mostly perform operations on streams of values. To handle these streams of values, Feldspar has a vector module. The function `sumSq`, from the previous sections, already used a vector, namely the function `(...)` generates a vector.

The vector module provides *symbolic vectors* which are to some extent comparable

to lists in Haskell. The vector module provides a higher level of programming offering functions like `map` and `fold`. This makes it in many cases unnecessary to use the low level core language loops like `while` and `parallel`.

A symbolic vector is constructed using the function `indexed`. `indexed` takes an integer length and an indexing function, and it produces a vector. The indexing function returns the value from the symbolic vector given the index.

```
indexed :: Data Length -> (Data Ix -> a) -> Vector a
```

This function is very similar to the `parallel` in the core language. However `indexed` never actually creates the vector in memory it just passes the arguments to the vector constructor `Indexed`.

The function `indexed` stores the length and the indexing function in the constructor `Indexed`. This way it is possible to to pattern match on symbolic vectors and modify the components, length and indexing function. For example, the function `squareList` (shown below) squares each element of the list `xs` using pattern matching. It pattern matches on the `Indexed` constructor and replaces the current indexing function `ixf` with a new indexing function `(square . ixf)`.

```
xs :: Vector (Data Int)
xs = indexed 5 (*2)

square :: Data Int -> Data Int
square x = x*x

squareList :: Vector (Data Int) -> Vector (Data Int)
squareList (Indexed sz ixf) = Indexed sz (square . ixf)
```

The function `squareList` only modifies the function component of the symbolic vector: there is no *real* vector involved.

In this thesis I will often use the type `DVector a`, this is equal to the type `Vector (Data a)`.

The Feldspar interpreter and compiler can also be used on programs containing vectors.

```
*Main> eval (squareList xs)
(5,[0,4,16,36,64])

*Main> printCore (squareList xs)
program = (5,v4)
  where
    v4 = parallel 5 ixf
      where
```

```
        ixf v1 = v3
          where
            v2 = v1 * 2
            v3 = v2 * v2
```

The result of the `eval` function is a pair containing the length of the resulting list and the core representation of the list. The core language code contains only a single `parallel` loop which computes the values of the output vector using its indices. For each value in the vector the index is multiplied by two and then squared.

Instead of using the knowledge about the underlying system of the vector module, for instance to perform pattern matching on vectors, a higher level of programming can be used to achieve the same results. For instance the higher order function `map` applies a function to each element of a vector.

```
squareList2 :: Vector (Data Int)
squareList2 = map square (indexed 5 (*2))
```

The generated core language of `squareList2` is exactly the same as the core language code of `squareList xs`.

```
*Main> eval squareList2
(5,[0,4,16,36,64])

*Main> printCore squareList2
program = (5,v4)
  where
    v4 = parallel 5 ixf
      where
        ixf v1 = v3
          where
            v2 = v1 * 2
            v3 = v2 * v2
```

### 3.2.1 Fusion

As seen in the generated code above, there is only one loop, while the Feldspar code would suggest there should be two loops: one loop that creates the initial vector and another that squares each element of the initial vector.

The Feldspar frontend can in many cases *fuse* structures, like these loops which are created by functions, into a single structure. This way it is possible to write efficient

programs without thinking about the underlying structure. In the generated code of `squareList2` the initial list `indexed 5 (*2)` is never created.

However when fusion is undesired, the function `memorize` can be used to avoid fusion. For example, below the core language code of `squareList3` is shown where the list `indexed 5 (*2)`, from the previous example, is replaced by `memorize $ indexed 5 (*2)`.

```
squareList3 :: Vector (Data Int)
squareList3 = map square (memorize $ indexed 5 (*2))

*Main> eval squareList3
(5,[0,4,16,36,64])

*Main> printCore squareList3
program = (5,v7)
  where
    v4 = parallel 5 ixf
      where
        ixf v2 = v3
          where
            v3 = v2 * 2
    v7 = parallel 5 ixf
      where
        ixf v1 = v6
          where
            v5 = v4 ! v1
            v6 = v5 * v5
```

The function `memorize` forces the computation of the initial vector `indexed 5 (*2)` and stores the result in `v4`. After that the vector stored in `v4` is used to compute the square or each element.

## 3.3 Matrices

Using the vector module from previous section it is possible to implement matrices in Feldspar. A matrix can be represented as a nested vector. To construct a matrix the `indexed` function from the vector module can be reused to make a similar constructor for matrices, `indexedMat`.

```
  -- Matrix
type Matrix a = Vector (DVector a)
```

```
  -- Create a Matrix given the width, height, and indexing function.
indexedMat :: Data Int -> Data Int ->
                     (Data Int -> Data Int -> Data a) -> Matrix a
indexedMat m n idx = indexed m (\k -> indexed n (\l -> idx k l))
```

In this thesis, only a few basic matrix operations are used. The most important is matrix-vector multiplication. Matrix-vector multiplication can be implemented as a scalar product of each row of the matrix with the input vector.

```
  -- Matrix vector multiplication
(**) :: (Numeric a) => Matrix a -> DVector a -> DVector a
(**) mat vec = map (scalarProd vec) mat
```

Next chapter introduces the case studies I have been working on during this thesis.

# Chapter 4

# Case Studies

Feldspar is a fairly new language and there are not many modules and functions available specifically designed for the digital signal processing domain. To figure out what functions are needed for digital signal processing and what modules to add to Feldspar I started to implement two different types of digital signal processing functions, namely filters and transforms.

## 4.1 Filters

First I focused on digital signal processing filters. They are used to filter data for specific characteristics. Filters often reuse previous input and output values to compute the next value.

Figure 4.1 explains the terms *previous input values* and *previous output values* I use in this thesis.

To compute a filter like function a general function, like `map`, which loops over

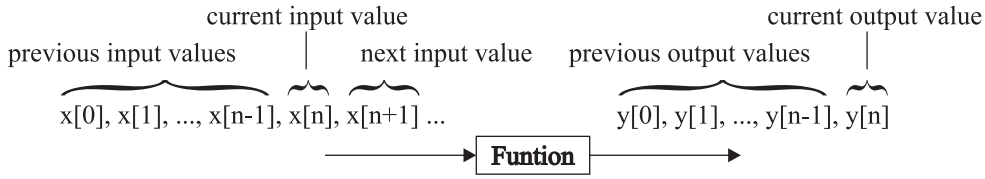


Figure 4.1: The *previous input values* are the values in the input vector $x$ at indices 0 to $n-1$. The *current input value* is at index $n$. *Previous computed output values* are the values in the output vector $y$ at indices 0 to $n-1$. Note that the value of the current output value $y[n]$ is unknown until it is actually computed.

the input vector is required. However, there are currently no general functions in Feldspar which are capable of providing both previous input and previous output values which are needed in many filters. The function `map` does not provide access to previous computed output values. Therefore I tried to design a *combinator*. A combinator is a Feldspar function which often replaces syntactic structures from other languages. This combinator should be broad enough such that many digital signal processing functions are able to use it and at the same time the generated core language code should be efficient. A function using the combinator should generate code which is comparable to the digital signal processing function specific code in complexity and evaluation time. Chapter 5 describes the work I have done in this area.

## 4.2 Transforms

Another interesting kind of digital signal processing functions are transforms. A transform maps a function or a signal from one domain into another domain, for instance the discrete Fourier transform transforms a signal from the time domain into the frequency domain. First, I will mention known algorithms for computing transforms, after that, a more mathematical approach is investigated.

### 4.2.1 Known Algorithms

For the transforms I used there are two different kind of algorithms, a naïve implementation which takes roughly $O(n^2)$ operations to compute, and a fast implementation which requires $O(n \log n)$ operations.

As will be shown in Chapter 6 the naïve implementations of the transforms are fairly straight forward. However, the fast implementations, which are in the most cases recursive, bare problems. Feldspar does not support recursion, which means that many efficient recursive algorithms should be implemented in an iterative manner or the core language code should be unrolled at compile time. Chapter 6 describes some important transforms and their possible implementations in Feldspar.

### 4.2.2 Algebraic Descriptions

Another project in the area of digital signal processing is the Spiral project [11]. Spiral makes efficient, high-level implementations for transforms possible using the Signal Processing Language (SPL). Most efficient algorithms for transforms

are based on their recursive algebraic descriptions. In SPL these recursive descriptions are used to implement transforms maintaining the mathematical structure (Chapter 7).

These mathematical descriptions for transforms look promising, they have a high level mathematical structure and they compute the result efficient like the fast implementations of transforms. Chapter 7 investigates these algebraic descriptions for transforms and introduces a matrix module for Feldspar based on them. The goal of this module is to implement the transforms using an algebraic style and generate efficient code.

# Chapter 5

# Combinators for Feldspar

A combinator is a general function which provides a higher level of abstraction for the programmer. This can for example be achieved when the combinator introduces a syntactic structure from another language to Feldspar. An example of such a combinator in Feldspar is the function `while` (Chapter 3.1.3). Another example of a combinator is the function `fold` in Feldspar. This function does not introduce syntactic structure from another language to Feldspar, however it simplifies certain sequences of operations. The function `fold` applies a binary function to the state and each value of an input list, passing the state from left to right, reducing the list to a single value, namely the final state: e.g. `fold (+) 7 (1...5)` results in the sum of the values 1 to 5 plus the initial value 7.

In this chapter I will design and implement several combinators for filters and other filter-like digital signal processing functions. Implementing filters and filter-like functions should be straightforward for experts in the digital signal processing domain using these combinators. First, I will discuss a combinator which provides *feedforward*, a function which uses previous input values (Figure 4.1) to compute the next output value. Next, a combinator which provides *feedback*, a function which uses previous computed output values to compute the next output value, is discussed. Finally, combinators which provide both, feedforward and feedback, are explored.

## 5.1 Feedforward Combinators

A feedforward combinator is a function which reuses previous input values, see Figure 4.1, to compute an output value. A good example of a function which needs a feedforward combinator is the *Moving Average* (MA) filter (Equation 2.1).

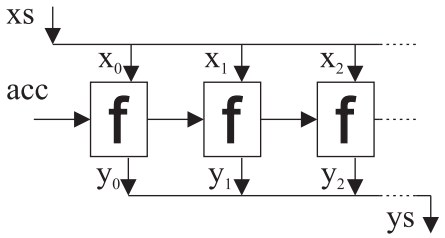


Figure 5.1: mapAccum: given the input list `xs`, the accumulator `acc`, and the function `f` compute for each input value the output value using the function `f` and the current accumulator `acc`.

**Code Block 1** Implementation of the MA filter using the `mapAccum` combinator.

```
maA :: (Numeric a) => DVector a -> DVector a -> DVector a
maA xs bs = snd $ mapAccum fun acc xs
    where
      acc = 0          -- the accumulator is used as counter here.
      m = length bs    -- the length of the filter coefficient sequence
      fun iEnd _ = let subVecX = ((take (min m (iEnd+1))) .
                                                  (drop (max 0 (iEnd+1-m)))) xs
                   in (iEnd+1, scalarProd bs (reverse subVecX))
```

The MA filter computes for each output value a scalar product of the coefficients vector ($b_i$) with a part of the input vector ($x[n-P]$ to $x[n]$). This means that we need a function which is able to loop over a vector and has access to the current and previous input values ($x[n-P]$ to $x[n]$).

A feedforward combinator suitable for the MA filter is `mapAccum`. `mapAccum` has the same functionality as the `mapAccumL` function in Haskell; this function behaves like a combination of `map` and `fold`. It applies a user defined function to each element of the input list and in addition passes an accumulating parameter from the beginning to the end. At the end the combinator returns the output list and the final accumulator (Figure 5.1). Below is the type signature of the `mapAccum` combinator, the Feldspar code this combinator can be found in Appendix A.

```
mapAccum :: (Storable a, Computable acc, Storable b) =>
    (acc -> Data a -> (acc, Data b)) -> acc -> Vector (Data a) ->
                                                (acc, Vector (Data b))
```

Code block 1 shows an implementation of the MA filter using the `mapAccum` combinator. The accumulator is used to keep track of the current index of the output vector, and the function `fun` yields the scalar product of the filter coefficient sequence and the reversed input starting at the current index.

**Code Block 2** Implementation of the MA filter using the functions `map` and `tails`.

```
maM :: (Numeric a) => DVector a -> DVector a -> DVector a
maM xs bs = map (scalarProd bs) xsTails
    where
      n = length xs
      m = length bs
      xsTails = map (reverse . (take m))
                    (take n $ tails $ replicate (m-1) 0 ++ xs)
```

As one could anticipate, the core language code of `maA` (Appendix B.1) produced by the compiler contains two nested while loops. The outer loop loops over the elements of the input list, and the inner loop represents the scalar product of two vectors, the coefficient vector $b_i$ and the input values $x[n-P]$ to $x[n]$.

In case of the MA filter the `mapAccum` combinator might not be the best solution. The combinator does not give direct access to the previous input values in the user defined function, e.g. these values are not a parameter of the user defined function.

Another way to implement the MA filter is to use the functions `map` and `tails` (Code block 2). When compiled this results also in two nested loops in the core language (Appendix B.2). However, this time the outer loop is a parallel construct. An advantage of a parallel loop is that it can be parallelized, though the current version of Feldspar does not support parallel code yet. Currently everything is computed sequentially.

To test the equality and correctness of both implementations, `maA` and `maM`, some important examples can be checked by hand against the definition (Equation 2.1) and MATLAB [16] implementations. After that QuickCheck [17] was used for further testing. Below, the result of both implementations applied the same input, the QuickCheck property, and one test-run of the QuickCheck property are shown.

```
*Test> eval $ maA (0 ... 9) (vector [1,5,2,4,3])
(10,[0,1,7,15,27,42,57,72,87,102])
*Test> eval $ maM (0 ... 9) (vector [1,5,2,4,3])
(10,[0,1,7,15,27,42,57,72,87,102])

  -- QuickCheck property to check the equality of maA and maM
maQC :: [Int] -> [Int] -> Property
maQC xs bs = (P.length xs) P.> (P.length bs) ==> (maA' P.== maM')
    where
      maA' = eval $ maA (vector xs) (vector bs)
      maM' = eval $ maM (vector xs) (vector bs)
```

```
*Test> quickCheck maQC
+++ OK, passed 100 tests.
```

The `mapAccum` combinator is not ideal yet, the implementation of MA using `map` and `tails` (Code block 2) gives better results than the implementation using `mapAccum` (Code block 1). The Feldspar code of `maM` is more straightforward and the generated core language code is more efficient. Later in this Chapter I will show another implementation of the MA filter, `maLoop`, using a different combinator. The function `maLoop` is not better than `maM` when looking at the generated core language code; however, the Feldspar code is even more straightforward than `maM`.

For now, the `mapAccum` combinator does not seem promising. The reason for this is that there are several other function which provide a structure to iterate over the input vector (e.g. `map` and `for`). The only thing we used the combinator for, in `maA`, was to store the index. Next Section will discuss functions which need feedback, thus in each iteration the function has to reuse previous computed output values.

## 5.2 Feedback Combinators

In contrast to the feedforward combinator a feedback combinator provides access to the previous computed output values. A good example of a digital signal processing function which uses feedback is the *Autoregressive* (AR) model (Equation 2.2). AR reuses previous computed output values to compute the next output value. A common way to access previous computed values is recursion; however, this is not possible in Feldspar. The core language, to which Feldspar code is translated, does not support run-time recursion. Therefore the combinator should provide access to the previous computed values.

When using the `mapAccum` combinator, from the previous section, to implement AR, the previous computed output values have to be stored in the accumulator such that they can be accessed in later iterations. This means that the accumulator acts like a buffer where, during each iteration, a new value is added to the front and the last value is removed. It is not possible to generate efficient code from a function using the accumulator of the `mapAccum` as a buffer. When a vector, which changes every iteration, is stored in the accumulator, the whole vector has to be rewritten in memory each iteration. Therefore we need a combinator which keeps track of the previously computed values and which generates efficient core language code.

The first approach to provide access to values computed in previous iterations is the combinator `feedbackLoop`. This combinator has a user defined function `body` and an integer, which represents the length, as arguments and returns a list of

**Code Block 3** AR implemented using the `feedbackLoop` combinator.

```
arFeedLoop :: (Numeric a) => DVector a -> DVector a -> DVector a
arFeedLoop xs as = feedbackLoop body $ length xs
    where
      body = \ys i -> (xs ! i) - (scalarProd as $ reverse $ take i ys)

*Test> eval $ arFeedLoop (0 ... 9) (replicate 5 1)
(10,[0,1,1,1,1,1,1,2,2,2])
```

output values. The function `body` computes, given the output values of previous iterations (the value at the current index is undefined) and the current index, the new output value. Similar to a for-loop, the `feedbackLoop` iterates over a list using indices, but additionally the `feedbackLoop` also has access to previous computed output values which are given as a parameter to the user defined function `body`. The code for `feedbackLoop` can be found in Appendix A.

```
feedbackLoop :: (Storable y) =>
    (DVector y -> Data Int -> Data y) -> Data Int -> DVector y
```

Code block 3 shows an implementation of the AR model using the `feedbackLoop`. In this implementation, the function `body` represents the equation of the current input value minus the scalar product of the previous output values and the coefficient vector. Code block 3 also contains the evaluation of `arFeedLoop` applied to example input.

Compilation of the code in Code Block 3 to the core language yields the result as depicted in Appendix B.4. The resulting code contains two nested loops as one would expect; the outer loop iterates over the input vector and the inner loop computes the scalar product.

The first two sections of this Chapter have shown that it is possible to reuse previous input values like in the MA and that it is possible to reuse previous output values like in the AR. The next step is to do both, feedforward and feedback, in a single combinator.

## 5.3 Feedforward and Feedback Combinators

A good example of a function containing feedforward and feedback is the *autoregressive moving average* (ARMA) model which is a combination of AR and MA and uses both, previous input values and previous output values (Equation 2.3).

In the previous section we have seen how AR can be conveniently implemented

**Code Block 4** ARMA implemented using the `ffLoop` combinator.

```
arma :: (Numeric a) => DVector a -> DVector a -> DVector a -> DVector a
arma inputxs bs as = ffLoop body inputxs
    where
      body = \ys xs -> (scalarProd bs xs) - (scalarProd as (tail ys))
```

with the `feedbackLoop`. Similarly, it is possible to implement the feedback part of ARMA with this combinator. The accumulator of the `feedbackLoop` can be used to store the current index of the iteration. With this index the feedforward part of ARMA can be implemented similar to the implementation of MA using the `mapAccum` combinator. As explained above it is possible to use the previous combinator to implement ARMA, it is not as straightforward as we want to achieve with combinators.

To make the implementation of the ARMA easier a new combinator called `ffLoop` is designed. This combinator provides access to the previous input values and previous computed output values. The `ffLoop` combinator has a function `body` and the input vector as arguments. The function `body` is a user defined function which has a vector with the previous output values `ys` and a vector with the current and the previous input values `xs` as arguments; and the function `body` returns a new output value as result. In the user defined function `body` the arguments `ys` and `xs` are reversed: this means that the head of `xs` is the current input value and the head of `ys` is undefined since it is computed in the current iteration. Below is the type definition of the `ffLoop` combinator. The code for `ffLoop` can be found in Appendix A.

```
ffLoop :: (Storable y) =>
    (DVector y -> DVector x -> Data y) -> DVector x -> DVector y
```

ARMA can be implemented fairly easily using `ffLoop` as shown in Code block 4. Furthermore, the other filters used in this Chapter, MA and AR, can also be implemented using the `ffLoop` combinator (Code Block 5). Their Feldspar code is more straightforward than the code using the `feedbackLoop` or `mapAccum` combinators; without loosing efficiency in the generated core language code (Appendix B.3 and B.5) compared to the core language code generated by the the previous implementations (Appendix B.1 and B.4).

**Code Block 5** MA and AR implemented using the `ffLoop` combinator.

```
ma xs bs = ffLoop body xs
    where
      body _ xs = (scalarProd bs xs)

ar xs as = ffLoop body xs
    where
      body ys xs = head xs - (scalarProd as $ tail ys)
```

## 5.4 Generalization

The combinator `ffLoop`, discussed in the previous Section, allows straightforward implementation and the generation of efficient code for several filters. However, this combinator is not very general: it can only be used when feedforward or feedback is required. Storing other accumulating values during the computation is not possible.

The Turbo encoder (Section 2.4.1 and Figure 2.3) is an example where the `ffLoop` is not sufficient. During the computation of two of the three output vectors of the turbo encoder, intermediate values have to be stored.

A more general combinator is the `streamState` combinator. This combinator is very similar to the `ffLoop`, however in the user defined function there is a big difference. The user defined function of `streamState` has access to a user defined buffer instead of the previous computed output values. This buffer represents a vector of finite length. Each iteration a new element is added to the buffer, and the last element is removed from the buffer.

```
streamState :: (Storable b, Storable c) =>
    (DVector a -> DVector b -> (Data c, Data b)) ->
       DVector b -> DVector a -> (DVector c, DVector b)
```

The combinator `streamState` has three arguments: the first argument is a user defined function, the second argument is an initial buffer, and the last argument is the input vector. At the end, the combinator returns the output vector containing the result of each iteration and the final buffer. The user defined function has two arguments, namely the previous input values and the buffer, and it returns a tuple containing a new output value and a new element for the buffer.

Similar to `ffLoop`, `streamState` iterates over the input list and provides access to previous input values. However, the function in `streamState` does not have direct access to the previous computed output values, but they can be stored in the buffer when needed. The code for `streamState` can be found in Appendix A.

**Code Block 6** ARMA implemented using the `streamState` combinator.

```
armaStream :: (Numeric a) => DVector a -> DVector a -> DVector a -> DVector a
armaStream xs bs as = fst $ streamState body initBuf xs
        where
          initBuf = replicate (length as) 0  -- initiate the buffer with 0s
          body xs ys = (newY, newY)          -- (new output, new buffer value)
            where
              newY = (scalarProd bs xs) - (scalarProd as (reverse ys))
```

To check the efficiency of the core language code generated by a function using the `streamState` combinator, I implemented ARMA using this combinator (Code block 6) and compared the resulting core language code (Appendix B.7) to the core language code of `arma` (Appendix B.6).

When comparing the complexity (counting the number of iterations of the loops) both combinators perform similar. They both have a large `while`-loop containing two small loops computing the two scalar products. However the core language code using the `streamState` combinator has one extra loop in the beginning, initializing the buffer, and some extra (constant time) operations, which handle the circular buffer in `streamState`.

## 5.5 Conclusion

Whenever a DSP function does not use feedback, a simple combinator like `map` usually suffices. However, when feedback is required, designing a good combinator gets significantly more complicated because the combinator has to provide access to previous computed output values and still generate efficient code.

In this Chapter, we have explored several combinators: `mapAccum`, `feedbackLoop`, `ffLoop`, and `streamState`. The combinators `ffLoop` and `streamState` seem the most efficient and applicable for filter like functions. They raise the level of abstraction with straightforward implementation of the filters and they generate efficient core language code. However, they are still not perfect. When using the `ffLoop` combinator it is not possible to use a user defined accumulator (or buffer), and on the other hand, the combinator `streamstate` lets the user define a buffer, however this is only one buffer and we might need more buffers or accumulating values.

An ideal combinator for DSP functions would be one which iterates over the input vector using a user defined function and returns the output vector. One of the arguments to the user defined function should be a list of buffers which could for example contain the previous input and output values. One could even consider

that the combinator should return several output vectors, which is required for example in the turbo encoder.

The correctness of the implementations used in this chapter, is checked by comparing the results, of Feldspar functions, with the results generated by MATLAB [16]. After that, QuickCheck is used to test if the different implementations, used in this chapter, generate the same result. Appendix C shows some of these QuickCheck properties.

# Chapter 6

# Transforms

Transforms are important in digital signal processing; they transform signals into signals with other properties. The algorithms describing the transforms can be put in two categories, naïve implementations and fast implementations. DSP systems often have limited resources which means that the fast implementations have preference above the naïve implementations; although, there is one problem: many of the fast transforms are written recursively and Feldspar does not support run-time recursion. This means that recursive functions need to be written iteratively or the recursive function should be unrolled at compile time. The latter we will call compile time recursion in this thesis.

After a short introduction to compile time recursion, this chapter will describe several implementations of DSP transforms in Feldspar.

**Compile Time Recursion** A simple example of compile time recursion is shown in the function `myExpo`. It represents a recursive implementation of the exponentiation function. `myExpo` raises the value `x` to the power of `n`.

```
myExpo :: Int -> Data Int -> Data Int
myExpo 0 x = 1
myExpo n x = x * (myExpo (n Prelude.- 1) x)
```

The type signature at the first line contains both Haskell and Feldspar parameters. The first parameter `n` is a Haskell variable, which means that the value should be known at compile-time such that the code can be unrolled. The core language code of the function `myExpo`, where in this case the variable `n` has been replaced by the value 8, is a short iterative program:

```
*Code> printCore $ myExpo 8
program v0 = v8
```

```
  where
    v1 = v0 * 1
    v2 = v0 * v1
    v3 = v0 * v2
    v4 = v0 * v3
    v5 = v0 * v4
    v6 = v0 * v5
    v7 = v0 * v6
    v8 = v0 * v7
```

The number of lines of the core language code increases linearly with the value of the exponent. A slightly more efficient implementation, in lines of code and in the number of computations, of the exponentiation function is `mySmartExpo`. It generates between $\log n + 1$ and $2 \log n + 1$ lines of code, where $n$ is the value of the exponent.

```
mySmartExpo :: Int -> Data Int -> Data Int
mySmartExpo 0 x = 1
mySmartExpo n x | even n = y * y
                | odd n  = y * y * x
  where y = mySmartExpo (Prelude.div n 2) x

*Code> printCore $ mySmartExpo 8
program v0 = v4
  where
    v1 = 1 * v0
    v2 = v1 * v1
    v3 = v2 * v2
    v4 = v3 * v3
```

## 6.1 Fourier Transform

### 6.1.1 Discrete Fourier Transform

The discrete Fourier transform (DFT), Equation 2.4, transforms a signal from the time domain into the frequency domain, thereby revealing the frequencies and their phase used in the input signal. Below is a naïve implementation of the DFT in Feldspar[1].

[1] At the time of writing the complex numbers were not part of Feldspar. Therefor I used an experimental module similar to the complex numbers module in Haskell.

```
1   -- The Discrete Fourier Transform.
dft :: DVector RealNum -> DVector ComplexReal
3 dft xs = map sumFun $ map intToReal (0...(ln-1))
    where
5     ln = length xs
        -- sumFun k = ∑_{n=0}^{ln}(ex x_n (k*n))
7     sumFun k = sum $ zipWith ex xs (map ((*k) . intToReal) (0...(ln-1)))
        -- ex x_n kn = x_n * e^{(-2πi/N) kn}
9     ex xn kn = mkPolar xn (-2 * (floatToReal pi) * kn / (intToReal ln))
```

This implementation of the DFT is very close to its mathematical formula (Equation 2.4). Line 3 of the code iterates over the indices of the output vector. The function `sumFun`, on line 6, represents the sum of the equation and sums over `xs` and its indices using the function `ex`. Function `ex` computes a complex value given the current index in the output vector, the current index in the input vector, and the current value of the input vector.

The discrete Fourier transform can also be represented as a matrix vector multiplication $y = Mx$ where $y$ is the result vector of the multiplication of the transform matrix M with the input vector $x$ [11]. The mathematical Equation is defined as follows:

$$DFT_n = \left[e^{\frac{-2\pi ikl}{n}}\right] \quad 0 \leq k, l < n$$

Below the Feldspar implementation of the DFT using this matrix description.

```
dftm :: DVector ComplexReal -> DVector ComplexReal
dftm xs = mat ** xs
    where
      n = length xs
      mat = indexedMat n n (\k l -> twiddle n (k*l))

twiddle :: Data Int -> Data Int -> Data ComplexReal
twiddle n m = mkPolar 1 (-2 * (floatToReal pi) *
                                    (intToReal m) / (intToReal n))
```

In this implementation the transform matrix $M$ is represented by `mat` and it is multiplied with the input vector `xs`. The transform matrix is constructed using the function `indexedMat` with the size of the input vector and the *twiddle* factor, $\omega_n^{kl} = e^{\frac{-2\pi ikl}{n}}$, as indexing function.

Both naïve implementations above generate core language code (Appendix D) which use $O(n^2)$ operations. A more efficient algorithm, the Cooley–Tukey algorithm, is shown in the next section.

### 6.1.2 Fast Fourier Transform

A fast Fourier transform (FFT) is an efficient algorithm for computing the discrete Fourier transform. The FFT uses $O(n \log n)$ instead of $O(n^2)$ operations as the DFT. The Cooley-Tukey algorithm is a recursive FFT algorithm. Below is an implementation in Feldspar which uses compile time recursion.

```
  -- Radix-2 Recursive FFT implementation of the Cooley-Tukey algorithm.
  -- The length of the input vector xs, namely n, should be known at
  -- compile time and n should be a power of 2.
fft :: Int -> DVector RealNum -> DVector ComplexReal
fft n xs = ffth n (map (\r -> r |+| 0) xs)

  -- Helperfunction for the FFT using complex numbers.
ffth :: Int -> DVector ComplexReal -> DVector ComplexReal
ffth 1 xs = replicate 1 $ head xs
ffth n xs = (memorize . uncurry (++) . unzip) $
                            zipWith btfly evenl (zipWith (*) oddl exs)
  where
    nVal  = value n
      -- ffth of the even indexed elements
    evenl = ffth (Prelude.div n 2) $ getEven xs
      -- ffth of the odd indexed elements
    oddl  = ffth (Prelude.div n 2) $ getOdd xs
      -- twiddle factors k = 0,...,N/2 - 1
    exs   = (map (expVal . intToReal) (0...(nVal `div` 2 - 1)))
      -- expVal k = e^(-2πi/N k)
    expVal k = mkPolar 1 (-2 * (floatToReal pi) * k / (intToReal nVal))

  -- Butterfly
btfly :: (Numeric a) => Data a -> Data a -> (Data a, Data a)
btfly x y = (x+y, x-y)

  -- Return the even indexed elements of the input vector.
getEven :: DVector a -> DVector a
getEven xs = indexed ((length xs) `div` 2) (\i -> xs ! (i*2))

  -- Return the odd indexed elements of the input vector.
  -- Note: the length of the list should be even.
getOdd :: DVector a -> DVector a
getOdd xs = indexed ((length xs) `div` 2) (\i -> xs ! (i*2+1))
```

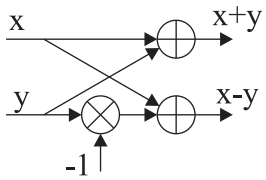


Figure 6.1: Butterfly network

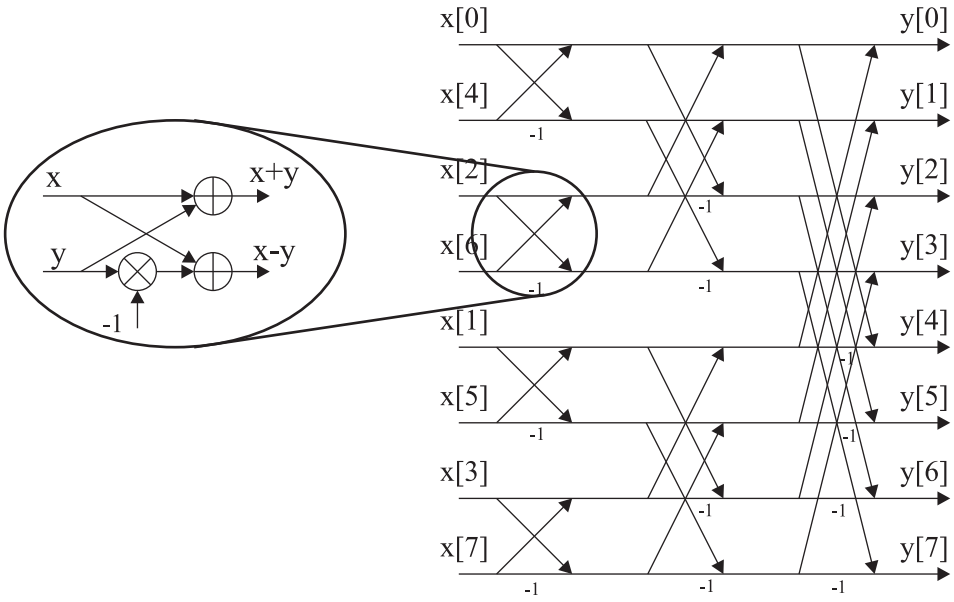


Figure 6.2: Fast Fourier Transform of size 8: showing the butterfly operations.

This implementation contains of several functions: `fft` is the main function which calls `ffth` with the complex valued input vector and returns the result, `ffth` is the recursive helper function which performs the actual computations, `btfly` represents the butterfly operation (Figure 6.1), `getEven` and `getOdd` return respectively the even and the odd indexed elements from a given list. The recursive helper function `ffth` first computes recursively the FFT of the even and odd indexed elements after which it combines the results using the butterfly operation. Figure 6.2 shows how the FFT of size 8 is build up out of butterfly operations.

The function `memorize` is used to store the intermediate result vectors in memory. Without this function the intermediate results are recomputed every time they are needed which is not very efficient (Section 3.2.1).

The core language code generated from the function `fft` has a complexity of $O(n \log n)$ when we look at the loops and count their iterations which is as we expected. Appendix D.3 shows the core language code for the FFT of size 8.

However the compiler currently produces lots of code: each loop contains an if statement with a butterfly operation and the computation of its twiddle factor. Since the loops are all fairly similar this is a point which should be considered for optimization.

In Lava [18] (Chapter 8) a slightly different approach for implementing the FFT is chosen. They have introduced several combinators which take care of splitting up the input vector, before going into the recursive steps, and putting the results back together, after computing the recursive steps. A similar approach should be possible in Feldspar, however I chose to keep my implementation as close as possible to the Cooley–Tukey algorithm.

## 6.2 Discrete Cosine Transform

### 6.2.1 Discrete Cosine Transform

The discrete cosine transform (DCT), Equation 2.5, is a transform like the DFT however it uses only real valued numbers. The DCT can be represented as a matrix vector multiplication $y = Mx$ where $y$ is the result vector of the multiplication of the transform matrix $M$ with the input vector $x$ [11]. The transform matrix of the DCT-II is defined as follows:

$$DCT\text{–II} = \left[\cos \frac{k(2l+1)\pi}{2n}\right] \quad 0 \leq k, l < n$$

where $k$ and $l$ represent the row and column of the matrix and $n$ the length of the input vector.

Below is an implementation of the DCT–II using matrix vector multiplication in Feldspar.

```
  -- Discrete Cosine Transform type-II
dct2 :: (DVector Float) -> (DVector Float)
dct2 xn = mat ** xn
    where
      n   = length xs
      mat = indexedMat n n (\k l -> dct2nkl n k l)

  -- Helper function defining all the values in the DCT-2n matrix
dct2nkl :: Data Int -> Data Int -> Data Int -> Data Float
dct2nkl n k l = cos ( (k' *(2*l' +1)*pi)/(2*n') )
  where
    (n',k',l') = (intToFloat n, intToFloat k, intToFloat l)
```

In this implementation `xn` is the input vector and `mat` is the transform matrix. The transform matrix is constructed using the function `indexedMat` with the size of the input vector and the indexing function `dct2nkl`. `dct2nkl` returns a value in the transform matrix given the size of the matrix, the row, and the column.

The matrix multiplication description of the DCT looks intuitive, short, and functions well as a specification. However, the disadvantage is that it takes $O(n^2)$ operations to compute.

### 6.2.2 Fast Cosine Transform

A more efficient implementation of the DCT, using $O(n \log n)$ operations, can for example be achieved by looking at the algebraic structure of the transform matrix. An interesting project about this is the Spiral project [11]. In Chapter 7 I will present the algebraic structure of transform matrices and the Spiral project.

There are also other efficient algorithms for computing the DCT: for instance the Fast Cosine Transform (FCT) or using the FFT. The FCT algorithms are recursive which means that they need to be implemented using compile time recursion. I have not implemented an FCT algorithm or an algorithm computing the DCT using the FFT in Feldspar.

## 6.3 Walsh–Hadamard transform

### 6.3.1 Walsh–Hadamard transform

The Walsh–Hadamard transform (WHT), Equation 2.6, is a generalized Fourier transform. The transform matrix is build out of size-2 DFTs and contains only the values 1 and −1. A straight forward way of implementing the WHT in Feldspar is to first generate the transform matrix and then multiply it with the input vector, as was done with the DCT above.

```
  -- Walsh-Hadamard transform
wht :: DVector Float -> DVector Float
wht xs = mat ** xs
  where
    n   = length xs
    mat = indexedMat n n (\k l -> intToFloat ((-1)^(countbwa k l)))
    countbwa k l = bitCount $ k .&. l
```

In this implementation, `xs` is the input vector and `mat` is the transform matrix which depends on the size of the input vector. The function `countbwa` computes a bitwise-and and counts the number of ones in the result.

Here the matrix multiplication description of the WHT looks again very intuitive, but the generated code contains 2 loops. The outer loop is a parallel construct and the inner loop is a while loop. This means that this implementation uses $O(n^2)$ operations to compute the result.

### 6.3.2 Fast Walsh–Hadamard transform

The Fast Walsh–Hadamard transform (FWHT) is an efficient algorithm to compute the WHT. It computes the WHT in $O(n \log n)$ operations instead of $O(n^2)$ as in the naïve implementation (shown above).

```
  -- Fast Walsh-Hadamard transform
fwht :: Int -> DVector Float -> DVector Float
fwht 1 xs = replicate 1 $ head xs
fwht n xs = (memorize . uncurry (++) . unzip) $ zipWith btfly front back
    where
      nVal  = value n  -- Haskell int to Feldspar int
      front = fwht (P.div n 2) $ take (nVal `div` 2) xs
      back  = fwht (P.div n 2) $ drop (nVal `div` 2) xs

  -- Butterfly
btfly :: (Numeric a) => Data a -> Data a -> (Data a, Data a)
btfly x y = (x+y, x-y)
```

The FWHT function in Feldspar is a compile time recursive function from which the size of the input vector should be known at compile-time such that the compiler can unroll the function. The size of the input vector should also be a power of two. In the recursive step the input vector is divided into halves after which the WHT of each half is computed. The halves, `front` and `back`, are put back together using the butterfly operation as done in the FFT. The core language code generated from the `fwht` uses $O(n \log n)$ operations as expected from a fast algorithm.

Since the WHT has a nice algebraic structure, being built out of size-2 DFTs, it is also possible to implement an efficient algorithm based on this property [12].

$$\mathrm{WHT}_{2^n} = \overbrace{\mathrm{WHT}_2 \otimes \ \ldots \ \otimes \mathrm{WHT}_2}^{n}$$

Here the WHT transform matrix of size $2^n$ is consists of $n-1$ tensor products ($\otimes$) of the $\mathrm{WHT}_2$, where $\mathrm{WHT}_2$ is equal to $\mathrm{DFT}_2$ [12]. For matrices the tensor

product is usually called the Kronecker product. In Chapter 7 I will explain what a tensor product is.

## 6.4 Conclusion

As seen from the three different transforms mentioned in this Chapter the naïve implementations have high level mathematical structures. However, they are not efficient when it comes to the number of operations the core language code performs, which is $O(n^2)$. On the contrary the fast algorithms, shown above, are efficient. The core language code generated performs $O(n \log n)$ operations, however here the high level mathematical structure is not visible anymore. Next Chapter presents transforms written as recursive matrices. These transform matrices have a high level mathematical description and should still generate efficient core language code.

# Chapter 7

# Matrices

Fast algorithms for computing transforms, like the FFT (Chapter 6), are based on the mathematical formulas of the transforms. However the mathematical structure in the implementation of these algorithms is not easy to recognize. Therefore we want to write these transforms using high level mathematical constructs, while at the same time allowing the generation of efficient code.

The formulas of the transforms, given in Chapter 2.4, can be rewritten as recursive matrix factorizations. Table 1 contains the matrix descriptions of the transforms used in this report. These algebraic descriptions come from [11], [12], and [19].

The bold font indicates the transform matrices, for example the DFT of size $n$ is shown as $\mathbf{DFT}_n$. The transform matrices can be recursively broken down into transforms of smaller sizes, generic matrices, and symbols; which are connected using matrix operations like multiplication ($AB$ or $A \cdot B$), direct sum ($A \oplus B$), and tensor product (or Kronecker product for matrices) ($A \otimes B$). The last two matrix operations are defined by:

$$A \oplus B = \begin{bmatrix} A & \\ & B \end{bmatrix}, \quad \text{and}$$
$$A \otimes B = [a_{k,l}B], \quad \text{where } A = [a_{k,l}].$$

The lower left and the upper right quadrant of the direct sum matrix contain only zeros. Table 2 contain the descriptions of the generic matrices and symbols used in the recursive descriptions of transforms. When an element in a matrix is not defined the value is zero.

The next section describes SPL, the Signal Processing Language of the Spiral project [11]. The section thereafter will show the matrix module I designed for Feldspar. Using this matrix module it is possible to express the algebraic transform

**Table 1** Algebraic descriptions of transforms.

$$\begin{aligned}
\mathbf{DFT}_n &= (\mathbf{DFT}_k \otimes I_m)\ T^n_m\ (I_k \otimes \mathbf{DFT}_m)\ L^n_k, && n = km \\
\mathbf{DFT}_2 &= F_2 \\
\mathbf{DCT\text{-}2}_n &= L^n_m\ (\mathbf{DCT\text{-}2}_m \oplus \mathbf{DCT\text{-}4}_m)\ (F_2 \otimes I_m)\ (I_m \oplus J_m), && n = 2m \\
\mathbf{DCT\text{-}2}_2 &= \mathrm{diag}(1, 1/\sqrt{2})\ F_2 \\
\mathbf{DCT\text{-}4}_n &= S_n\ \mathbf{DCT\text{-}2}_n\ \mathrm{diag}\left(\frac{1}{\cos\frac{2k+1}{4n}}\right), && 0 \le k < n \\
\mathbf{DCT\text{-}4}_2 &= J_2\ R_{13\pi/8} \\
\mathbf{WHT}_n &= (\mathbf{WHT}_2 \otimes I_m)\ (I_2 \otimes \mathbf{WHT}_m), && n = 2m \\
\mathbf{WHT}_2 &= F_2
\end{aligned}$$

descriptions in Feldspar. This Chapter will end by showing several implementations of transforms in Feldspar and a short conclusion.

## 7.1 Spiral

Spiral [11] is a project for automatic generation and code optimization of DSP algorithms for various hardware platforms. In Spiral the mathematical formulas are represented in the language SPL [20]. Below I will shortly discuss SPL, but I will not go into other parts of the Spiral project, for more information refer to [11].

SPL [20] is a language for describing matrix factorizations; in particular it can be used for describing the transforms shown in Table 1. SPL expressions consists of generic matrices, symbols, and transforms; and may involve several matrix operations including composition, direct sum, and tensor product.

Table 3 contains the algebraic description of the $\mathbf{DFT}_{16}$ and the corresponding SPL program [11].

Algorithms for the SPL language are captured and generated by *rules*. There are two types of rules in Spiral: breakdown rules and manipulation rules. A breakdown rule defines how a transform can be decomposed into a product of smaller matrices. An example of a breakdown rule is:

$$\mathbf{DCT\text{-}2}_4 = L^4_2\ (\mathbf{DCT\text{-}2}_2 \oplus \mathbf{DCT\text{-}4}_2)\ (F_2 \otimes J_2)\ (I_2 \oplus J_2)$$

**Table 2** Generic matrices and symbols used in the transforms in Table 1.

$$
\begin{aligned}
I_n &= \text{The identity matrix of size } n \times n. \\
J_n &= \text{The row-reversed identity matrix of} \\
&\quad \text{size } n \times n. \\
\operatorname{diag}(a_0, \ldots, a_{n-1}) &= \text{Diagonal matrix: } a_0, \ldots, a_{n-1} \text{ are} \\
&\quad \text{the diagonal entries of the matrix.}
\end{aligned}
$$

$$
T_k^n = \begin{array}{l} \operatorname{diag}(\ \mathrm{tw}_n^{0\cdot 0}, \ldots, \mathrm{tw}_n^{(k-1)\cdot 0}, \\ \qquad \mathrm{tw}_n^{0\cdot 1}, \ldots, \mathrm{tw}_n^{(k-1)\cdot 1}, \\ \qquad \vdots \\ \qquad \mathrm{tw}_n^{0\cdot (n/k-1)}, \ldots, \mathrm{tw}_n^{(k-1)\cdot (n/k-1)}) \end{array} \qquad \textit{(Twiddle matrix)}
$$

$$
\mathrm{tw}_n^k = e^{-2\pi k i/n} \qquad \textit{(Twiddle factor)}
$$

$$
L_k^n = i\frac{n}{k} + j \longmapsto jk + i \quad \text{s.t.} \qquad \textit{(Stride perm. matrix)}
$$

$$
\begin{aligned}
&0 \le i < k, \\
&0 \le j < \frac{n}{k}
\end{aligned}
$$

$$
S_n = \begin{bmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & 1 & 1 \\ & & & & 1 \end{bmatrix}
$$

$$
F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \qquad \textit{(Butterfly matrix)}
$$

$$
R_\alpha = \begin{bmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{bmatrix} \qquad \textit{(Rotation matrix)}
$$

**Table 3** The algebraic description of the $\mathbf{DFT}_{16}$ and the corresponding SPL program

$$\mathbf{DFT}_{16} = (\mathbf{DFT}_4 \otimes I_4)\ T_4^{16}\ (I_4 \otimes \mathbf{DFT}_4)\ L_4^{16},$$
$$\mathbf{DFT}_4 = (\mathbf{DFT}_2 \otimes I_2)\ T_2^4\ (I_2 \otimes \mathbf{DFT}_2)\ L_2^4$$

```
(define F4
  (compose (tensor (F 2) (I 2)) (T 4 2) (tensor (I 2) (F 2)) (L 4 2)))
#subname fft16
  (compose (tensor F4 (I 4)) (T 16 4) (tensor (I 4) F4) (L 16 4))
```

A manipulation rule defines how a SPL formula, which does not contain any transforms, can be manipulated in a different SPL formula, for example resulting in less computations. An example of a manipulation rule is:

$$L_n^{kmn} = (L_n^{kn} \otimes I_m)\ (I_k \otimes L_n^{mn})$$

These rules are used to optimize the SPL formulas in the implementation and code optimization level of Spiral, more information about how these rules are used can be found in [11].

## 7.2 Feldspar Matrix Module

The idea of using high level algebraic descriptions of transforms in combination with the generation of efficient core language code is appealing. Therefore I designed a matrix module for Feldspar which allows the writing of the algebraic descriptions of transforms in Feldspar and simultaneously the generation of efficient code.

A matrix in this new matrix module has type `Matr a` where `a` represents the type of the values in the matrix. For example the type of `a` can be integer `Int`, float `Float`, or complex `ComplexReal`.

Based on the algebraic descriptions, shown in Table 1, the following seven matrices were identified for implementation:

- The identity matrix ($I_n$) : `IdMatr h w`
  where `h` and `w` are the height and width of the matrix.

- The row-reversed identity matrix ($J_n$): `JdMatr h w`
  where `h` and `w` are the height and width of the matrix.
- The diagonal matrix ($diag(a_0, \ldots, a_{n-1})$): `Diag h w xs`
  where `h` and `w` are the height and width of the matrix and `xs` contains the values of the diagonal.
- The result of a tensor product ($A \otimes B$): `Tensor h w A B`
  where `h` and `w` are the height and width of the matrix; and `A` and `B` are the arguments of the tensor product.
- The result of a direct sum ($A \oplus B$): `DirectSum h w A B`
  where `h` and `w` are the height and width of the matrix; and `A` and `B` are the arguments of the direct sum.
- The result of a matrix multiplication ($A \cdot B$): `Compose h w A B`
  where `h` and `w` are the height and width of the matrix; and `A` and `B` are the matrices which are multiplied.
- A general matrix: `IndexedMatr h w ix`
  where `h` and `w` are the height and width of the matrix; and `ix` is the indexing function of the matrix. The function `ix` is comparable to the function `indexedMat` explained in Section 6.2.1.

For each of the matrices above there is a function to construct the matrix.

```
idMatr :: (Numeric a) => Data Int -> Matr a
jdMatr :: (Numeric a) => Data Int -> Matr a
diagMatr :: (Numeric a) => DVector a -> Matr a
tensor :: (Numeric a) => Matr a -> Matr a -> Matr a
directSum :: (Numeric a) => Matr a -> Matr a -> Matr a
compose :: (Numeric a) => Matr a -> Matr a -> Matr a
indexedMatr :: (Numeric a) =>
     Data Int -> Data Int -> (Data Int -> Data Int -> Data a) -> Matr a
```

Using these functions it is possible to construct the algebraic transform matrices from Table 1, given that the transform specific symbols, like $T^n_k$ and $L^n_k$, are implemented.

However this matrix module does not compute anything yet, it just makes it possible to represent the transform matrices in Feldspar. To compute a transform, the corresponding transform matrix should be multiplied with the input vector. The function `(**)` is a generic function which multiplies matrices ($M_c = M_a M_b$), multiplies vectors ($a = xy$), and multiplies a matrice with a vector ($y = Mx$)

and ($y = xM$). In this thesis the transforms are written as a multiplication of a transform matrix with an input vector ($y = Mx$); therefore, I will only use the function `mulMatrVec` which is a specific instance of `(**)` for computing $y = Mx$.

```
mulMatrVec :: (Numeric a) => Matr a -> DVector a -> DVector a
```

To multiply the transform matrix with the input vector efficiently we need to use the structure of the transform matrix. Instead of first computing the complete transform matrix and multiplying the result with the input vector, the input vector should first be multiplied with the components of the transform matrix after which their results are combined into the output vector. By first multiplying the input vector with the components of the transform matrix and then combining the result, less operations are performed in total.

Each type of matrix has different properties; and therefore a different way of multiplying a matrix with a vector. For example multiplying the identity matrix with the input vector results in an output vector which is identical to the input vector. Applendix E shows the implementation of the Matr module including a short explanation of the `mulMatrVec` function for each of the different matrices.

Now, transform matrices can be expressed in Feldspar and transforms, the transform matrix multiplied with the input vector, can be evaluated. Moreover, core language code can be generated from transforms. In the next section I will show the implementations of the transforms in Table 1 in Feldspar.

## 7.3 Transforms in Feldspar

In this section I will implement the transforms, shown in Table 1, using the new matrix module in Feldspar.

### 7.3.1 Discrete Fourier Transform

In Feldspar the DFT is a direct translation from the algebraic formula. The DFT breaks recursively down into smaller instances of the DFT, the identity matrix, the twiddle matrix, and the stride permutation matrix.

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m)\ T^n_m\ (I_k \otimes \mathbf{DFT}_m)\ L^n_k, \qquad n = km$$

```
  -- The discrete Fourier transform
dfts :: Int -> (DVector ComplexReal) -> (DVector ComplexReal)
```

```
dfts n xn = (dftn n) ** xn

  -- The discrete Fourier transform matrix
dftn :: Int -> Matr ComplexReal
dftn 2 = f2c
dftn n = compose
            (compose (tensor (dftn kh) (idMatr mf)) (twiddleMatr nf mf))
            (compose (tensor (idMatr kf) (dftn mh)) (strideMatrc nf kf))
    where
      nf = value n
      mh = n `Prelude.div` kh
      mf = nf `div` kf
      kh = 2
      kf = value kh
```

The function `dfts` multiplies the transform matrix, generated by the function `dftn`, with the input vector `xn`. The function `dftn` is a compile time recursive function and therefor the size of the input vector should be known at compile time. The implementations of the stride permutation matrix ($L^n_k$), twiddle matrix ($T^n_m$), and the DFT of size two ($F_2$) can be found in Appendix F.1.

### 7.3.2 Discrete Cosine Transform

The DCT–2 breaks recursively down into the DCT–2, the DCT–4, the stride permutation matrix, the size two DFT, the identity matrix, and the row-reversed identity matrix. Next to the recursive description of the DCT–2 we also need the recursive description of the DCT–4. The DCT–4 breaks down into the DCT–2, the bi-diagonal matrix ($S_n$), and the diagonal matrix, where the diagonal is represented by the function $d[k] = \frac{1}{\cos\frac{2k+1}{4n}}$.

$$\mathbf{DCT\text{-}2}_n = L^n_m\ (\mathbf{DCT\text{-}2}_m \oplus \mathbf{DCT\text{-}4}_m)\ (F_2 \otimes I_m)\ (I_m \oplus J_m), \qquad n = 2m$$

$$\mathbf{DCT\text{-}4}_n = S_n\ \mathbf{DCT\text{-}2}_n\ \mathrm{diag}\left(\frac{1}{\cos\frac{2k+1}{4n}}\right), \qquad 0 \le k < n$$

```
  -- Discrete Cosine Transform type 2.
dct2s :: Int -> (DVector Float) -> (DVector Float)
dct2s n xn = (dct2n n) ** xn

  -- Discrete Cosine Transform matrix type 2.
```

```
dct2n :: Int -> Matr Float
dct2n 2 = compose (diagMatr (vector [1,1/(Prelude.sqrt 2)])) (f2)
dct2n n = compose
   (compose (strideMatr nf mf) (directSum (dct2n mh) (dct4n mh)))
   (compose (tensor f2 (idMatr mf)) (directSum (idMatr mf) (jdMatr mf)))
      where
        nf = value n
        mf = nf `div` 2
        mh = n `Prelude.div` 2
```

The function `dct2s` represents the transform; it multiplies the transform matrix with the input vector `xn`. The transform matrix is generated by the function `dct2n`. Appendix F.2 contains the implementation of the stride permutation matrix (`strideMatr`) and the size two DCT (`f2`).

```
  -- Discrete Cosine Transform type 4.
dct4s :: Int -> (DVector Float) -> (DVector Float)
dct4s n xn = (dct4n n) ** xn

  -- Discrete Cosine Transform matrix type 4.
dct4n :: Int -> Matr Float
dct4n 2 = compose (jdMatr 2) (rotateMatr)
dct4n n = compose (sDiagn nf) (compose (dct2n n) (diagMatr diagvec))
    where
      nf  = value n
      fnf = intToFloat nf
      fi  = intToFloat i
      diagvec = indexed nf $ \i -> 1/(2 * cos(((2*fi+1)*pi)/(4*(fnf))))
```

The implementation of the type 4 DCT is similar to the previous implementations of transforms. It has a function `dct4s` which represents the transform and it has a compile time recursive function `dct4n` which represents the transform matrix. The implementations of the functions `sDiagn` and `rotateMatr`, respectively representing the bi-diagonal matrix and the rotation matrix, can be found in Appendix F.2.

### 7.3.3 Walsh–Hadamard transform

For the Walsh–Hadamard transform I found several algebraic descriptions [11, 12]:

$$\mathbf{WHT}_n = (\mathbf{WHT}_2 \otimes I_m)\ (I_2 \otimes \mathbf{WHT}_m), \qquad n = 2m \tag{7.1}$$

$$\mathbf{WHT}_{2^n} = \prod_{i=1}^{n} (I_{2^{i-1}} \otimes \mathbf{WHT}_2 \otimes I_{2^{n-i}}) \tag{7.2}$$

$$\mathbf{WHT}_{2^n} = \overbrace{\mathbf{WHT}_2 \otimes \ ... \ \otimes \mathbf{WHT}_2}^{n} \tag{7.3}$$

For my implementation of the WHT in Feldspar I used Equation 7.1 because the structure is the most straight forward when using the new matrix module. Equations 7.2 and 7.3 require matrix operations on more than two matrices while the operations implemented, `compose` and `tensor`, are binary. Below the implementation of the WHT from Equation 7.1.

```
whts :: Int -> DVector Float -> DVector Float
whts n xn = (whtn n) ** xn

whtn :: Int -> Matr Float
whtn 2 = f2
whtn n = compose (tensor (whtn 2) (idMatr mf))
                 (tensor (idMatr 2) (whtn mh))
    where
      mf = (value n) `div` 2
      mh = n `Prelude.div` 2
```

In this implementation the function `whts` represents the transform, this function multiplies the transform matrix with the input vector. The transform matrix is generated by the function `whtn` and consists of the multiplication of two tensor products.

## 7.4 Conclusion

As shown in the previous sections, transforms can be written as recursive matrices and can be implemented in Feldspar using the same structure as mathematical formulas.

The core language code generated from the transforms in Table 1 is currently not as efficient as we would expect: the runtime complexity is $O(n^2)$ instead of $O(n \log n)$. Due to the short amount of time I have been working on this matrix module and these transforms I have not found the exact problem yet. However there are currently several points where the problem might be and which can be improved.

First, The function `mulMatrVec` of the matrix module contains already nineteen different cases (Applendix E), but these can be split up even more or be generalized. Also the implementation of some cases can be improved. Second, The breakdown rules used in the SPL language from Spiral [11] are not implemented. These rules provide directions to break down the individual symbols (Table 2), used in the transform matrices, to even smaller instances. In the current implementations of the transforms inefficient implementations are used for the symbols. For example the implementation of the stride permutation matrix `strideMatrc` (Appendix F.1). As last, Feldspar is not very domain specific yet and some functions are not optimized for the use of vectors and matrices.

The general idea of writing transforms in Feldspar using their high level mathematical structure is promising, as shown in the Spiral project [11] and in this Chapter. However, the current implementation of the matrix module does not give the efficiency of the fast algorithms ($O(n \log n)$). As mentioned above there are several points which could improve this.

# Chapter 8

# Related Work

In this section I will briefly go into other languages and concepts designed for DSP systems.

**Lava** is a programming language to assist circuit designers in specifying, designing, verifying, and implementing hardware [18]. It consists of a collection of Haskell modules. In [18] the authors describe how to implement the fast Fourier transform (FFT) using Lava after which it can be translated to VHDL, which is a hardware description language. They describe how different elements of the FFT algorithm could be implemented separately and in the end compose these basic elements to the FFT. A Lava like approach to implementing the FFT in Feldspar should also be possible, however in this thesis the implementation of the FFT is as close as possible to the Cooley–Tukey algorithm.

**Bjesse** [21] has implemented a DSP library for a radar application in the functional language Haskell during his master thesis. Radar applications usually have high data-rates and much noise which demand much processing power. Bjesse proposes that annotations should be provided to aid the transformation system such that the code could be divided onto distributed architectures. To improve speed of the programs and to make it use less memory, he also uses deforestation. Deforestation is removing or simplifying the intermediate result lists between two functions like fusion in Feldspar. Bjesse does not mention how to translate from Haskell to an imperative language or platform specific system. The code in his thesis uses recursion which makes it harder to apply in Feldspar since Feldspar does not have recursion.

**SequenceL** SequenceL [22] is a Turing-complete, high level, general purpose language with a single data structure, the sequence. SequenceL declares an intended solution to the problem and repeatedly applies a "Normalize-Transpose-Distribute" operation to functions and operators to discover the missing procedural aspects of the solution. These missing aspects could for instance be a conversion between two types. It is an automatic approach for handling calculations without looking at their types.

SequenceL is not a domain specific language like Feldspar. The idea of not bothering about the types of the arguments when you apply operations on them, but let the system figure it out, gives the idea of a higher level of programming. In this thesis a similar approach is taken with the `(**)` function (Chapter 7). This function multiplies two matrices, two vectors, and matrices with vectors. However, Feldspar does not change the arguments of the function to fit the function, like in SequenceL, but for each case a different function is applied to the arguments.

**Spiral** Spiral [11] is a project for automatic generation of DSP code for hardware platforms. SPL the signal processing language from Spiral exploits the mathematical structure of transform algorithms to generate high performance code. In the Spiral project several rules, breakdown and manipulation rules, are used to generate efficient code. Next to these rules it also uses a feedback-optimizer, which uses search and learning techniques, to optimize code for a given platform.

In Spiral the mathematical formulas are represented in the language SPL [20]. SPL is a convenient language for describing matrix operations including composition, direct sum, and tensor product. The SPL optimizer reduces the total number of calculations used in matrix operations compared to the naive way of calculating the same operations. In this thesis I implemented a matrix module which allows efficient implementations of the transforms described in the Spiral project. More about this in Chapter 7.

**Realtime Signal Processing - Dataflow, Visual, and Functional Programming** Reekie [23] provides a framework for programming real-time signal processing systems. The framework has three components: a high-level textual language: Haskell, a visual language: visual Haskell, and the data flow process network model of computation. In chapter 5 (Static Process Networks) Reekie describes data types and functions which capture the key operations for vectors and streams. Using these types and functions he implements several digital signal processing functions. Many functions used are defined recursively which make them less convenient for Feldspar, however some nice ideas for combinators (Chapter 5) are shown.

**Embedded MATLAB** [24] is a subset of the MATLAB language which is a weakly dynamically typed but high-level programming language. Embedded MATLAB generates efficient and readable C code for prototyping and deploying embedded systems, and accelerating of fixed-point algorithms. In this thesis I have used MATLAB to test the correctness of some of the functions implemented.

**The Haskell DSP library** [25] is a collection of Haskell modules containing digital signal processing functions. Many of the digital signal processing functions used in this thesis are also available in the Haskell DSP library. However, both implementations are rather different. Two of those reasons are that Feldspar does not have recursion and in this thesis the focus was not to implement those DSP function, but to provide helper functions to make implementing DSP functions easier. The implementations in the Haskell DSP library helped me to understand the functions and design helper functions.

# Chapter 9

# Conclusion

Currently, much of the code developed for digital signal processing systems is written in a low-level language like C. Writing efficient, platform dependent code in such a language is a time consuming and error prone process. The relatively new programming language Feldspar developed by Ericsson, Chalmers University of Technology, and Eötvös Loránd University, has been introduced as a solution to this problem. Feldspar allows programmers to develop code for digital signal processing systems in a high-level, functional programming language similar to Haskel. Programs written in Feldspar are compiled via an intermediate language, the core language, into platform specific code.

The programming language Feldspar is still in its infancy; its current version offers only basic functionality. This thesis describes the design and implementation of two new additions to Feldspar: combinators (Chapter 5) and the matrix module (Chapter 7). The combinators are designed to make the implementation of filter-like digital signal processing functions straightforward for domain experts. The matrix module allows high level mathematical implementation for transforms.

To implement filters, for digital signal processing, such that the compiler generates efficient code, Chapter 5 proposes several combinators. A combinator is a generic helper function which simplifies the implementation of filter-like functions. From a general perspective, the combinators take a vector and a user defined function as input; and return the output vector. The user defined function can be seen as the body of a loop: it is applied to each element of the input vector. Three types of combinators are distinguished in Chapter 5: a feedforward combinator, a feedback combinator, and a general combinator which provides both, feedforward and feedback. From the discussion in Sections 5.3, 5.4, and 5.5, it follows that the general combinators are the most versatile ones, offering intuitive and efficient implementation possibilities for a wide variety of functions, including Moving

Average, Autoregressive model, and Autoregesive Moving Average.

For future work on filters, it would be interesting to implement the filters as matrix multiplications like the transforms in this thesis. For example the MA filter can be implemented by multiplying a filter matrix with the input vector, where each row of the filter matrix contains a shifted version of the filter coefficients.

Chapter 6 describes how transforms, like the discrete Fourier transform, can be implemented in Feldspar. Several implementations of important transforms, such as the discrete Fourier transform (Section 6.1), the discrete Cosine transform (Section 6.2), and the Walsh-Hadamard transform (Section 6.3), are described. For each transform, respectively a naïve and a fast implementation is provided. One of the challenges encountered during the implementation of transforms in Feldspar, is the lack of support for recursion in Feldspar. The latter is due to the fact that the core language of Feldspar does not support recursion. There are two solutions to circumvent this problem, namely implement the algorithms in a iterative manner and unrolling the recursion at compile time. In this thesis the latter one is used.

The naïve implementations of the transforms are similar to their mathematical representations and allow high level representations in Feldspar. The transforms are implemented as matrix multiplications where the transform is represented by a matrix which is multiplied with the input vector. The fast recursive implementations achieve the same outcomes as their naïve counterparts, but they are significantly more efficient when it comes to the number of computations required to compute the transform. Both solutions have clear disadvantages, the naïve implementations are not very efficient and the fast implementations do not have a very high level structure and generate lots of code. Consequently, a trade-off between respectively code efficiency, generated code size, and readability has to be made when using these algorithms. An alternative solution to these algorithms is shown in Chapter 7.

After we have seen two different implementations of several transforms in Chapter 6 we would like to achieve the best of both, the nice mathematical structure of the naive implementations and the efficient core language code generated by the fast implementations. In the Spiral project [11] the authors have shown that it is possible to write high level recursive transform matrices, in their SPL language, and generate efficient code (using $O(n \log n)$ operations). To make this possible in Feldspar section 7.3 introduces a matrix module which can be used to implement these recursive matrix factorizations. The transforms written using the matrix module have a high level structure and the original mathematical formulas are easily recognizable. However, currently the core language code generated from these transforms is not as efficient as we expected. The core language code generated by Feldspar uses $O(n^2)$ instead of $O(n \log n)$ operations. Section 7.4 discusses

several possible problems in the current version of the matrix module and in the implementations of the transforms. It would be interesting to see if these problems can be solved in the future. In general the implementation of the matrix module can be optimized by using more efficient Feldspar functions and improving the implementation of the matrix-vector-multiplication function. Also the manipulation and breakdown rules described in Spiral [11] can be used to improve the efficiency of the transform specific implementations. During this thesis I was also not able to test the implementations of the DFT and the DCT using QuickCheck. The reason for this is that the results of those functions contain Floats with a certain rounding error. It would be nice to see a generic way to use QuickCheck on these transforms.

In short, both approaches, described in this thesis, raise the level of abstraction and make programming in Feldspar easier for domain experts.

# Acknowledgements

In this Section I would like thank all of those who have generously supported and contributed to the development of this thesis, especially my supervisor Emil and my examiner Mary. They created the opportunity of working on this project and giving me the help and support I needed during my thesis.

Of course I cannot forget Joris. I want to thank you for supporting me, reading my work carefully and giving helpful comments such that I could improve my work.

I would also like to thank the baseband research group from Ericsson for providing a nice working space and answers for all the questions I had about digital signal processing.

The Feldspar project is funded by Ericsson, Vetenskapsrådet, the Swedish Foundation for Strategic Research and the Hungarian National Development Agency.

# Bibliography


[1] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda, "Feldspar: A domain specific language for digital signal processing algorithms," in *Proc. Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE*. IEEE, 2010.

[2] S. K. K. Mitra, *Digital Signal Processing: A Computer-Based Approach*, 2nd ed. McGraw-Hill Higher Education, 2000, pp. 1–9, 41–49.

[3] G. Hutton, *Programming in Haskell*. Cambridge University Press, jan 2007.

[4] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

[5] P. Grant, B. Mulgrew, and J. Thompson, *Digital Signal Processing: Concepts & Applications*. Palgrave Macmillan, 2003, pp. 39–53, 109–112, 126–135, 150–152, 240–252, 267–272.

[6] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*. Macmillan Publishing Company, 1992, ch. 2, pp. 102 – 107.

[7] "Introduction to DSP," May 2010. [Online]. Available: http://www.bores.com/courses/intro/index.htm

[8] D. Liu, A. Nilsson, E. Tell, D. Wu, and J. Eilert, "Bridging dream and reality: programmable baseband processors for software-defined radio," *Comm. Mag.*, vol. 47, no. 9, pp. 134–140, 2009.

[9] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software-defined radios," *International Journal of Wireless Information Networks*, vol. 8, no. 4, pp. 203–215, October 2001.

[10] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. [Online]. Available: http://dx.doi.org/10.2307/2003354

[11] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[12] J. Johnson and M. Püschel, "In search of the optimal walsh-hadamard transform," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, 2000, pp. 3347–3350.

[13] "Users guide to the Feldspar language," May 2010. [Online]. Available: http://feldspar.inf.elte.hu/feldspar/documents/language/FeldsparLanguage.html

[14] "Feldspar," May 2010. [Online]. Available: http://feldspar.inf.elte.hu/

[15] G. Dévai, M. Tejfel, Z. Gera, G. Páli, G. Nagy, Z. Horváth, E. Axelsson, M. Sheeran, A. Vajda, B. Lyckegård, and A. Persson, "Efficient Code Generation from the High-level Domain-specific Language Feldspar for DSPs," in *Proc. ODES-8: 8th Workshop on Optimizations for DSP and Embedded Systems, assoc. with IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[16] "MATLAB," June 2010. [Online]. Available: http://www.mathworks.com/

[17] J. Hughes, "QuickCheck: An automatic testing tool for haskell," June 2010. [Online]. Available: http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

[18] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," in *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 1998, pp. 174–184.

[19] M. Püschel and J. M. F. Moura, "Algebraic signal processing theory," *CoRR*, vol. abs/cs/0612077, 2006.

[20] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," in *Programming Languages Design and Implementation (PLDI)*, 2001, pp. 298–308.

[21] P. Bjesse, "Specification of signal processing programs in a pure functional language and comilation to distributed architectures," Master's thesis, Chalmers University of Technology, 1997.

[22] D. E. Cooke, J. N. Rushton, B. Nemanich, R. G. Watson, and P. Andersen, "Normalize, transpose, and distribute: An automatic approach for handling nonscalars," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 2, pp. 1–49, 2008.

[23] H. J. Reekie, "Realtime signal processing: Dataflow, visual, and functional programming," Ph.D. dissertation, University of Technology at Sydney, 1995. [Online]. Available: http://citeseerx.ist.psu.eu/viewdoc/summary?doi=10.1.1.27.8657

[24] The MathWorks Inc., "Embedded MATLAB™ getting started guide," March 2010. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/toolbox/eml/

[25] "Haskell DSP libraries," April 2010. [Online]. Available: http://haskelldsp.sourceforge.net/

# Appendix A

# Combinators

```
module Combinators where

import qualified Prelude
import Feldspar
import Feldspar.Range


  -- Given the input list xs, the accumulator acc, and the function f compute
  -- for each input value the output value using the function f and the current
  -- accumulator acc.
mapAccum :: (Storable a, Computable acc, Storable b)
    => (acc -> Data a -> (acc, Data b))
    -> acc -> Vector (Data a) -> (acc, Vector (Data b))
mapAccum f init vecA = (final,vecB)
    where
      (vecB,final) = unfoldVec (length vecA) init $ \i acc ->
        let (acc',b) = f acc (vecA!i) in (b,acc')


  -- A loop which takes a function body, and an int end.
  -- Body takes a list of previous calculated y, and the current index and
  -- produces the new y.
  -- loop produces a list of output values of the function body.
feedbackLoop :: (Storable y) =>
    (DVector y -> Data Int -> Data y) -> Data Int -> DVector y
feedbackLoop body leng = unfreezeVector leng ys'
    where
      end = leng - 1
      ys  = array ((mapMonotonic fromInteger $ dataSize leng) :> universal) []
      ys' = for 0 end ys (\i ys'' ->
                          setIx ys'' i (body (unfreezeVector leng ys'') i))
```

```
  -- ffLoop takes a
  --  - function which takes the previous calculated ys (ys is reversed of
  --    which the head is undefined since it is the element which is currently
  --    calculated), the previous xs, reversed with current x as head, and it
  --    produces the new y.
  --  - list of input values: xs,
  -- and it produces a list of ys.
ffLoop :: (Storable y) =>
    (DVector y -> DVector x -> Data y) -> DVector x ->  DVector y
ffLoop body xs = unfreezeVector n ys
    where
      n   = length xs
      ys  = for 0 (length xs - 1) (ys') forBody
      ys' = array ((mapMonotonic fromInteger $ dataSize n) :> universal) []
      forBody i ys'' = setIx ys'' i yNew
        where
          yNew = body (reverse $ take (i+1) $
                       unfreezeVector n ys'') (reverse $ take (i+1) xs)


  -- loop is similar to ffLoop except it has a state like mapAccum.
loop :: (Storable y, Computable a) =>
    (DVector y -> DVector x -> a -> (Data y, a)) -> a -> DVector x -> DVector y
loop body state xs = unfreezeVector n ys
    where
      n = length xs
      ys' = array ((mapMonotonic fromInteger $ dataSize n) :> universal) []
      ys = fst $ for 0 (length xs - 1) (ys',state) forBody
      forBody = \i (ys'',state') ->
        let (yNew,stateNew) = (body (reverse $ take (i+1) $
                     unfreezeVector n ys'') (reverse $ take (i+1) xs) state)
        in (setIx ys'' i yNew, stateNew)


streamState :: (Storable b, Storable c)
    => (DVector a -> DVector b -> (Data c, Data b))
    -> DVector b -> DVector a -> (DVector c, DVector b)
streamState f initBuf inp =
               (unfreezeVector n finOutArr, unfreezeVector m finBuf)
  where
    n  = length inp
    m  = length initBuf
    initOutArr = array ((mapMonotonic fromInteger $
                                          dataSize n) :> universal) []
    (finOutArr,finBuf) = for 0 (n - 1) (initOutArr, freezeVector initBuf) body
      where
        body i (arr,buf) = (arr',buf')
          where
```

```
cycle = permute (\_ j -> (j + (i`mod`m)) `mod` m)
bufV  = cycle $ unfreezeVector m buf
(c,b) = f (reverse $ take (i+1) inp) bufV
arr'  = setIx arr i c
buf'  = setIx buf (i`mod`m) b
```

# Appendix B

# Core Language Code Filters

## B.1 Core Language Code of maA

```
*PrintCodes> printCoreMaA
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1)) = (v0_0_0,v49)
  where
    v3 = v0_0_0 - 1
    v46 = []
    (v47_0,(v47_1_0,v47_1_1)) = while cont body (0,(v46,0))
      where
        cont (v2_0,(v2_1_0,v2_1_1)) = v4
          where
            v4 = v2_0 <= v3
        body (v5_0,(v5_1_0,v5_1_1)) = (v6,(v44,v45))
          where
            v6 = v5_0 + 1
            v12 = v5_1_1 + 1
            v13 = v12 - v0_1_0
            v15 = v13 > 0
            v16 = if v15 then thenBranch () else elseBranch ()
              where
                thenBranch v11 = v13
                elseBranch v14 = 0
            v17 = v0_0_0 - v16
            v19 = v17 > 0
            v20 = if v19 then thenBranch () else elseBranch ()
              where
                thenBranch v10 = v17
                elseBranch v18 = 0
            v23 = v5_1_1 + 1
            v25 = v23 < v0_1_0
            v26 = if v25 then thenBranch () else elseBranch ()
```

```
            where
              thenBranch v22 = v23
              elseBranch v24 = v0_1_0
          v27 = v20 < v26
          v28 = if v27 then thenBranch () else elseBranch ()
            where
              thenBranch v9 = v20
              elseBranch v21 = v26
          v30 = v28 < v0_1_0
          v31 = if v30 then thenBranch () else elseBranch ()
            where
              thenBranch v8 = v28
              elseBranch v29 = v0_1_0
          v32 = v31 - 1
          v37 = v28 - 1
          (v43_0,v43_1) = while cont body (0,0)
            where
              cont (v7_0,v7_1) = v33
                where
                  v33 = v7_0 <= v32
              body (v34_0,v34_1) = (v35,v42)
                where
                  v35 = v34_0 + 1
                  v36 = v0_1_1 ! v34_0
                  v38 = v37 - v34_0
                  v39 = v38 + v16
                  v40 = v0_0_1 ! v39
                  v41 = v36 * v40
                  v42 = v34_1 + v41
          v44 = setIx (v5_1_0,v5_0,v43_1)
          v45 = v5_1_1 + 1
    v49 = parallel v0_0_0 ixf
      where
        ixf v1 = v48
          where
            v48 = v47_1_0 ! v1
```

## B.2 Core Language Code of `maM`

```
*PrintCodes> printCoreMaM
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1)) = (v7,v40)
  where
    v2 = v0_1_0 - 1
    v3 = v2 + v0_0_0
    v4 = v3 + 1
    v6 = v4 < v0_0_0
    v7 = if v6 then thenBranch () else elseBranch ()
      where
```

```
      thenBranch v1 = v4
      elseBranch v5 = v0_0_0
  v40 = parallel v7 ixf
    where
      ixf v8 = v39_1
        where
          v13 = v3 - v8
          v15 = v13 > 0
          v16 = if v15 then thenBranch () else elseBranch ()
            where
              thenBranch v12 = v13
              elseBranch v14 = 0
          v18 = v16 < v0_1_0
          v19 = if v18 then thenBranch () else elseBranch ()
            where
              thenBranch v11 = v16
              elseBranch v17 = v0_1_0
          v21 = v19 < v0_1_0
          v22 = if v21 then thenBranch () else elseBranch ()
            where
              thenBranch v10 = v19
              elseBranch v20 = v0_1_0
          v23 = v22 - 1
          v32 = v19 - 1
          (v39_0,v39_1) = while cont body (0,0)
            where
              cont (v9_0,v9_1) = v24
                where
                  v24 = v9_0 <= v23
              body (v25_0,v25_1) = (v26,v38)
                where
                  v26 = v25_0 + 1
                  v27 = v0_1_1 ! v25_0
                  v33 = v32 - v25_0
                  v34 = v33 + v8
                  v35 = v34 < v2
                  v36 = if v35 then thenBranch v34 else elseBranch v34
                    where
                      thenBranch v28 = 0
                      elseBranch v29 = v31
                        where
                          v30 = v29 - v2
                          v31 = v0_0_1 ! v30
                  v37 = v27 * v36
                  v38 = v25_1 + v37
```

## B.3 Core Language Code of `ma`

```
*PrintCodes> printCoreMa
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1)) = (v0_0_0,v32)
  where
    v3 = v0_0_0 - 1
    v29 = []
    (v30_0,(v30_1_0,v30_1_1)) = while cont body (0,(v29,()))
      where
        cont (v2_0,(v2_1_0,v2_1_1)) = v4
          where
            v4 = v2_0 <= v3
        body (v5_0,(v5_1_0,v5_1_1)) = (v6,(v28,()))
          where
            v6 = v5_0 + 1
            v11 = v5_0 + 1
            v12 = v0_0_0 < v11
            v13 = if v12 then thenBranch () else elseBranch ()
              where
                thenBranch v9 = v0_0_0
                elseBranch v10 = v11
            v15 = v13 < v0_1_0
            v16 = if v15 then thenBranch () else elseBranch ()
              where
                thenBranch v8 = v13
                elseBranch v14 = v0_1_0
            v17 = v16 - 1
            v22 = v13 - 1
            (v27_0,v27_1) = while cont body (0,0)
              where
                cont (v7_0,v7_1) = v18
                  where
                    v18 = v7_0 <= v17
                body (v19_0,v19_1) = (v20,v26)
                  where
                    v20 = v19_0 + 1
                    v21 = v0_1_1 ! v19_0
                    v23 = v22 - v19_0
                    v24 = v0_0_1 ! v23
                    v25 = v21 * v24
                    v26 = v19_1 + v25
            v28 = setIx (v5_1_0,v5_0,v27_1)
    v32 = parallel v0_0_0 ixf
      where
        ixf v1 = v31
          where
            v31 = v30_1_0 ! v1
```

## B.4 Core Language Code of `arFeedLoop`

```
*PrintCodes> printCoreArFeedLoop
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1)) = (v0_0_0,v33)
  where
    v3 = v0_0_0 - 1
    v30 = []
    (v31_0,v31_1) = while cont body (0,v30)
      where
        cont (v2_0,v2_1) = v4
          where
            v4 = v2_0 <= v3
        body (v5_0,v5_1) = (v6,v29)
          where
            v6 = v5_0 + 1
            v7 = v0_0_1 ! v5_0
            v12 = v0_0_0 < v5_0
            v13 = if v12 then thenBranch () else elseBranch ()
              where
                thenBranch v10 = v0_0_0
                elseBranch v11 = v5_0
            v15 = v13 < v0_1_0
            v16 = if v15 then thenBranch () else elseBranch ()
              where
                thenBranch v9 = v13
                elseBranch v14 = v0_1_0
            v17 = v16 - 1
            v22 = v13 - 1
            (v27_0,v27_1) = while cont body (0,0)
              where
                cont (v8_0,v8_1) = v18
                  where
                    v18 = v8_0 <= v17
                body (v19_0,v19_1) = (v20,v26)
                  where
                    v20 = v19_0 + 1
                    v21 = v0_1_1 ! v19_0
                    v23 = v22 - v19_0
                    v24 = v5_1 ! v23
                    v25 = v21 * v24
                    v26 = v19_1 + v25
            v28 = v7 - v27_1
            v29 = setIx (v5_1,v5_0,v28)
    v33 = parallel v0_0_0 ixf
      where
        ixf v1 = v32
          where
            v32 = v31_1 ! v1
```

## B.5 Core Language Code of `arLoop`

```
*PrintCodes> printCoreArLoop
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1)) = (v0_0_0,v47)
  where
    v3 = v0_0_0 - 1
    v44 = []
    (v45_0,(v45_1_0,v45_1_1)) = while cont body (0,(v44,()))
      where
        cont (v2_0,(v2_1_0,v2_1_1)) = v4
          where
            v4 = v2_0 <= v3
        body (v5_0,(v5_1_0,v5_1_1)) = (v6,(v43,()))
          where
            v6 = v5_0 + 1
            v9 = v5_0 + 1
            v10 = v0_0_0 < v9
            v11 = if v10 then thenBranch () else elseBranch ()
              where
                thenBranch v7 = v0_0_0
                elseBranch v8 = v9
            v12 = v11 - 1
            v13 = v12 - 0
            v14 = v0_0_1 ! v13
            v20 = v5_0 + 1
            v21 = v0_0_0 < v20
            v22 = if v21 then thenBranch () else elseBranch ()
              where
                thenBranch v18 = v0_0_0
                elseBranch v19 = v20
            v23 = v22 - 1
            v25 = v23 > 0
            v26 = if v25 then thenBranch () else elseBranch ()
              where
                thenBranch v17 = v23
                elseBranch v24 = 0
            v28 = v26 < v0_1_0
            v29 = if v28 then thenBranch () else elseBranch ()
              where
                thenBranch v16 = v26
                elseBranch v27 = v0_1_0
            v30 = v29 - 1
            v35 = v22 - 1
            (v41_0,v41_1) = while cont body (0,0)
              where
                cont (v15_0,v15_1) = v31
                  where
                    v31 = v15_0 <= v30
```

```
                body (v32_0,v32_1) = (v33,v40)
                  where
                    v33 = v32_0 + 1
                    v34 = v0_1_1 ! v32_0
                    v36 = v32_0 + 1
                    v37 = v35 - v36
                    v38 = v5_1_0 ! v37
                    v39 = v34 * v38
                    v40 = v32_1 + v39
            v42 = v14 - v41_1
            v43 = setIx (v5_1_0,v5_0,v42)
    v47 = parallel v0_0_0 ixf
      where
        ixf v1 = v46
          where
            v46 = v45_1_0 ! v1
```

## B.6 Core Language Code of `arma`

```
*PrintCodes> printCoreArmaLoop
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1),(v0_2_0,v0_2_1)) = (v0_0_0,v60)
  where
    v3 = v0_0_0 - 1
    v57 = []
    (v58_0,(v58_1_0,v58_1_1)) = while cont body (0,(v57,()))
      where
        cont (v2_0,(v2_1_0,v2_1_1)) = v4
          where
            v4 = v2_0 <= v3
        body (v5_0,(v5_1_0,v5_1_1)) = (v6,(v56,()))
          where
            v6 = v5_0 + 1
            v11 = v5_0 + 1
            v12 = v0_0_0 < v11
            v13 = if v12 then thenBranch () else elseBranch ()
              where
                thenBranch v9 = v0_0_0
                elseBranch v10 = v11
            v15 = v13 < v0_1_0
            v16 = if v15 then thenBranch () else elseBranch ()
              where
                thenBranch v8 = v13
                elseBranch v14 = v0_1_0
            v17 = v16 - 1
            v22 = v13 - 1
            (v27_0,v27_1) = while cont body (0,0)
              where
                cont (v7_0,v7_1) = v18
```

```
              where
                v18 = v7_0 <= v17
            body (v19_0,v19_1) = (v20,v26)
              where
                v20 = v19_0 + 1
                v21 = v0_1_1 ! v19_0
                v23 = v22 - v19_0
                v24 = v0_0_1 ! v23
                v25 = v21 * v24
                v26 = v19_1 + v25
          v33 = v5_0 + 1
          v34 = v0_0_0 < v33
          v35 = if v34 then thenBranch () else elseBranch ()
            where
              thenBranch v31 = v0_0_0
              elseBranch v32 = v33
          v36 = v35 - 1
          v38 = v36 > 0
          v39 = if v38 then thenBranch () else elseBranch ()
            where
              thenBranch v30 = v36
              elseBranch v37 = 0
          v41 = v39 < v0_2_0
          v42 = if v41 then thenBranch () else elseBranch ()
            where
              thenBranch v29 = v39
              elseBranch v40 = v0_2_0
          v43 = v42 - 1
          v48 = v35 - 1
          (v54_0,v54_1) = while cont body (0,0)
            where
              cont (v28_0,v28_1) = v44
                where
                  v44 = v28_0 <= v43
              body (v45_0,v45_1) = (v46,v53)
                where
                  v46 = v45_0 + 1
                  v47 = v0_2_1 ! v45_0
                  v49 = v45_0 + 1
                  v50 = v48 - v49
                  v51 = v5_1_0 ! v50
                  v52 = v47 * v51
                  v53 = v45_1 + v52
          v55 = v27_1 - v54_1
          v56 = setIx (v5_1_0,v5_0,v55)
  v60 = parallel v0_0_0 ixf
    where
      ixf v1 = v59
        where
```

```
          v59 = v58_1_0 ! v1
```

## B.7 Core Language Code of armaStream

```
*PrintCodes> printCoreArmaStream
program ((v0_0_0,v0_0_1),(v0_1_0,v0_1_1),(v0_2_0,v0_2_1)) = (v0_0_0,v65)
  where
    v3 = v0_0_0 - 1
    v29 = v0_2_0 - 1
    v35 = v0_2_0 - 1
    v44 = v0_2_0 < 0
    v47 = v0_2_0 > 0
    v60 = []
    v62 = parallel v0_2_0 ixf
      where
        ixf v61 = 0
    (v63_0,(v63_1_0,v63_1_1)) = while cont body (0,(v60,v62))
      where
        cont (v2_0,(v2_1_0,v2_1_1)) = v4
          where
            v4 = v2_0 <= v3
        body (v5_0,(v5_1_0,v5_1_1)) = (v6,(v57,v59))
          where
            v6 = v5_0 + 1
            v11 = v5_0 + 1
            v12 = v0_0_0 < v11
            v13 = if v12 then thenBranch () else elseBranch ()
              where
                thenBranch v9 = v0_0_0
                elseBranch v10 = v11
            v15 = v13 < v0_1_0
            v16 = if v15 then thenBranch () else elseBranch ()
              where
                thenBranch v8 = v13
                elseBranch v14 = v0_1_0
            v17 = v16 - 1
            v22 = v13 - 1
            (v27_0,v27_1) = while cont body (0,0)
              where
                cont (v7_0,v7_1) = v18
                  where
                    v18 = v7_0 <= v17
                body (v19_0,v19_1) = (v20,v26)
                  where
                    v20 = v19_0 + 1
                    v21 = v0_1_1 ! v19_0
                    v23 = v22 - v19_0
                    v24 = v0_0_1 ! v23
```

```
                v25 = v21 * v24
                v26 = v19_1 + v25
          v37 = rem (v5_0,v0_2_0)
          (v55_0,v55_1) = while cont body (0,0)
            where
              cont (v28_0,v28_1) = v30
                where
                  v30 = v28_0 <= v29
              body (v31_0,v31_1) = (v32,v54)
                where
                  v32 = v31_0 + 1
                  v33 = v0_2_1 ! v31_0
                  v36 = v35 - v31_0
                  v38 = v36 + v37
                  v39 = rem (v38,v0_2_0)
                  v40 = v39 + v0_2_0
                  v42 = v39 /= 0
                  v43 = v38 > 0
                  v45 = v43 && v44
                  v46 = v38 < 0
                  v48 = v46 && v47
                  v49 = v45 || v48
                  v50 = v42 && v49
                  v51 = if v50 then thenBranch () else elseBranch ()
                    where
                      thenBranch v34 = v40
                      elseBranch v41 = v39
                  v52 = v5_1_1 ! v51
                  v53 = v33 * v52
                  v54 = v31_1 + v53
          v56 = v27_1 - v55_1
          v57 = setIx (v5_1_0,v5_0,v56)
          v58 = rem (v5_0,v0_2_0)
          v59 = setIx (v5_1_1,v58,v56)
  v65 = parallel v0_0_0 ixf
    where
      ixf v1 = v64
        where
          v64 = v63_1_0 ! v1
```

# Appendix C

# QuickCheck

Below are the QuickCheck properties used to check the equality of the different implementations of the filters.

```
module TestFilters where

import qualified Prelude as P
import Feldspar
import Test.QuickCheck hiding (vector)

import Filters


  -- Compare the implementations of MA
  -- maLoop, maA, and maM
maQC :: [Int] -> [Int] -> Property
maQC xs bs = (P.length xs) P.> (P.length bs) ==>
               (maL' P.== maA') P.&& (maL' P.== maM')
    where
      maL' = eval $ maLoop (vector xs) (vector bs)
      maA' = eval $ maA (vector xs) (vector bs)
      maM' = eval $ maM (vector xs) (vector bs)

  -- Compare the implementations of AR
  -- arLoop, arFeedLoop, and arMapAcc
arQC :: [Int] -> [Int] -> Property
arQC xs as = (P.length xs) P.> (P.length as) ==>
               (arL' P.== arA') P.&& (arL' P.== arF')
    where
      arA' = eval $ arMapAcc (vector xs) (vector as)
      arF' = eval $ arFeedLoop (vector xs) (vector as)
      arL' = eval $ arLoop (vector xs) (vector as)

  -- Compare the implementations of ARMA
```

```
  -- armaLoop and armaStream
armaQC :: [Int] -> [Int] -> [Int] -> Property
armaQC xs bs as = ((P.length xs) P.> (P.length as)) P.&&
                    ((P.length xs) P.> (P.length bs)) ==>
                      (armaL' P.== armaS')
    where
      armaL' = eval $ armaLoop (vector xs) (vector bs) (vector as)
      armaS' = eval $ armaStream (vector xs) (vector bs) (vector as)
```

# Appendix D

# Core Language Code Transforms

## D.1 Core Language Code of `dft`

```
program (v0_0,v0_1) = (v3,v40)
  where
    v1 = v0_0 - 1
    v2 = v1 - 0
    v3 = v2 + 1
    v7 = v0_0 - 1
    v8 = v7 - 0
    v9 = v8 + 1
    v11 = v9 < v0_0
    v12 = if v11 then thenBranch () else elseBranch ()
      where
        thenBranch v6 = v9
        elseBranch v10 = v0_0
    v13 = v12 - 1
    v18 = floatToReal 0.0
    v19 = floatToReal 2.0
    v20 = floatToReal 3.1415927
    v21 = mul_RealNum (v19,v20)
    v30 = intToFloat v0_0
    v31 = floatToReal v30
    v36 = floatToReal 0.0
    v37 = floatToReal 0.0
    v38 = mkComplexReal (v36,v37)
    v40 = parallel v3 ixf
      where
        ixf v4 = v39_1
          where
            v22 = v4 + 0
            v23 = intToFloat v22
```

```
            v24 = floatToReal v23
            v25 = mul_RealNum (v21,v24)
            (v39_0,v39_1) = while cont body (0,v38)
              where
                cont (v5_0,v5_1) = v14
                  where
                    v14 = v5_0 <= v13
                body (v15_0,v15_1) = (v16,v35)
                  where
                    v16 = v15_0 + 1
                    v17 = v0_1 ! v15_0
                    v26 = v15_0 + 0
                    v27 = intToFloat v26
                    v28 = floatToReal v27
                    v29 = mul_RealNum (v25,v28)
                    v32 = div_RealNum (v29,v31)
                    v33 = sub_RealNum (v18,v32)
                    v34 = mkPolar_ComplexReal (v17,v33)
                    v35 = add_ComplexReal (v15_1,v34)
```

## D.2 Core Language Code of `dftm`

```
program (v0_0,v0_1) = (v0_0,v28)
  where
    v3 = v0_0 - 1
    v8 = floatToReal 1.0
    v9 = floatToReal 0.0
    v10 = floatToReal 2.0
    v11 = floatToReal 3.1415927
    v12 = mul_RealNum (v10,v11)
    v17 = intToFloat v0_0
    v18 = floatToReal v17
    v24 = floatToReal 0.0
    v25 = floatToReal 0.0
    v26 = mkComplexReal (v24,v25)
    v28 = parallel v0_0 ixf
      where
        ixf v1 = v27_1
          where
            (v27_0,v27_1) = while cont body (0,v26)
              where
                cont (v2_0,v2_1) = v4
                  where
                    v4 = v2_0 <= v3
                body (v5_0,v5_1) = (v6,v23)
                  where
                    v6 = v5_0 + 1
                    v7 = v0_1 ! v5_0
```

```
      v13 = v1 * v5_0
      v14 = intToFloat v13
      v15 = floatToReal v14
      v16 = mul_RealNum (v12,v15)
      v19 = div_RealNum (v16,v18)
      v20 = sub_RealNum (v9,v19)
      v21 = mkPolar_ComplexReal (v8,v20)
      v22 = mul_ComplexReal (v7,v21)
      v23 = add_ComplexReal (v5_1,v22)
```

## D.3 Core Language Code of the function `fft`

Below is the core language code of the function `fft` for an input vector of size 8. The Feldspar code contained compile time recursion which is unrolled here.

```
program (v0_0,v0_1) = (8,v252)
  where
    v8 = v0_1 ! 0
    v9 = floatToReal 0.0
    v10 = mkComplexReal (v8,v9)
    v11 = v0_1 ! 4
    v12 = mkComplexReal (v11,v9)
    v13 = floatToReal 1.0
    v14 = floatToReal 0.0
    v15 = floatToReal 2.0
    v16 = floatToReal 3.1415927
    v17 = mul_RealNum (v15,v16)
    v22 = floatToReal 2.0
    v29 = floatToReal 3.1415927
    v30 = mul_RealNum (v15,v29)
    v36 = floatToReal 2.0
    v44 = parallel 2 ixf
      where
        ixf v6 = v43
          where
            v42 = v6 < 1
            v43 = if v42 then thenBranch v6 else elseBranch v6
              where
                thenBranch v7 = v27
                  where
                    v18 = v7 + 0
                    v19 = intToFloat v18
                    v20 = floatToReal v19
                    v21 = mul_RealNum (v17,v20)
                    v23 = div_RealNum (v21,v22)
                    v24 = sub_RealNum (v14,v23)
                    v25 = mkPolar_ComplexReal (v13,v24)
                    v26 = mul_ComplexReal (v12,v25)
                    v27 = add_ComplexReal (v10,v26)
                elseBranch v28 = v41
                  where
                    v31 = v28 - 1
                    v32 = v31 + 0
                    v33 = intToFloat v32
                    v34 = floatToReal v33
                    v35 = mul_RealNum (v30,v34)
                    v37 = div_RealNum (v35,v36)
```

```
                v38 = sub_RealNum (v14,v37)
                v39 = mkPolar_ComplexReal (v13,v38)
                v40 = mul_ComplexReal (v12,v39)
                v41 = sub_ComplexReal (v10,v40)
v48 = v0_1 ! 2
v49 = mkComplexReal (v48,v9)
v50 = v0_1 ! 6
v51 = mkComplexReal (v50,v9)
v52 = floatToReal 3.1415927
v53 = mul_RealNum (v15,v52)
v58 = floatToReal 2.0
v65 = floatToReal 3.1415927
v66 = mul_RealNum (v15,v65)
v72 = floatToReal 2.0
v80 = parallel 2 ixf
  where
    ixf v46 = v79
      where
        v78 = v46 < 1
        v79 = if v78 then thenBranch v46 else elseBranch v46
          where
            thenBranch v47 = v63
              where
                v54 = v47 + 0
                v55 = intToFloat v54
                v56 = floatToReal v55
                v57 = mul_RealNum (v53,v56)
                v59 = div_RealNum (v57,v58)
                v60 = sub_RealNum (v14,v59)
                v61 = mkPolar_ComplexReal (v13,v60)
                v62 = mul_ComplexReal (v51,v61)
                v63 = add_ComplexReal (v49,v62)
            elseBranch v64 = v77
              where
                v67 = v64 - 1
                v68 = v67 + 0
                v69 = intToFloat v68
                v70 = floatToReal v69
                v71 = mul_RealNum (v66,v70)
                v73 = div_RealNum (v71,v72)
                v74 = sub_RealNum (v14,v73)
                v75 = mkPolar_ComplexReal (v13,v74)
                v76 = mul_ComplexReal (v51,v75)
                v77 = sub_ComplexReal (v49,v76)
v82 = floatToReal 3.1415927
v83 = mul_RealNum (v15,v82)
v88 = floatToReal 4.0
v98 = floatToReal 3.1415927
v99 = mul_RealNum (v15,v98)
v104 = floatToReal 4.0
v112 = parallel 4 ixf
  where
    ixf v4 = v111
      where
        v110 = v4 < 2
        v111 = if v110 then thenBranch v4 else elseBranch v4
          where
            thenBranch v5 = v93
              where
                v45 = v44 ! v5
                v81 = v80 ! v5
                v84 = v5 + 0
```

```
                v85 = intToFloat v84
                v86 = floatToReal v85
                v87 = mul_RealNum (v83,v86)
                v89 = div_RealNum (v87,v88)
                v90 = sub_RealNum (v14,v89)
                v91 = mkPolar_ComplexReal (v13,v90)
                v92 = mul_ComplexReal (v81,v91)
                v93 = add_ComplexReal (v45,v92)
            elseBranch v94 = v109
              where
                v95 = v94 - 2
                v96 = v44 ! v95
                v97 = v80 ! v95
                v100 = v95 + 0
                v101 = intToFloat v100
                v102 = floatToReal v101
                v103 = mul_RealNum (v99,v102)
                v105 = div_RealNum (v103,v104)
                v106 = sub_RealNum (v14,v105)
                v107 = mkPolar_ComplexReal (v13,v106)
                v108 = mul_ComplexReal (v97,v107)
                v109 = sub_ComplexReal (v96,v108)
v118 = v0_1 ! 1
v119 = mkComplexReal (v118,v9)
v120 = v0_1 ! 5
v121 = mkComplexReal (v120,v9)
v122 = floatToReal 3.1415927
v123 = mul_RealNum (v15,v122)
v128 = floatToReal 2.0
v135 = floatToReal 3.1415927
v136 = mul_RealNum (v15,v135)
v142 = floatToReal 2.0
v150 = parallel 2 ixf
  where
    ixf v116 = v149
      where
        v148 = v116 < 1
        v149 = if v148 then thenBranch v116 else elseBranch v116
          where
            thenBranch v117 = v133
              where
                v124 = v117 + 0
                v125 = intToFloat v124
                v126 = floatToReal v125
                v127 = mul_RealNum (v123,v126)
                v129 = div_RealNum (v127,v128)
                v130 = sub_RealNum (v14,v129)
                v131 = mkPolar_ComplexReal (v13,v130)
                v132 = mul_ComplexReal (v121,v131)
                v133 = add_ComplexReal (v119,v132)
            elseBranch v134 = v147
              where
                v137 = v134 - 1
                v138 = v137 + 0
                v139 = intToFloat v138
                v140 = floatToReal v139
                v141 = mul_RealNum (v136,v140)
                v143 = div_RealNum (v141,v142)
                v144 = sub_RealNum (v14,v143)
                v145 = mkPolar_ComplexReal (v13,v144)
                v146 = mul_ComplexReal (v121,v145)
                v147 = sub_ComplexReal (v119,v146)
```

```
v154 = v0_1 ! 3
v155 = mkComplexReal (v154,v9)
v156 = v0_1 ! 7
v157 = mkComplexReal (v156,v9)
v158 = floatToReal 3.1415927
v159 = mul_RealNum (v15,v158)
v164 = floatToReal 2.0
v171 = floatToReal 3.1415927
v172 = mul_RealNum (v15,v171)
v178 = floatToReal 2.0
v186 = parallel 2 ixf
  where
    ixf v152 = v185
      where
        v184 = v152 < 1
        v185 = if v184 then thenBranch v152 else elseBranch v152
          where
            thenBranch v153 = v169
              where
                v160 = v153 + 0
                v161 = intToFloat v160
                v162 = floatToReal v161
                v163 = mul_RealNum (v159,v162)
                v165 = div_RealNum (v163,v164)
                v166 = sub_RealNum (v14,v165)
                v167 = mkPolar_ComplexReal (v13,v166)
                v168 = mul_ComplexReal (v157,v167)
                v169 = add_ComplexReal (v155,v168)
            elseBranch v170 = v183
              where
                v173 = v170 - 1
                v174 = v173 + 0
                v175 = intToFloat v174
                v176 = floatToReal v175
                v177 = mul_RealNum (v172,v176)
                v179 = div_RealNum (v177,v178)
                v180 = sub_RealNum (v14,v179)
                v181 = mkPolar_ComplexReal (v13,v180)
                v182 = mul_ComplexReal (v157,v181)
                v183 = sub_ComplexReal (v155,v182)
v188 = floatToReal 3.1415927
v189 = mul_RealNum (v15,v188)
v194 = floatToReal 4.0
v204 = floatToReal 3.1415927
v205 = mul_RealNum (v15,v204)
v210 = floatToReal 4.0
v218 = parallel 4 ixf
  where
    ixf v114 = v217
      where
        v216 = v114 < 2
        v217 = if v216 then thenBranch v114 else elseBranch v114
          where
            thenBranch v115 = v199
              where
                v151 = v150 ! v115
                v187 = v186 ! v115
                v190 = v115 + 0
                v191 = intToFloat v190
                v192 = floatToReal v191
                v193 = mul_RealNum (v189,v192)
                v195 = div_RealNum (v193,v194)
```

```
                v196 = sub_RealNum (v14,v195)
                v197 = mkPolar_ComplexReal (v13,v196)
                v198 = mul_ComplexReal (v187,v197)
                v199 = add_ComplexReal (v151,v198)
            elseBranch v200 = v215
              where
                v201 = v200 - 2
                v202 = v150 ! v201
                v203 = v186 ! v201
                v206 = v201 + 0
                v207 = intToFloat v206
                v208 = floatToReal v207
                v209 = mul_RealNum (v205,v208)
                v211 = div_RealNum (v209,v210)
                v212 = sub_RealNum (v14,v211)
                v213 = mkPolar_ComplexReal (v13,v212)
                v214 = mul_ComplexReal (v203,v213)
                v215 = sub_ComplexReal (v202,v214)
v220 = floatToReal 3.1415927
v221 = mul_RealNum (v15,v220)
v226 = floatToReal 8.0
v236 = floatToReal 3.1415927
v237 = mul_RealNum (v15,v236)
v242 = floatToReal 8.0
v250 = parallel 8 ixf
  where
    ixf v2 = v249
      where
        v248 = v2 < 4
        v249 = if v248 then thenBranch v2 else elseBranch v2
          where
            thenBranch v3 = v231
              where
                v113 = v112 ! v3
                v219 = v218 ! v3
                v222 = v3 + 0
                v223 = intToFloat v222
                v224 = floatToReal v223
                v225 = mul_RealNum (v221,v224)
                v227 = div_RealNum (v225,v226)
                v228 = sub_RealNum (v14,v227)
                v229 = mkPolar_ComplexReal (v13,v228)
                v230 = mul_ComplexReal (v219,v229)
                v231 = add_ComplexReal (v113,v230)
            elseBranch v232 = v247
              where
                v233 = v232 - 4
                v234 = v112 ! v233
                v235 = v218 ! v233
                v238 = v233 + 0
                v239 = intToFloat v238
                v240 = floatToReal v239
                v241 = mul_RealNum (v237,v240)
                v243 = div_RealNum (v241,v242)
                v244 = sub_RealNum (v14,v243)
                v245 = mkPolar_ComplexReal (v13,v244)
                v246 = mul_ComplexReal (v235,v245)
                v247 = sub_ComplexReal (v234,v246)
v252 = parallel 8 ixf
  where
    ixf v1 = v251
      where
```

```
v251 = v250 ! v1
```

# Appendix E

# Implementation of the Matr Module

```
module Matr where

import qualified Prelude
import Feldspar hiding ((**))
import MissingFunc

{--
Matr is an efficient matrix module inspired by the Spiral project.

Note the core language code generated using this module does not provide the
efficiency which it should.
- implementation of mulMatrVec could be improved
- implementation of Transform specific symbols (Like stride permutation)
  should be improved

Karin Keijzer 2010
--}

-------------------------------------------------------------------------------

data Matr a =
    IndexedMatr (Data Length) (Data Length)
                (Data Ix -> Data Ix -> Data a) -- height width indexfunction
  | Ident     (Data Length) (Data Length) -- height width
  | Jdent     (Data Length) (Data Length) -- height width
  | Diag      (Data Length) (Data Length) (DVector a) -- height width diag
  | Tensor    (Data Length) (Data Length) (Matr a) (Matr a) -- height width l r
  | DirectSum (Data Length) (Data Length) (Matr a) (Matr a) -- height width l r
  | Compose   (Data Length) (Data Length) (Matr a) (Matr a) -- height width l r
```

```
-------------------------------------------------------------------------------

-- Creates a Matr out of a Matrix
matrixToMatr :: (Numeric a) => Matrix a -> Matr a
matrixToMatr xss = IndexedMatr (length xss) (length $ head xss) ixf
  where
    ixf k l = xss ! k ! l

-- Create a matrix given a function and the height and width
indexedMatr :: (Numeric a) =>
    Data Int -> Data Int -> (Data Ix -> Data Ix -> Data a) -> Matr a
indexedMatr h w idx = IndexedMatr h w idx

-- Create a Identity matrix of size n*n
idMatr :: (Numeric a) => Data Int -> Matr a
idMatr n = Ident n n

-- Create a Identity matrix of size n*n
jdMatr :: (Numeric a) => Data Int -> Matr a
jdMatr n = Jdent n n

-- Create a diagonal matrix of size (length xs)*(length xs).
-- The diagonal contains the values of xs.
diagMatr :: (Numeric a) => DVector a -> Matr a
diagMatr xs = Diag (length xs) (length xs) xs

-- Create a Tensor matrix of the matrices a and b
tensor :: (Numeric a) => Matr a -> Matr a -> Matr a
tensor a b = Tensor k l a b
    where
      k = (height a) * (height b)
      l = (width a) * (width b)

-- Create a DirectSum matrix of the matrices a and b
directSum :: (Numeric a) => Matr a -> Matr a -> Matr a
directSum a b = DirectSum k l a b
    where
      k = (height a) + (height b)
      l = (width a) + (width b)

-- Create a matrix by multiplying two matrices
compose :: (Numeric a) => Matr a -> Matr a -> Matr a
compose a b = Compose (height a) (width b) a b

-------------------------------------------------------------------------------

-- Gives the height of a matrix
height :: Matr a -> Data Int
height (IndexedMatr h w ix) = h
```

```
height (Ident h w) = h
height (Jdent h w) = h
height (Diag h w v) = h
height (Tensor h w a b) = h
height (DirectSum h w a b) = h
height (Compose h w a b) = h

-- Gives the width of a matrix
width :: Matr a -> Data Int
width (IndexedMatr h w ix) = w
width (Ident h w) = w
width (Jdent h w) = w
width (Diag h w v) = w
width (Tensor h w a b) = w
width (DirectSum h w a b) = w
width (Compose h w a b) = w

-- chops a vector in pieces of length x
chop :: Data Int -> DVector a -> Vector (DVector a)
chop x vec = indexed (div (length vec) x) $ \k ->
                indexed x $ \l ->
                  vec ! (k*x + l)

-- scale multiplies a vector with a constant
scale :: (Numeric a) => DVector a -> Data a -> DVector a
scale vec c = map (* c) vec
```

---

```
-- Transposes a Matrix
transposeMatr :: (Numeric a) => Matr a -> Matr a
transposeMatr (IndexedMatr h w ix) = IndexedMatr w h (\k l -> ix l k)
transposeMatr (Ident h w)         = Ident w h
transposeMatr (Jdent h w)         = Jdent w h
transposeMatr (Diag h w v)        = Diag w h v
transposeMatr (Tensor h w a b)    = Tensor w h (transposeMatr a)
                                               (transposeMatr b)
transposeMatr (DirectSum h w a b) = DirectSum w h (transposeMatr a)
                                               (transposeMatr b)
transposeMatr (Compose h w a b)   = Compose w h (transposeMatr b)
                                               (transposeMatr a)
```

---

```
-- Matr ** Matr
instance (Numeric a) => Mul (Matr a) (Matr a)
  where
    type Prod (Matr a) (Matr a) = (Matr a)
    (**) = mulMatrMatr
```

```
-- Matr ** Vector
instance (Numeric a) => Mul (Matr a) (DVector a)
  where
    type Prod (Matr a) (DVector a) = (DVector a)
    (**) = mulMatrVec

-- Vector ** Matr
instance (Numeric a) => Mul (DVector a) (Matr a)
  where
    type Prod (DVector a) (Matr a) = (DVector a)
    (**) = mulVecMatr

--------------------------------------------------------------------------------

mulMatrVec :: (Numeric a) => Matr a -> DVector a -> DVector a

  -- Multiply each row of the matrix with the input vector.
mulMatrVec (IndexedMatr h w ix) vec =
    memorize $ map (\k -> scalarProd vec (indexed w (ix k))) (0...(h-1))

  -- Input vector remains unchanged.
mulMatrVec (Ident h w) vec = vec

  -- Input vector is reversed.
mulMatrVec (Jdent h w) vec = reverse vec

  -- Element-wise multiplication of diagonal vector and the input vector.
mulMatrVec (Diag h w v) vec = memorize $ zipWith (*) v vec

  -- First multiply matrix B with the input vector and after that multiply the
  -- result with matrix A.
  -- (AB)x = A(Bx)
mulMatrVec (Compose h w a b) vec =
    memorize $ mulMatrVec a (mulMatrVec b vec)

  -- Multiply matrix a with the first part of the input vector and multiply
  -- matrix b with the second part of the vector after that concatenate the
  -- result vectors.
  -- [ A   ] (x++y) = (Ax)++(By)
  -- [   B ]
mulMatrVec (DirectSum h w a b) vec = memorize $
    (mulMatrVec a (take wa vec)) ++ (mulMatrVec b (drop wa vec))
      where
        wa = width a

  -- Cut the input vector in pieces and multiply matrix b with each piece and
  -- concatenate the result.
```

-- $I_n \otimes B = \begin{bmatrix} B & & & \\ & B & & \\ & & \ddots & \\ & & & B \end{bmatrix} (x{+}{+}y{+}{+}\ldots{+}{+}z) = (Bx){+}{+}(By){+}{+}\ldots{+}{+}(Bz)$

```
mulMatrVec (Tensor h w (Ident ih iw) b) vec =
    memorize $ flatten $ map (mulMatrVec b) (chop (width b) vec)

  -- Cut the input vector in pieces, reverse the pieces, and multiply matrix b
  -- with each piece and concatenate the result.
mulMatrVec (Tensor h w (Jdent ih iw) b) vec =
    memorize $ flatten $
      map (mulMatrVec b) (reverse $ chop (width b) vec)

  -- Cut the input vector in pieces, for each row of the matrix:
  -- multiply each element of the row with one piece of the input vector and
  -- sum the result such that each row of the matrix gives one output value.
mulMatrVec (Tensor h w (IndexedMatr ah aw ix) (Ident ih iw)) vec =
    memorize $ flatten $ map (\k-> map sum (transpose $
           zipWith scale (chop iw vec) (indexed w (ix k)))) (0...(ah-1))

  -- Cut the input vector in pieces and scale each piece with an element from
  -- the diagonal vector. After scaling the pieces are concatenated into one
  -- vector.
mulMatrVec (Tensor h w (Diag ah aw v) (Ident ih iw)) vec =
    memorize $ flatten $ zipWith scale (chop iw vec) v

  -- Cut the input vector in two pieces, where the first half has the length
  -- (the width matrix a multiplied with the width of the identity matrix),
  -- after that multiply the first part with the tensor product of matrix a
  -- and the identity matrix; and multiply the second part with the tensor
  -- product of matrix b and the identity matrix.
mulMatrVec (Tensor h w (DirectSum ah aw a b) i@(Ident ih iw)) vec =
    memorize $ (mulMatrVec (tensor a i) (take halfvec vec)) ++
                           (mulMatrVec (tensor b i) (drop halfvec vec))
      where
        halfvec = (width a) * iw

  -- Cut the input vector in pieces then multiply a vector containing the x^th
  -- elements of each piece with row x of the matrix.
  -- After that reorder and flatten the result.
mulMatrVec (Tensor h w t@(Tensor ah aw a b) (Ident ih iw)) vec =
    memorize $ flatten $ transpose $
           map (mulMatrVec t) (transpose (chop iw vec))

  -- Cut the input vector in pieces then multiply a vector containing the x^th
  -- elements of each piece with row x of the matrix.
  -- After  that reorder and flatten the result.
mulMatrVec (Tensor h w c@(Compose ah aw a b) (Ident ih iw)) vec =
```

```
    memorize $ flatten $ transpose $
           map (mulMatrVec c) (transpose (chop iw vec))

  -- Cut the input vector in pieces and reverse each piece, for each row of
  -- the matrix: multiply each element of the row with one piece of the input
  -- vector and sum the result such that each row of the matrix gives one
  -- output value.
mulMatrVec (Tensor h w (IndexedMatr ah aw ix) (Jdent ih iw)) vec =
    memorize $ flatten $ map (\k-> map sum (transpose $ zipWith
     scale (map reverse $ chop iw vec) (indexed w (ix k)))) (0...(ah-1))

  -- Cut the input vector in pieces, reverse the elements of each piece, and
  -- scale each piece with an element from the diagonal vector. After scaling
  -- the pieces are concatenated into one vector.
mulMatrVec (Tensor h w (Diag ah aw v) (Jdent ih iw)) vec =
    memorize $ flatten $ zipWith scale (map reverse $ chop iw vec) v

  -- Cut the input vector in two pieces, where the first half has the length
  -- (the width matrix a multiplied with the width of the row-reversed
  -- identity matrix), after that multiply the first part with the tensor
  -- product of matrix a and the row-reversed identity matrix; and multiply
  -- the second part with the tensor product of matrix b and the identity
  -- matrix.
mulMatrVec (Tensor h w (DirectSum ah aw a b) j@(Jdent ih iw)) vec =
    memorize $ (mulMatrVec (tensor a j) (take halfvec vec)) ++
                       (mulMatrVec (tensor b j) (drop halfvec vec))
      where
        halfvec = (width a) * iw

  -- Cut the input vector in pieces and reverse the order of the elements of
  -- each piece then multiply a vector containing the x^th elements of
  -- each piece with row x of the matrix. After that reorder and flatten the
  -- result.
mulMatrVec (Tensor h w t@(Tensor ah aw a b) (Jdent ih iw)) vec =
    memorize $ flatten $ transpose $ map (mulMatrVec t)
                          (transpose (map reverse $ chop iw vec))

  -- Cut the input vector in pieces and reverse the order of the elements of
  -- each piece then multiply a vector containing the x^th elements of
  -- each piece with row x of the matrix. After that reorder and flatten the
  -- result.
mulMatrVec (Tensor h w c@(Compose ah aw a b) (Jdent ih iw)) vec =
    memorize $ flatten $ transpose $ map (mulMatrVec c)
                          (transpose (map reverse $ chop iw vec))

  -- General tensor matrix multiplied with a vector.
  -- Cut the input vector in pieces; multiply matrix B with each piece
  -- resulting in a vector Y containing the result vectors (Y can thus be seen
  -- as a matrix). After that multiply matrix A with each column of Y. At the
```

```
  -- end concatenate the results.
mulMatrVec (Tensor h w a b) vec = memorize $ flatten $ transpose $
    map (mulMatrVec a) (transpose $ map (mulMatrVec b)
                                          (chop (width b) vec))

-------------------------------------------------------------------------------

-- Vector ** Matrix
mulVecMatr :: (Numeric a) => DVector a -> Matr a -> DVector a
mulVecMatr vec mat = mulMatrVec (transposeMatr mat) vec

-------------------------------------------------------------------------------

-- Matrix ** Matrix
mulMatrMatr :: (Numeric a) => Matr a -> Matr a -> Matr a
mulMatrMatr a b = compose a b

-------------------------------------------------------------------------------

-- Transforms a Matr into a Matrix (nested vectors)
matrToMatrix :: (Numeric a) => Matr a -> Matrix a
matrToMatrix (IndexedMatr h w ix) = indexedMat h w ix
matrToMatrix (Ident h w) = indexedMat h w (\k l -> (k == l) ? (1,0))
matrToMatrix (Jdent h w) = indexedMat h w (\k l -> (h-k-1 == l) ? (1,0))
matrToMatrix (Diag h w v) = indexedMat h w (\k l -> (k == l) ? (v ! k,0))
matrToMatrix (Compose h w a b) = (matrToMatrix a) ** (matrToMatrix b)

matrToMatrix (Tensor h w a b)    = indexedMat h w idxfun
    where
      n = height a
      m = width a
      s = height b
      t = width b
      a' = matrToMatrix a
      b' = matrToMatrix b
      idxfun k l = ((a' ! y1) ! x1) * ((b' ! y2) ! x2)
        where
          x1 = l `div` t
          y1 = k `div` s
          x2 = l `mod` t
          y2 = k `mod` s

matrToMatrix (DirectSum h w a b) = indexedMat h w idxfun
    where
      n = height a
      m = width a
      s = height b
      t = width b
      a' = matrToMatrix a
```

```
      b' = matrToMatrix b
      idxfun k l = (?) ((k < n) && (l < m))  -- inrange of mat a
                       ((a' ! k) ! l,
                        (?) ((k >= n) && (l >= m))  -- inrange of mat b
                                        ((b' ! (k-n)) ! (l-m),
                                         0)) -- not in a not in b
```

---

```
-- Transforms a Matr into a Matr of type IndexedMatr
matrToIxMatr :: (Numeric a) => Matr a -> Matr a
matrToIxMatr (IndexedMatr h w ix) = IndexedMatr h w ix
matrToIxMatr (Ident h w) = IndexedMatr h w (\k l -> (k == l) ? (1,0))
matrToIxMatr (Jdent h w) = IndexedMatr h w (\k l -> (h-k-1 == l) ? (1,0))
matrToIxMatr (Diag h w v) = IndexedMatr h w (\k l -> (k == l) ? (v ! k,0))
matrToIxMatr (Compose h w a b) = IndexedMatr h w (\k l ->
      scalarProd (indexed wa $ idxa k) (indexed hb $ (\x y -> idxb y x) l))
    where
      (IndexedMatr ha wa idxa) = matrToIxMatr a
      (IndexedMatr hb wb idxb) = matrToIxMatr b

matrToIxMatr (Tensor h w a b)    = IndexedMatr h w idxfun
    where
      (IndexedMatr _ _ idxa) = matrToIxMatr a
      (IndexedMatr s t idxb) = matrToIxMatr b
      idxfun k l = (idxa y1 x1) * (idxb y2 x2)
        where
          x1 = l `div` t
          y1 = k `div` s
          x2 = l `mod` t
          y2 = k `mod` s

matrToIxMatr (DirectSum h w a b) = IndexedMatr h w idxfun
    where
      (IndexedMatr n m idxa) = matrToIxMatr a
      (IndexedMatr _ _ idxb) = matrToIxMatr b
      idxfun k l = (?) ((k < n) && (l < m))  -- inrange of mat a
                       (idxa k l,
                        (?) ((k >= n) && (l >= m))  -- inrange of mat b
                                       (idxb (k-n) (l-m),
                                        0)) -- not in a not in b
```

# Appendix F

# Transforms in Feldspar

## F.1 Discrete Fourier Transform

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes I_m)\ T^n_m\ (I_k \otimes \mathbf{DFT}_m)\ L^n_k, \qquad n = km$$
$$\mathbf{DFT}_2 = F_2$$

```
  -- Main DFT function.
dfts :: Int -> (DVector ComplexReal) -> (DVector ComplexReal)
dfts n xn = (dftn n) ** xn

  -- DFT transform matrix
dftn :: Int -> Matr ComplexReal
dftn 2 = f2c
dftn n = compose
            (compose (tensor (dftn kh) (idMatr mf)) (twiddleMatr nf mf))
            (compose (tensor (idMatr kf) (dftn mh)) (strideMatrc nf kf))
    where
      nf = value n
      mh = n `Prelude.div` kh
      mf = nf `div` kf
      kh = 2
      kf = value kh

  -- DFT of size 2 (with complex numbers)
f2c :: Matr ComplexReal
f2c = indexedMatr 2 2 (\k l -> ((k+l) < 2) ? (1 |+| 0,-1 |+| 0))

  -- Stride permutation matrix Lmn
```

```
strideMatrc :: Data Int -> Data Int -> Matr ComplexReal
strideMatrc n m = indexedMatr n n ix
    where
      ix k l = (k==n-1) ?
               ( (k==l) ? (1 |+| 0,0 |+| 0)
               , (((k*m) `mod` (n - 1))==l) ? (1 |+| 0,0 |+| 0))

  -- Twiddle matrix
twiddleMatr :: Data Int -> Data Int -> Matr ComplexReal
twiddleMatr n m =
    diagMatr ((flatten . map
                  (\k -> map (\l -> twiddle n (l*k)) (0...(m-1))))
                        (0...(n `div` m-1)))

  -- Twiddle factor
twiddle :: Data Int -> Data Int -> Data ComplexReal
twiddle n m = mkPolar 1 (-2 * rpi * rm / rn) -- e^{-2πmi/N}
    where
      rpi = floatToReal pi
      rm  = intToReal m
      rn  = intToReal n
```

## F.2 Discrete Cosine Transform

The discrete cosine transform type two.

$$\mathbf{DCT\text{-}2}_n = L^n_m \ (\mathbf{DCT\text{-}2}_m \oplus \mathbf{DCT\text{-}4}_m) \ (F_2 \otimes I_m) \ (I_m \oplus J_m), \qquad n = 2m$$
$$\mathbf{DCT\text{-}2}_2 = \mathrm{diag}(1, 1/\sqrt{2}) \ F_2$$

```
dct2s :: Int -> (DVector Float) -> (DVector Float)
dct2s n xn = (dct2n n) ** xn

-- Discrete Cosine Transform type 2.
dct2n :: Int -> Matr Float
dct2n 2 = compose (diagMatr (vector [1,1/(Prelude.sqrt 2)])) (f2)
dct2n n = compose
   (compose (strideMatr nf mf) (directSum (dct2n mh) (dct4n mh)))
   (compose (tensor f2 (idMatr mf)) (directSum (idMatr mf) (jdMatr mf)))
      where
        nf = value n
        mf = nf `div` 2
        mh = n `Prelude.div` 2
```

```
-- DFT of size 2
f2 :: (Numeric a) => Matr a
f2 = matrixToMatr $ matrix [[1,1],[1,-1]]

-- Stride permutation matrix Lmn
strideMatr :: (Numeric a) => Data Int -> Data Int -> Matr a
strideMatr n m = indexedMatr n n ix
    where
      ix k l = (k==n-1) ?
                 ((k==l) ? (1,0), (((k*m) `mod` (n - 1))==l) ? (1,0))
```

The discrete cosine transform type four.

$$\begin{aligned} \mathbf{DCT\text{-}4}_n &= S_n\ \mathbf{DCT\text{-}2}_n\ \mathrm{diag}\left(\frac{1}{\cos\frac{2k+1}{4n}}\right), \qquad 0 \le k < n \\ \mathbf{DCT\text{-}4}_2 &= J_2\ R_{13\pi/8} \end{aligned}$$

```
dct4s :: Int -> (DVector Float) -> (DVector Float)
dct4s n xn = (dct4n n) ** xn

-- Discrete Cosine Transform type 4.
dct4n :: Int -> Matr Float
dct4n 2 = compose (jdMatr 2) (rotateMatr)
dct4n n = compose (sDiagn nf) (compose (dct2n n) (diagMatr diagvec))
    where
      nf  = value n
      fnf = intToFloat nf
      fi  = intToFloat i
      diagvec = indexed nf $ \i -> 1/(2 * cos(((2*fi+1)*pi)/(4*(fnf))))

-- Bi-diagonal matrix Sn
sDiagn :: (Numeric a) => Data Int -> Matr a
sDiagn n = indexedMatr n n (\k l -> ((k == l) || ((k+1) == l)) ? (1,0))

-- Rotation Matrix
rotateMatr :: Matr Float
rotateMatr =
    matrixToMatr $
      (replicate 1 ((replicate 1 (cosa)) ++ (replicate 1 (sina)))) ++
      (replicate 1 ((replicate 1 (- sina)) ++ (replicate 1 (cosa))))
    where
      a = 13*pi/8
      cosa = cos a
      sina = sin a
```