Thesis for the Degree of Licentiate of Engineering

Towards a Functional Programming Language for Baseband Signal Processing

Anders Persson

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2014

Towards a Functional Programming Language for Baseband Signal Processing
Anders Persson

© 2014 Anders Persson

Technical Report 121L
ISSN 1652-876X
Department of Computer Science and Engineering
Functional Programming Research Group

Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers Reproservice
Göteborg, Sweden 2014

Towards a Functional Programming Language for Baseband Signal Processing
Anders Persson
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Developing software for resource constrained embedded systems is a daunting task. In addition to getting the functionality right, a programmer must consider several non-functional properties, including data layout, the target memory hierarchy, concurrency and parallelism. The programmer has to decide on a specific and reasonable solution on these properties upfront during development However, these decisions are necessarily based on incomplete information, since the optimial solution depends on the actual deployment on the target hardware.

In this thesis we explore a generative approach, with a staged functional domain specific language, to postpone many of these decisions to a much later stage of the development process.

The staged approach gives the progammer explicit control over much of the code generation by using combinators to build the application from reusable components and skeletons.

Currently, Feldspar is focused on the data processing part of signal processing in embedded systems. The Feldspar compiler generates single C functions that can be integrated into existing systems. The Feldspar language is based on a combination of shallow and deep embedding, which makes it modular both for the language developer and the end user. This modularity made it easy to add monadic expressions to support mutable updates and deterministic parallelism. Future work includes generation, deployment and orchestration of systems of data processing functions and control processing.

Additionally, this thesis presents a technique, and supporting library that allows the programmer to test, profile and analyze compiled Feldspar programs from the host language environment.

Keywords functional programming, domain specific languages, signal processing, code generation

Contents

Acknowledgements **vii**

Publications **1**

1 Introduction **3**
- 1.1 Background . . . 4
  - 1.1.1 Decisions, Decisions, Decisions . . . 5
  - 1.1.2 Decisions are neither Final nor Orthogonal . . . 6
  - 1.1.3 No Encoding of Decisions . . . 6
  - 1.1.4 Summary . . . 7
  - Reader's Guidelines . . . 7
- 1.2 The Feldspar Language . . . 8
  - 1.2.1 Functional Programming . . . 9
  - 1.2.2 Domain Specific Languages . . . 9
  - 1.2.3 Domain Specific Embedded Languages . . . 10
  - 1.2.4 Deep vs Shallow Embedding . . . 10
  - 1.2.5 Data Centric . . . 11
  - 1.2.6 Size Inference . . . 14
  - 1.2.7 Testing . . . 16
- 1.3 Extending Feldspar . . . 17
  - 1.3.1 Monads . . . 18
  - 1.3.2 Push Vectors . . . 18
- 1.4 Plugging Feldspar In . . . 19
  - 1.4.1 Data Types . . . 20
  - 1.4.2 Calling Feldspar functions from C . . . 20
  - 1.4.3 Calling C functions from Feldspar . . . 21
  - 1.4.4 Calling C functions from Haskell . . . 21
  - 1.4.5 Calling compiled Feldspar functions from Haskell . . . 22
  - 1.4.6 Automating the Process . . . 22
- 1.5 Case study: The Fourier Transform . . . 23
  - 1.5.1 Discrete Fourier Transform . . . 23
  - 1.5.2 Removing Redundant Computations . . . 24
  - 1.5.3 Cooley-Tukey Fast Fourier Transform . . . 25
  - 1.5.4 Push Vector FFT . . . 27

1.5.5 Benchmark 32
1.6 Design Flow of a Feldspar function 34
1.7 Related Work 34
1.8 Discussion 36
1.9 Future Work 37
1.10 Conclusion 37

Appendix A Terminology **39**
A.1 Parametric Polymorphism 39
A.2 Algebraic Data Type 40
A.3 Generalized Algebraic Data Type 41
A.4 Type Class 41

Paper A – Feldspar: A Domain Specific Language for Digital Signal Processing **47**

Paper B – The Design and Implementation of Feldspar **59**

Paper C – Generic Monadic Constructs for Embedded Languages **77**

Acknowledgements

I extend my sincerest thanks to my supervisor Mary Sheeran, for her support, insights and guidance. I am grateful that you encouraged me to make the jump into academia. I admire your perseverance reading and commenting on the many versions of this thesis.

I thank Emil Axelsson and Josef Svenningsson for helping me fulfil the Feldspar vision. You are a never ending source of knowledge and inspiration.

Thanks to former and current Feldspar project members, especially Gergely Dévai, Gábor Páli, Peter Jonsson and Karl-Filip Faxén for your hard work, ideas and insights. Thanks to my colleagues in the research group at Chalmers, who make this a wonderful place to work. Thanks to my colleagues at Ericsson for your support and encouragement.

Finally, and most importantly, I express my deepest gratitude to my wife and children for being loving and supporting of me during this time and attention consuming venture. Without you, this would not have been possible!

This research is funded by a grant from the Swedish Research Council.

Publications

This thesis includes the following papers.

Paper A Emil Axelsson, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András 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 Computer Society, 2010

This paper is based on my vision of the Feldspar language. The writing was a team effort.

Paper B Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. "The Design and Implementation of Feldspar – an Embedded Language for Digital Signal Processing". In: *22nd International Symposium, IFL 2010.* Vol. 6647. LNCS. 2011

This paper is based on my vision of the Feldspar language. The writing was a team effort.

Paper C Anders Persson, Emil Axelsson, and Josef Svenningsson. "Generic monadic constructs for embedded languages". In: *23rd International Symposium on Implementation and Application of Functional Languages, IFL 2011.* Vol. 7257. 2011

I am the lead author of this paper.

This paper was awarded the Peter Landin award for best paper at the *International Symopsium on Implementation and Application of Functional Languages, IFL 2011.*

Chapter 1

Introduction

Baseband Signal Processing is the part of a mobile base station that converts bit-streams of user data to and from resilient radio signals. This processing consists of several computationally intensive steps to prepare signals for transmission and to recover received signals. On the down-link side, the bit-streams are prepared for transmission by adding forward error correcting encoding. The resulting bit-streams are modulated and sent to the radio. On the uplink side, the opposite processing is performed, by first demodulating the signal and then unwrapping the error correcting code. But first, since the radio transmission is imperfect, the signal has to be recovered from distortions, reflections and interference. All of these steps have to be performed within tight latency requirements, on processors with limited memory, which means that computations have to be efficient.

Tough latency and resource requirements, which baseband signal processing shares with many other kinds of embedded software, have a great influence on how the software is developed. There is a strong focus on performance, and other less immediate aspects such as portability and re-usability have to stand back.

I assert that this focus on performance is a *premature optimization*!

It is premature, partly because the highly optimized code is difficult to maintain and to port to other processors. But more importantly because it forces the developers to make too early decisions based only on local information (section 1.1.1). Once the decisions are made, they are difficult to change since they, through the implementation, are tightly coupled with each other (section 1.1.2).

Predominantly, embedded systems are developed using low-level, imperative languages. This is driven by said desire to extract maximum performance from the processor and memory subsystem. Lately, the need for high efficiency has also been driven by increased power efficiency requirements. The low-level languages have

constructs that are well suited to fine-grained control of the processor and memory subsystem.

However, while good for extracting maximum performance from a processor, the low-level languages provide little or no means of encoding the decisions made during design and implementation (section 1.1.3).

Functional programming provides abstraction, generalization and modularity through rich type systems, higher-order functions and lazy evaluation (section 1.2.1). These properties enable the programmer to build the application by assembling pre-verified components into larger systems.

Functional programming languages are rarely considered for embedded systems development. The main reasons are that it is difficult to give performance guarantees and resource bounds on functional programs, especially with lazy evaluation.

This thesis presents a work in progress to bring functional programming into the domain of embedded systems. While motivated by the author's experience in development of baseband signal processing software, the language and techniques in this thesis are applicable to embedded software development in general.

The Feldspar language (section 1.2) is a functional domain specific language (section 1.2.3) designed to be suitable for implementation of embedded software in the domain of digital signal processing. The syntax and semantics of the Feldspar language are designed to give the programmer predictable performance.

1.1 Background

Traditionally, Digital Signal Processing (DSP) applications are written using hand tuned imperative code, often in languages such as C or assembler. Low-level languages are used since their syntax and semantics are closely modelled on the execution model of the processor, which makes it possible to unlock the performance critical in applications such as baseband processing in mobile base stations.

However, while good for extracting maximum performance from a processor, the low-level languages are a poor match for the mathematical notations and concepts used in designing the algorithms.

Further, in signal processing, as in other fields of embedded software, the programmer has to take several design constraints into account. To make the application fit the constraints, it is often optimized heavily. But too early optimization forces the programmer to make early design decisions, resulting in a rigid architecture. Also, it is hard to use generic frameworks such as collection and traversal libraries, without incurring a runtime cost.

1.1.1 Decisions, Decisions, Decisions

One reason for the low level code is that it gives fine control of *what* is executed *where* and *when*. The programmer *must* exercise this control and thus the programmer needs detailed knowledge of the execution pipeline and of the runtime characteristics of the real-time operating system.

However, this fine grained control comes at a cost. The problem of implementing an algorithm has been transformed into the problem of implementing the algorithm for a specific architecture.

In effect, the application programmer needs to consider the function to be implemented at several levels simultaneously:

(1) **Functionality** What does the function add to the system?

(2) **Architecture** How should the code be structured to make its functionality clear and maintainable?

(3) **Parallelism** Can the algorithm be executed in parallel over the data?

(4) **Concurrency** How should locking/atomicity be handled?

(5) **Scheduling** In what order should different concurrent or parallel parts be allowed to access resources?

(6) **Data Layout** How should the data be stored and accessed?

(7) **Memory Hierarchy** How is the memory structured on the processor.

(8) **Code size** Is the program small enough to fit into local memories or caches?

(9) **Data size** Is the data set small enough to fit into local memories or caches?

(10) **Locality** Does data need to be transferred between execution units (cores, processors)

(11) **Transport** Can data be transferred in the background using Direct Memory Access (DMA) or pre-fetching?

(12) **Caching** Is caching available in the memory hierarchy?

(13) **Instruction Set** Single Instruction Multiple Data (SIMD), Very Long Instruction Word (VLIW), 8, 16, 32 or 64-bit? Intrinsics?

(14) **Accelerators** Can parts of the algorithm be accelerated in hardware?

(15) **Heterogeneity** Does the hardware platform include different processor architectures?

(16) **Portability** Can the same code be used for different hardware architectures? How much effort is involved in rewriting (porting) it to a different platform?

The list above is not exhaustive, but it illustrates some of the diverse and interlocking concerns that the programmer has to consider during development.

1.1.2 Decisions are neither Final nor Orthogonal

Each and every developer in a project has to consider the questions in section 1.1.1 for every function they write. Based on their previous knowledge and experience, the developers *may come to different conclusions each time*.

Also, the questions above are not orthogonal. A decision for one question may have consequences for other questions.

Consider the example below. Numbers in parenthesis are references back to the list of questions in section 1.1.1.

> For example, the code size (8) and data size (9) combined with the lack of caching (12) on a particular version of hardware may force (2) parallelizing (3) a function over several cores (4,10). However, running a large set of tasks concurrently (4) with other tasks mean careful prioritization (5) has to be made between tasks.
>
> Of course, for a different version of the hardware (16) the decisions may be very different.

1.1.3 No Encoding of Decisions

The programming language of choice for embedded systems is often a dialect of C. Since C has few abstractions, and almost no type information, it is difficult to encode and communicate the decisions as part of the source code. Instead, some decisions get documented in design specifications, code comments or preprocessor pragmas. Sometimes, the decisions are not encoded at all, either because the developer forgot or did not think it important, possibly due to lack of knowledge or experience. Yet another reason for the lack of decision encoding, is that the question was not deemed relevant at the time the source code was written.

The lack of encoded information makes it very difficult to enforce or change project wide architectural decisions. Since the encoded information that still exists is non-formal, e.g. comments or textual specifications, it is difficult to check that encoded decisions are reflected in the source code. Thus the risk increases that parts of the code do not work together or interfere in subtle ways.

Consequently, development is error-prone and has a significant lead-time. For much the same reasons, code portability is limited, even when code is ported between processors from the same vendor.

1.1.4 Summary

Forcing early decisions is a form of premature optimization. It is premature since not all information is available when the decision has to be made.

> We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

Donald Knuth, [17]

Section 1.1 and subsections try to shine a light on the many, interlocking, issues that have to be considered when developing high-performance embedded software. Combined with the low-level languages, the interlocking nature of these issues make it well nigh impossible to create software that is both portable and high-performance. The low-level languages force the programmer, prematurely, to focus on low-level details instead of functionality and architecture. The decisions have to be made based on local information. Together, these problems lead to *low re-usability* and *low modularity*.

In response to these challenges, we are developing the Feldspar language.

The language (section 1.2 and paper A), design decisions (section 1.2.5 and paper B) and a use case (section 1.5) are presented in the following chapters and papers.

Reader's Guidelines

The following sections will refer to concepts that may be unfamiliar to readers without Haskell experience. Some of these concepts are explained in appendix A.

1.2 The Feldspar Language

Many of the issues in section 1.1.1 are related to a lack of *modularity*. Some of the issues are directly related to modularity, such as functionality (1) and architecture (2). But the relations of other issues to modularity are less obvious. For example, we would like to treat parallelism (3), concurrency (4) and scheduling (5) as modular concepts with respect to the functionality. In many languages, especially imperative languages, this is very difficult to do.

It is of course possible to add code for parallelism into the function and use runtime conditional statements to switch between the different implementations. But often it is also necessary to change the architecture and to repartition the code into smaller pieces to make it parallel. These changes lead to extra control paths and may interfere with compiler optimizations.

Ideally, we would like to treat all of the issues modularly and separately. Then, we could create libraries of reusable (partial) solutions to the issues, and combine them to solve larger and more complex problems. See section 1.5 for an example.

In the seminal paper "Why Functional Programming Matters" [15], Hughes argues that the modularity of software can be greatly improved by the use of functional programming.

> Therefore, to increase one's ability to modularise a problem conceptually, one must provide new kinds of glue in the programming language.
>
> John Hughes, [15]

These new kinds of glue are higher-order functions and lazy (non-strict) evaluation. In Feldspar we make extensive use of higher-order functions to build combinator libraries. Non-strict evaluation gives the ability to rearrange the evaluation order of an expression. Exploiting this makes it possible to avoid evaluating unused intermediate results. Feldspar provides rewriting optimizations to this effect.

The Feldspar language is a functional Domain Specific Language (DSL) designed to be suitable for implementation of embedded software in the domain of digital signal processing functions. Feldspar is a staged language, which compiles to C code. The syntax and semantics of Feldspar are designed to give the programmer predictable performance.

The Feldspar language and compiler are released as open source software[1]. The code in this thesis is implemented using version 0.7 of both `feldspar-language` and `feldspar-compiler`.

To install the language and compiler on a machine with the Haskell Platform:

```
$ cabal install feldspar-compiler-0.7
```

[1] Feldspar website: https://feldspar.github.io

1.2.1 Functional Programming

In a functional language with *pure semantics*, functions must always return the same value when given the same arguments. In particular, this means that the functions cannot carry any internal state or perform any other implicit interactions with the environment. This property allows pure functions to evaluate in parallel as long as their data dependencies are satisfied.

Also, since the functions do not implicitly interact with the environment, it is possible to apply high-level optimizations through local equational reasoning. Such high-level optimizations may not be valid or require global reasoning in non-pure languages.

A higher-order function, often referred to as a *combinator*, allows the programmer to assemble new functions from smaller functions. For example, in Haskell, the function `map :: (a → b) → ([a] → [b])` takes a function `f :: a → b` that given an `a` produces a `b`, and returns a function that given a list `[a]` produces a list `[b]`. Note that the last parentheses are optional, but they are included here to show that map is a function from one function type to another.

The `map` function is only concerned with traversing the list and it uses the function `f` to transform each element of the list. Since the types `a` and `b` are polymorphic (see section A.1), the map function has no way of inspecting the elements and thus, the separation of concerns is enforced.

In Haskell, many of the small functions and constructions used in everyday programming are in fact not syntax, but combinators. For example, a crucial concept such as *function composition* is implemented as a library function in the Haskell `Prelude`.

```
1  -- | Function application operator
2  ($) :: (a → b) → a → b
3  f $ a = f a
4
5  -- | Function composition operator
6  (.) :: (b → c) → (a → b) → a → c
7  f . g = λx → f (g x)
```

The application operator `$` is right-associative and allows us to omit parentheses in some cases, e.g. `f $ g $ h x = f (g (h x))`. The implementation of the composition operator `.` is an example of an *anonymous function* or lambda abstraction. In Haskell, an anonymous function is defined by a λ followed by patterns. The body of the function is to the right of the arrow $\rightarrow$.

1.2.2 Domain Specific Languages

A *Domain Specific Language* (DSL) [14, 32, 13] is a language tailored to a certain problem or solution. The syntax and constructs in the language are designed to be

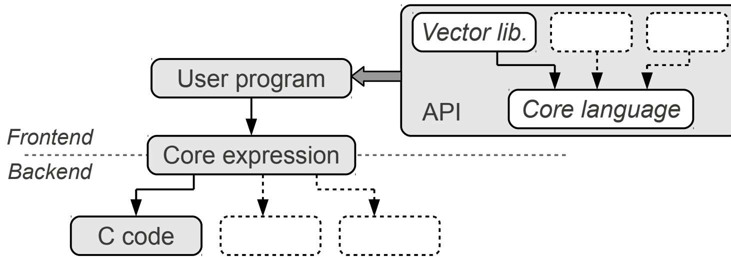

Figure 1.1: Feldspar language architecture

as close as possible to the terms used by domain experts reasoning about a problem or solution.

Examples of DSLs include LaTeX for document preparation, Makefiles for dependency and build specification, and VHDL for hardware description.

An important reason to implement domain specific languages is to encapsulate concepts and idioms into reusable building blocks. Using combinators, these building blocks can be assembled into larger programs.

1.2.3 Domain Specific Embedded Languages

One way of implementing a domain specific language is as a library in a host language. This technique is called *embedding* [14, 13]. The main purpose of embedding a DSL is to save on the implementation effort. An embedded language can reuse many of the parts and services of the host language and compilers. For example, the syntax, parser, type checker and additional libraries are commonly reused.

The functional language Haskell [22] is often used as a host language. Many popular libraries (parser generation, pretty printing, hardware design [7, 3], testing [10], image synthesis and manipulation [13], and GPU programming [29]) have been written as domain specific languages in Haskell.

1.2.4 Deep vs Shallow Embedding

Embedding comes in different flavours. For compilation of embedded domain specific languages, it is natural to use an *Algebraic Data Type* (ADT) to represent the *Abstract Syntax Tree* (AST). This is known as *deep embedding*. In a deep embedding, the ADT can become large, since it needs to have a constructor for each supported construct in the syntax. For an introduction to (generalized) algebraic data types, see sections A.2 and A.3.

Unlike a deep embedding, a *shallow embedding* does not require an abstract syntax tree. Instead, the embedding is encoded as functions representing its semantics. Necessarily, a shallow embedding is limited to one interpretation – the semantics used for its representation.

In Feldspar, we use a combination of deep and shallow embeddings. The core language is a deep embedding, which makes it easy to analyse, interpret and optimize. However, user-facing libraries use a shallow embedding to provide a nicer syntax in the API. See figure 1.1.

For further details on the technique of combining deep and shallow embeddings, see reference [28].

Furthermore, since Feldspar program specifications have lazy semantics, functions are not evaluated until they are needed. This means that only the parts that contribute to the end result are considered when building the syntax tree.

Feldspar uses the `syntactic` library [2, 28] to implement the deep embedding. The `Syntactic` class provides a bridge between shallow embeddings and the deep embedding.

1.2.5 Data Centric

Programming in a functional language is different from programming in an imperative language. In particular, it is common to transform data in bulk operations rather than loops. Many computations can be expressed as sequences of transformations, where each transform operates on the entire data set instead of individual elements. This allows for a high-level programming style based on reusable functions rather than loop- and index-based traversals.

As a simple example, we take the scalar product, defined as

$$a \cdot b = \sum_{i=0}^{N-1} a_i b_i \tag{1.1}$$

where a and b are N-element vectors.

In an imperative language such as C, we write this as:

```
int32_t scalarProd(int N, int32_t *xs, int32_t *ys)
{
  int32_t sum = 0;
  for (int i = 0; i < N; i++)
  {
    sum += xs[i] * ys[i];
  }
  return sum;
}
```

At first glance, the C code looks as if it is a good representation of equation (1.1). However, there are a few problems in this definition.

- The implementation is specialized to the signed integer type `int32_t`. The programmer has to write the same algorithm over again for other types. (Lines 1 and 3).
- The caller has to assert that both vectors `xs` and `ys` have at least `N` elements. (Lines 4 and 6)
- It is not compositional. Hence, it cannot be combined and optimized together with other functions, for example the producers of `xs` and `ys`, without breaking encapsulation.

Writing the scalar product using the same imperative style in Feldspar is possible, but not idiomatic:

```
1 scalarProdInt :: Data Length → Data [Int32] → Data [Int32] → Data Int32
2 scalarProdInt n xs ys = forLoop n 0 $ λi s → s + xs!i * ys!i
```

In the code above, the type `Data a` represents a Feldspar program fragment. An expression with type `Data Length` produces a value of type `Length`. The notation with square brackets `[a]` is used to represent arrays with elements of type `a`. The function

```
1 forLoop :: Data Length → Data a → (Data Index → Data a → Data a) → Data a
```

is a combinator that captures the iteration scheme. The first parameter is the number of iterations. The second is the initial loop state and the third parameter is the iteration step function, which calculates a new state from the loop index and the old state. The step function, or loop body, also has access to the variables `xs` and `ys` since they are in scope.

We can improve the implementation, by making it polymorphic in its element type, and by calculating the length `n` as the minimum of the lengths of the vectors.

```
3 scalarProd2 :: (Type a, Numeric a)
4             => Data [a] → Data [a] → Data a
5 scalarProd2 xs ys = forLoop n 0 $ λi s → s + xs!i * ys!i
6   where
7     n = min (getLength xs) (getLength ys)
```

This is, however, still not idiomatic Feldspar, and the code is still fragile. What if the programmer mistakenly extracted the value from `xs` twice in the loop, instead of one value from each of `xs` and `ys`? That code would still type check but it would not behave correctly.

In idiomatic Feldspar, we strive to build more complex functions by composing smaller, pre-verified, functions, see section 1.2.7. The compositional style often gives shorter, more succinct definitions:

```
1  data Vector a = Indexed (Data Length) (Data Index → a)
2
3  zipWith :: (a → b → c) → Vector a → Vector b → Vector c
4  zipWith f xs ys = indexed n $ λi → f (xs!i) (ys!i)
5    where
6      n = min (length xs) (length ys)
7
8  sum :: (Num a) => Vector1 a → Data a
9  sum = fold (+) 0
```

Listing 1.1: Parts of the Vector library API

```
8   scalarProd :: (Type a, Numeric a)
9              => Vector (Data a) → Vector (Data a) → Data a
10  scalarProd xs ys = sum (zipWith (*) xs ys)
```

One of the most important extensions to core Feldspar is the Vector library (`Feldspar.Vector`). This library provides a rich set of combinators and functions to work with flat or possibly nested arrays. A small subset of the API can be found in listing 1.1. The purpose of each combinator is to capture a pattern or aspect and make it reusable.

The scalar product is calculated by first multiplying the vectors `xs` and `ys` element-wise (`zipWith`) and then reducing the result with addition (`sum`).

The idiomatic Feldspar implementation is polymorphic, and can accept elements of any type that belongs to the `Numeric` and `Type` type classes. For more information on type classes, see section A.4. By providing a type signature, we can compile `scalarProd` for 32-bit signed integers:

```
11  ghci> icompile' defaultOptions{rules=nativeArrayRules} "scalarProd" (scalarProd ::
         Vector (Data Int32) -> Vector (Data Int32) -> Data Int32)
12  void scalarProd(int32_t v0[], int32_t v1[], int32_t * out)
13  {
14    uint32_t len0;
15
16    len0 = min(getLength(v0), getLength(v1));
17    *out = 0;
18    for (uint32_t v2 = 0; v2 < len0; v2 += 1)
19    {
20      *out = (*out + (v0[v2] * v1[v2]));
21    }
22  }
```

We note that the code still needs to calculate the minimum of the lengths of the input vectors. While Feldspar has support for size inference (see section 1.2.6) to help calculate the lengths, the programmer may need to provide information

```
1  data Range a = Range { lowerBound :: a, upperBound :: a }
2  data AnySize = AnySize
3
4  type family   Size a
5  type instance Size Int32 = Range Int32
6  type instance Size Float = AnySize
7  type instance Size [a]   = Range Length :> Size a
8  type instance Size (a,b) = (Size a, Size b)
```

Listing 1.2: Partial interface of Size inference

about the lengths. To mimic the C implementation, we wrap the generic `scalarProd` function to assert the length of the vectors:

```
23 scalarProdN n xs ys = scalarProd (newLen n xs) (newLen n ys)
```

With the assertion propagated into `scalarProd`, the length calculations are optimized away.

```
24 ghci> icompile' defaultOptions{rules=nativeArrayRules} "scalarProdN" (scalarProdN
        :: Data Length -> Vector (Data Int32) -> Vector (Data Int32) -> Data Int32)
25 void scalarProdN(uint32_t v0, int32_t v1[], int32_t v2[], int32_t * out)
26 {
27   *out = 0;
28   for (uint32_t v3 = 0; v3 < v0; v3 += 1)
29   {
30     *out = (*out + (v1[v3] * v2[v3]));
31   }
32 }
```

1.2.6 Size Inference

Generic programs should have the same semantics for all inputs. However, when certain conditions are met, the actual implementation can be optimized to save execution time or memory or even improve concurrency or parallelism. In the most generic case, it is necessary to introduce conditional control flow to select between different code paths. But these conditionals may interfere with optimizations and result in a performance overhead. If possible we want to short-circuit these conditionals by calculating them at compile time.

To this end, Feldspar employs *constant folding* and *size inference* as part of the rewriting optimizations.

Every expression in a Feldspar program is annotated with the size of its result. The interface to size inference is shown in listing 1.2. To make it possible to have differently typed annotations on expressions of different types, we use a type family

`Size`. The type family `Size a` acts as a function on the type level. It calculates the actual type of the annotation from the expression type.

For example, integers are annotated with a `Range` consisting of lower and upper inclusive bounds. For arrays (denoted as `[a]`), the size is represented by the range of the length parameter and the size information for the stored elements. Size propagation is not performed for floating point values, and thus they are annotated with the singleton type `AnySize`.

Consider the following array:

```
1  parallel 5 (*2) :: Data [Int32]
```

Here, `parallel :: Data Length → (Data Index → a) → Data [a]` is the constructor of arrays in Feldspar. The first parameter is the number of elements in the array, and the second parameter is a function from an index to a value. This style of representing arrays is borrowed from the language Pan [13].

We inspect the size annotation with `drawDecor`

```
1  ghci> drawDecor $ parallel 5 (*2)
2  <<[WordN] | Range {lowerBound = 5, upperBound = 5} :> Range {lowerBound = 0,
       upperBound = 8}>>
3   [0,2,4,6,8]
```

The length is a singleton `Range {5, 5}` as it is a constant. Perhaps more interestingly, the range of the elements is the smallest `Range {0, 8}` that will fit all values in the array.

The size annotations are used in analyses and optimizations in Feldspar, e.g.

- Singleton ranges are replaced by literal constants.
- Comparisons (==,<,>,<=,>=) can be defaulted if the ranges of the operands are disjoint.

Note that these optimizations happen even for expressions with free variables.

Consider the function to calculate the minimum of two values; `min`.

```
1  min :: (Ord a) => Data a → Data a → Data a
2  min x y = condition (x < y) x y
```

With constant folding, the condition can be short circuited if `x` and `y` are constants known at compile time. The size inference and propagation improves on constant folding by allowing us to compare ranges of values instead of actual values. It is sufficient that the compiler can deduce that the ranges of `x` and `y` are disjoint and that the upper bound of the range of `x` is smaller than the lower bound of the range of `y`.

The size inference in Feldspar is not perfect. For example, it can not yet infer the size of the result of a `for`-loop. It is possible by taking a fix-point of the analysis, but we have not implemented it yet.

To help the inference the Feldspar language and libraries provide a set of annotation functions:

```
type SizeCap a = Data a → Data a
between   :: (Type a, Bounded a, Size a ~ Range a)
          => Data a → Data a → SizeCap a
cap       :: Type a => Size a → SizeCap a
notAbove :: (Type a, Bounded a, Size a ~ Range a)
          => Data a → SizeCap a
notBelow :: (Type a, Bounded a, Size a ~ Range a)
          => Data a → SizeCap a
sizeProp :: (Syntax a, Type b)
          => (Size (Internal a) → Size b) → a → SizeCap b

newLen    :: (Syntax a)
          => Data Length → Vector a → Vector a
withLen   :: (Syntax a, Syntax b)
          => Data Length → (Vector a → Vector b) → Vector a → Vector b
```

With these functions the programmer can annotate the code with assertions about the size of an expression. The `SizeCap` combinators will `meet` the cap the infered size. For example `newLen` informs the inference that a vector has a specific length. The expression `withLen l f` assures that the length is preserved when calling the function `f`. To aid the size inference, we use these annotation functions in sections 1.5.3 and 1.5.4.

1.2.7 Testing

Feldspar programs are constructed by composing smaller parts into larger programs, and thus it is important that the parts and the combinators are functionally correct, and that the composed programs preserve this property. The Feldspar language is embedded in Haskell, which means that we can use existing infrastructure for property based random testing to check the functionality of Feldspar programs. The Haskell package `QuickCheck` [10] provides the frameworks needed.

For example, consider a function `rev` that reverses the order of elements in an array.

```
rev :: Data [Word8] → Data [Word8]
rev arr = parallel len $ λi → arr ! (len - 1)
  where
    len = getLength arr
```

To check that this function has the correct functionality, we state properties about it, e.g. that the elements of the array should be presered.

```
5 prop_preseves_elements f xs =
6     Data.List.sort xs === Data.List.sort (eval f xs)
```

Here `Data.List.sort` is a Haskell library function that sorts the elements of a Haskell list in ascending order. The Feldspar `eval` function evaluates the Feldspar syntax tree to a function that accepts Haskell arguments.

We run the property with the `quickCheck` function, which will apply our `rev` function to random input and check that the `prop_preseves_elements` property holds.

```
7  ghci> quickCheck (prop_preseves_elements rev)
8  *** Failed! Falsifiable (after 4 tests):
9  [1,0]
10 [0,1] /= [0,0]
```

`QuickCheck` alerts us to the fact that our `rev` function is faulty; it returns the same element in all positions. The last line in the output shows the actual values tried. Each list contains only two elements since `QuickCheck` has shrunk the inputs to provide us with a smaller counter example, to make it easier to pinpoint the error.

We correct our error and subtract `i` before indexing:

```
11 reverse :: Data [Word8] → Data [Word8]
12 reverse arr = parallel len $ λi → arr ! (len - 1 - i)
13   where
14     len = getLength arr
```

and now the property holds for all tested inputs.

```
15 ghci> quickCheck (prop_preseves_elements reverse)
16 +++ OK, passed 100 tests.
```

However, `QuickCheck` will run the property several times, by default a hundred. In each such run we call `eval` to evaluate the expression, which might be a costly operation since it runs on the unoptimized expression tree. In section 1.4, we present a technique that can accelerate the evaluation, by compiling the expression to C and linking the object file back into Haskell.

1.3 Extending Feldspar

The Feldspar language is designed to be modular and extensible (figure 1.1 and paper B [4]). The core language (the deep embedding) contains the constructs of the language. These constructs are the tangible components of the language and form the basis of analysis, evaluation, optimization, translation and compilation. On top of these core constructs we build libraries of ephemeral functions that build

and modify trees of core constructs. The library functions are merely syntactic sugar that encapsulate concepts and make the language easier to work with for the programmer.

Most additions to the Feldspar language are made as libraries, without the need to extend the core language.

1.3.1 Monads

One addition that extends the core language with new constructs is the support for *monadic expressions*. Feldspar is a language with pure expressions, i.e. computations that are free of side effects. This makes the expressions easy to analyse, optimize and parallelize. However, this purity makes it impossible to efficiently express algorithms that, for performance, rely on a specific evaluation order, access pattern or destructive updates of data structures. The problem manifests itself in both extra memory for storing intermediate results and extra execution time for copying data.

Monads, popularized by Wadler [30], provide ways of adding impure computations to languages with otherwise pure semantics.

To solve issues with extensive copying in Feldspar generated C code, we added support for monadic expressions. This work is presented in paper C [21].

In particular, we have implemented two monads in Feldspar: the Mutable monad and the Par monad. The Mutable monad, which is described in paper C [21], adds support for expressions with destructive, and possibly in-place, expressions. The Par monad is modelled after the Haskell monad with the same name [19] and supports deterministic parallelism.

1.3.2 Push Vectors

To enable expression of array and vector computations with mutable updates or specific evaluation orders, Feldspar provides the `MutableArray` core construct. This construct allows the programmer full control over allocation of arrays and assignment of array elements. However, this interface is very low level and imperative.

To make it easier to program with the mutable arrays we added a construct to the `Vector`-library; the `PushVector`. The name `PushVector` warrants an explanation. The original vectors in the vector library are modeled on a design from the language Pan [13]:

```
1 data Vector a = Indexed { length :: Data Length, index :: Data Index → a }
```

A `Vector` is a pair of a length (number of elements) and a function that given an index in the vector calculates the element. All of the elements of a `Vector` are

computed independently and can therefore be computed in parallel. This design also allows for a lightweight yet powerful implementation of vector fusion (see paper B). The consumer is in control of the evaluation order and conceptually, the elements are pulled out of the vector.

With the `PushVector` on the other hand, the producer is in control of the evaluation order and the elements are pushed into place in some memory. The two flavours of vectors are duals. Our `PushVector` type is based on the same idea as the push arrays in reference [11].

The `PushVector` type is slightly more complex than the `Vector`.

```
1 data PushVector a
2   where
3     Push :: ((Data Index → a → M ()) → M ()) → Data Length → PushVector a
```

The first argument to `Push` is a function that decides the evaluation order. It will call the `Data Index → a → M ()` function to write each element in turn.

As an example, consider a function that reverses the elements in a `Vector` by swapping elements starting from both ends.

```
1 reverse :: (Syntax a) => Vector a → PushVector a
2 reverse v = Push loop (length v)
3   where
4     loop write = forM (1 + length v / 2) $ λi → do
5                    let j = length v - 1 - i
6                    write i $ v ! j
7                    write j $ v ! i
```

For larger examples see the case study in sections 1.5.3 and 1.5.4.

It is worth noting that `PushVectors` fuse in the same way as `Vector` and that is possible to convert at `Vector` to a `PushVector` without allocating memory. However, to convert from a `PushVector` to a `Vector` allocation is necessary.

1.4 Plugging Feldspar In

This section presents previously unpublished work by the author.

Programs written in Feldspar need to communicate with programs written in other languages. In the following sections we will detail how to call out to functions written in other languages as well as calling into a Feldspar function from other languages.

Finally, we present a method and apparatus for using compiled Feldspar programs as part of the development and testing work flow.

1.4.1 Data Types

In code generated from Feldspar, the type of arguments, local variables and results are uniquely determined by the Feldspar types.

Feldspar	C-type
`Int8`	`int8_t`
`Word64`	`uint64_t`
`[a]`	`struct array *`
`(Int8,Word16)`	`struct s_signedS8_unsignedS16`

Table 1.1: Examples of Feldspar-to-C type conversions

Arrays are represented using a data structure that carries the length and a pointer to the actual data.

```
// ... from feldspar_array.h ...
struct array {
    void *   buffer;    // pointer to the buffer of elements
    int32_t  length;    // current number of elements in the array
    int32_t  elemSize;  // size of elements in bytes
    uint32_t bytes;     // number of bytes the buffer can hold
};
```

Array structures can be created and manipulated with the API provided by the `feldspar_array.h` header file.

1.4.2 Calling Feldspar functions from C

The code generated by the Feldspar compiler has a simplified calling convention.

- Scalar values are passed by value
- Structured values (structs, unions, arrays) are passed by reference
- Arrays are represented using a data structure `struct array`, see section 1.4.1
- All functions return `void`
- Return values are passed through caller provided pointers.

With this convention, the function `feldspar_function :: Data [a] → Data Word32` will be represented by the following function in C.

```
void feldspar_function(struct array * v0, uint32_t * out)
```

It may be argued that this calling convention makes it difficult for a C compiler to optimize such a function in context. However, the idea is that Feldspar should be used for any such optimization and that the fully optimized code can be called without further ado.

1.4.3 Calling C functions from Feldspar

A Feldspar function can call out to arbitrary C functions using the Feldspar Foreign Function Interface (FFI). The `foreignImport` function in the `Feldspar.Core.Frontend.FFI` module takes two parameters, the name of the external function, and the Haskell semantics of the function. The Haskell semantics is needed when evaluating the Feldspar expression. Note that evaluation can happen as part of optimization rewrites, e.g. constant propagation. To ensure correct evaluation, the semantics should match the semantics of the imported function. If the function is available on the host system, section 1.4.4 gives a way of reusing the C semantics.

For example, the following code defines a Feldspar function

```
1  max :: (Type a, Ord a)
2      => Data a → Data a → Data a
3  max = foreignImport "MAX" (λx y → if x Prelude.< y then y else x)
```

which translates into a call of the function `MAX` in the generated C code:

```
4  ghci> icompile (Main.max -:: tData tIndex ↣ id)
5  void test(uint32_t v0, uint32_t v1, uint32_t * out)
6  {
7    *out = MAX(v0, v1);
8  }
```

1.4.4 Calling C functions from Haskell

Using the Haskell FFI, a Haskell program can call out to arbitrary external functions:

```
9  foreign import ccall "sin" c_sin :: Float → Float
```

This instructs the Haskell linker to *statically* import the C function `sin` as the Haskell function `c_sin`.

We can make use of the imported function as the semantics of a Feldspar `sin` function.

```
10 sin :: Data Float → Data Float
11 sin = foreignImport "sin" c_sin
```

1.4.5 Calling compiled Feldspar functions from Haskell

Using the Haskell FFI and the Feldspar Compiler, we can import compiled Feldspar functions and call them from Haskell.

But first we have to consider the Haskell type of a compiled Feldspar function. Due to our calling convention above, the translation from a Feldspar type to a Haskell type is straight forward.

The Feldspar function

```
1 sin :: Data Float → Data Float
```

has the following C signature

```
1 void sin(float v0, float * out);
```

We note that the C signature returns `void`, which in Haskell is modelled using `()`. Also, the second parameter is a pointer to a preallocated area capable of storing a `float`, which we model using `Ptr Float`.

Since we can use the `Ptr` only in the `IO` monad we arrive at the following Haskell type:

```
1 foreign import ccall "feldspar_sin" h_sin_worker
2     :: Float → Ptr Float → IO ()
```

By creating a wrapper around the `h_sin_worker` function we can allocate the `Ptr`, call the `h_sin_worker` function and extract the result.

```
3 h_sin_wrapper :: Float → IO Float
4 h_sin_wrapper a =
5   alloca $ λv1 → do
6     h_sin_worker a v1
7     peek v1
```

With these declarations, we can call the `h_sin_wrapper` function to get the value calculated by our Feldspar function.

```
8 ghci> h_sin_wrapper 45
9 0.8509035
```

1.4.6 Automating the Process

The Haskell FFI also supports dynamic loading and linking of external symbols, e.g.:

```
1 foreign import ccall "dynamic" h_sin_ptr
2     :: FunPtr (Float → Ptr Float → IO ()) → (Float → Ptr Float → IO ())
```

The external name is replaced by `"dynamic"` and the type represents a function converting a function pointer to a Haskell function.

With dynamic loading and linking all the pieces to work with compiled Feldspar functions are available. However, the foreign import statements, the compilation code, the loading code and the marshalling wrapper is cubmbersome to write.

Using Template Haskell the entire infrastructure can be generated from the name and type of a Feldspar expression. The expression `$(loadFun ['scalarProd])` expands to all the necessary declarations (some details are elided for brevity).

```
1     foreign import ccall unsafe "dynamic" c_scalarProdInt_factory
2     c_scalarProdInt_builder :: ...
3     c_scalarProdInt_raw ::
4         WordN → Ptr (SA Int32) → Ptr (SA Int32) → Ptr (SA Int32) → IO ()
5     c_scalarProdInt_worker :: WordN → [Int32] → [Int32] → IO Int32
6     c_scalarProdInt :: WordN → [Int32] → [Int32] → Int32
```

The `_builder` function is responsible for compiling and loading the Feldspar expression. The `_raw` function is a very thin wrapper around the loaded function, and the `_worker` wraps the `_raw` function in marshalling code.

See the benchmarks in section 1.5.5 and listing 1.11 on page 33 for a larger example.

1.5 Case study: The Fourier Transform

1.5.1 Discrete Fourier Transform

The Discrete Fourier Transform (DFT) is an important algorithm in signal processing. It can be calculated using the following equation

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \quad k = 0, \ldots, N-1 \tag{1.2}$$

where x_n are complex values.

The DFT can be expressed in Feldspar as

```
1 dft :: Vector1 (Complex Double) → Vector1 (Complex Double)
2 dft v = indexed len $ λk → sum $ zipWith (*) v (roots k)
3   where
4     len     = length v
5     roots k = indexed len $ λn →
6         cis ((-2) * pi * i2f (k*n) / i2f len)
```

where `cis r = exp (0 +. r)`.

However, this naive implementation of DFT is very inefficient.

```
7  ghci> icompile' defaultOptions{rules=nativeArrayRules} "dft" (dft -:: newLen 8 >->
       id)
8  void dft(double complex v0[], double complex out[8])
9  {
10   double complex e0;
11 
12   assert(out);
13   for (uint32_t v1 = 0; v1 < 8; v1 += 1)
14   {
15     e0 = (0.0f+0.0fi);
16     for (uint32_t v2 = 0; v2 < 8; v2 += 1)
17     {
18       e0 = (e0 + (v0[v2] * cis_fun_double((((double)((v1 * v2))) *
       -6.283185307179586) / 8.0))));
19     }
20     out[v1] = e0;
21   }
22 }
```

As we can see from the code above, the implementation has two nested loops and $\mathcal{O}(N^2)$ complexity. We will address this problem in section 1.5.3.

1.5.2 Removing Redundant Computations

The `dft` function above shows another deficiency. For every element calculated in the innermost loop, it also calculates the twiddle factor $e^{-i2\pi k\frac{n}{N}}$. Instead of recalculating the twiddle factors, we can create a table of pre calculated factors and just make a lookup inside the loop.

To achieve the tabling, we rewrite the `dft` definition to expose the `roots` vector. See listing 1.3.

Note that exposing the vector as a parameter gives us the *opportunity* to table the calculations. However, Feldspar will eagerly in-line and fuse operations and thus `dftParam` and `dft` will generate the same code.

We can check that they agree by stating it as a property and test it with QuickCheck [10].

```
34 ghci> quickCheck $ λxs → (fmap cround $ eval dft xs) === (fmap cround $ eval
       dftParam xs)
35 +++ OK, passed 100 tests.
```

Note that in general it is not safe, due to error propagation through numerical operations, to compare floating point number for equality. In the example above

```
23  twids :: Data Length → Vector1 (Complex Double)
24  twids len = indexed len $ λi → cis ((-2) ∗ pi ∗ i2f i / i2f len)
25
26  dft' :: (Syntax a, Num a)
27       => Vector a → Vector a → Vector a
28  dft' ws v = indexed len $ λk → sum $ zipWith (∗) v (roots k)
29    where
30      len     = length v
31      roots k = permute (λl i → (k∗i) `mod` l) ws
32
33  dftParam v = dft' (twids (length v)) v
```

Listing 1.3: DFT parameterized on twiddle factors

the function `cround` rounds each complex number to six decimal places before comparison.

As part of optimizations, Feldspar expressions are analysed for potential common sub-expressions. Common sub-expressions are shared and hoisted as far up as possible in the syntax tree, stopping just below the innermost binder of the free variables in the sub-expression.

This hoisting, also known as *code motion*, is however, not enough to create a table of the twiddle factors. Since the twiddle factor is calculated from two loop parameters, the code motion will never lift it outside the loops. The reason for this is that it would require duplicating the loop nest and allocating memory to store the results. However, the programmer might want to make the trade-off between space (more code and memory) and execution time, and Feldspar provides the tools for this.

To achieve a shared table we must forbid in-lining and explicitly share the vector as in `dftTabled`.

```
36  dftTabled v = share (twids (length v)) $ λws → dft' ws v
```

The resulting code can be seen in listing 1.4.

1.5.3 Cooley-Tukey Fast Fourier Transform

The complexity of the Fourier Transform implementation can be reduced to $\mathcal{O}(N \log N)$ by using a Fast Fourier Transform (FFT) which exploits the symmetries in the computations. One of the most used FFTs is the Cooley-Tukey algorithm [12].

In Feldspar we can implement the Cooley-Tukey Radix-2 as in listing 1.5 (with some helper function from listing 1.7 on page 29). The implementation is inspired by the pipelined FFT circuit in reference [6].

```
37 ghci> icompile' defaultOptions{rules=nativeArrayRules} "dftTabled" (dftTabled -::
       newLen 8 >-> id)
38 static double complex const v8[8] = {(1.0f+-0.0fi),
39                                      (0.7071067811865476f+-0.7071067811865475fi),
40                                      (6.123233995736766e-17f+-1.0fi),
41                                      (-0.7071067811865475f+-0.7071067811865476fi),
42                                      (-1.0f+-1.2246467991473532e-16fi),
43                                      (-0.7071067811865477f+0.7071067811865475fi),
44                                      (-1.8369701987210297e-16f+1.0fi),
45                                      (0.7071067811865475f+0.7071067811865477fi)};
46
47 void dftTabled(double complex v0[], double complex out[8])
48 {
49   double complex e0;
50
51   assert(out);
52   for (uint32_t v9 = 0; v9 < 8; v9 += 1)
53   {
54     e0 = (0.0f+0.0fi);
55     for (uint32_t v10 = 0; v10 < 8; v10 += 1)
56     {
57       e0 = (e0 + (v0[v10] * v8[((v9 * v10) % 8)]));
58     }
59     out[v9] = e0;
60   }
61 }
```

Listing 1.4: DFT_8 with shared table of twiddle factors

While short, this implementation warrants a more detailed description.

First, let's start with a picture of the algorithm. Figure 1.2 shows the data flow network for the `fft` on a vector of eight elements. The three ($\log_2 8$) stages show the successive decomposition using the `chunk` combinator. The `chunk` combinator is a divide and conquer algorithm that divides the input vector into smaller parts, applies a function to each part, and reassembles the results.

The `dft2` function simply calculates the DFT_2 for two elements and a twiddle factor. The first application of `dft2` in each stage is highlighted in the figure.

More interestingly, the `butterfly` combinator helps us decompose the algorithm. First, using `halve :: Vector a → Vector (a,a)`, it divides the input vector in two and regroups the elements by pairing them element-wise. This `Vector` of pairs is combined with the twiddle factors. Finally, `unhalve :: Vector (a,a) → Vector a` redistributes the elements to form the result `Vector`.

The high-level description generates efficient code (see listing 1.9 on page 30). The outer loop takes $\mathcal{O}(\log N)$ iterations while the inner loop takes $\mathcal{O}(N)$ iterations.

However, due to technical limitations in the Vector library the inner loop includes

```
40 dft2 :: Num a => a → (a,a) → (a,a)
41 dft2 w (x0,x1) = (x0+x1, (x0-x1)*w)
42
43 butterfly :: (Syntax a)
44           => (a → (a,a) → (a,a))
45           → Vector a → Vector a → Vector a
46 butterfly f ws = uncurry (++) . unzip
47                . zipWith f ws
48                . uncurry zip . halve
49
50 fft :: (Syntax a, Num a)
51     => Vector a → Vector a → Vector a
52 fft ws vs = forLoop (ilog2 len) vs stage
53   where
54     len     = length vs
55     stage s = withLen len
56             $ chunk (1 .<<. s) (len .>>. s) (butterfly dft2 (ixmap (.<<. s) ws))
```

Listing 1.5: Cooley-Tukey Radix-2 Decimation in Frequency FFT

a condition, which may interfere with optimizations.

Note that the figure 1.2 is generated from the same source as the C code in listing 1.9. To achive this the element type is set to a list recording the element indices that contribute to each node. A small Haskell script takes the list and converts it into Ti*k*Z commands which are rendered by LaTeXwhen typesetting the thesis.

1.5.4 Push Vector FFT

The implementation in section 1.5.3 iterates over the entire vector and reads each element twice and uses a condition to decide which calculation to make. This access pattern is wasteful as it needlessly increases the used memory bandwidth. The

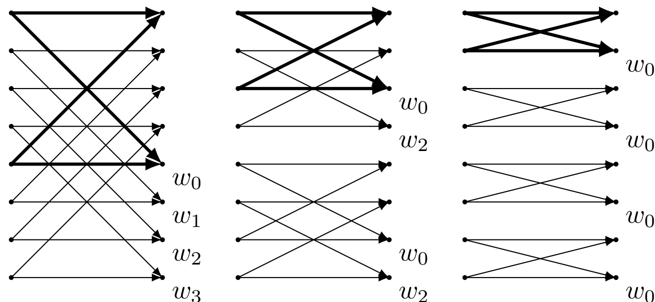

Figure 1.2: Butterfly network of an eight-point FFT

```
50 dft2 :: Num a => a → (a, a) → (a,a)
51 dft2 w (x0,x1) = (x0+x1, (x0-x1)*w)
52
53 butterfly :: (Syntax a, Num a)
54           => (a → (a,a) → (a,a))
55           → Vector a → Vector a → PushVector a
56 butterfly f ws = unhalve . zipWith f ws . uncurry zip . halve
57
58 -- | Cooley-Tukey Radix-2 Decimation In Frequency Fast Fourier Transfrom
59 fft :: (Syntax a, Num a)
60     => Vector a → Vector a → Vector a
61 fft ws vs = forLoop (ilog2 len) vs stage
62   where
63     len = length vs
64     stage s = withLen len
65             $ freezeToVector
66             . chnk (1 .<<. s) (len .>>. s) (butterfly dft2 (ixmap (.<<. s) ws))
```

Listing 1.6: Cooley-Tukey Radix-2 Decimation in Frequency FFT (Push)

reason is that the `Vector` type does not capture an evaluation order, instead it is up to the consumer of the `Vector`.

With the help of `PushVector` (section 1.3.2) we can control the evaluation order to produce and write both elements of the `dft2` in one iteration, see listing 1.6. The skeleton of the algorithm remains the same. Only the implementations of the combinators `halve`, `unhalve`, and `chunk` have changed, see listing 1.8.

In the generated code (listing 1.10 on page 31), the inner loop writes two elements in each iteration. The middle loop runs at most for 512 iterations (`0 =< v25 < 10; v35 = 1 << v25`).

With a quick test we check that the `fft` still has the same semantics as the `dft`. Note that the `fft` leaves the data in *bit-reversed order*, which is why we need to rearrange the data with `bitRev`.

```
1 ghci> quickCheck $ forAll (vectorOf 16 arbitrary) $ λps →
2     let xs = fmap Data.Complex.cis ps
3     in (fmap cround $ eval dft xs) === (fmap cround $ eval (bitRev . fft (twids
      16)) xs)
4 +++ OK, passed 100 tests.
```

```
15 ixmap :: (Syntax a)
16       => (Data Index → Data Index) → Vector a → Vector a
17 ixmap f = permute (const f)
18
19 halve :: (Syntax a) => Vector a → (Vector a, Vector a)
20 halve v = splitAt (length v `div` 2) v
21
22 chunk :: (Syntax a, Syntax b)
23       => Data Length → Data Length → (Vector a → Vector b) → Vector a →
      Vector b
24 chunk r c f v = flatten $ map f $ split v
25   where
26     l       = length v
27     split w = indexed r $ λi →
28                 indexed c $ λj → w ! (c*i+j)
29
30     flatten :: Syntax a => Vector (Vector a) → Vector a
31     flatten arr = indexed l ixf
32       where
33         ixf i = arr ! y ! x
34           where
35             y = i `div` c
36             x = i `mod` c
```

Listing 1.7: Helper functions for working with chunked Vectors

```
26 chnk :: (Pushy arr1, Syntax b)
27       => Data Length          -- ^ Number of chunks
28       → Data Length           -- ^ Size of the chunks
29       → (Vector a → arr1 b) -- ^ Applied to every chunk
30       → Vector a
31       → PushVector b
32 chnk r c f v = Push loop $ length v
33   where loop func = forM r $ λi →
34                       do let (Push k _) = toPush $ f (take c (drop (c*i) v))
35                          k (λj a → func (c*i + j) a)
36
37 unhalve :: (Syntax a)
38         => Vector (a,a) → PushVector a
39 unhalve xs = unpairWith (stride 1 (length xs)) xs
40
41 stride :: Data Length → Data Length
42        → (Data Index → a → M b)
43        → Data Index → (a,a) → M b
44 stride n k f ix (a1,a2) = f (n*ix) a1 >> f (n*ix+k) a2
```

Listing 1.8: Helper functions for working with chunked Push-vectors

```
1  ghci> icompile' defaultOptions{rules=nativeArrayRules} "fft" (fft -:: tVec1 (
       tComplex tDouble) . newLen 512 >-> newLen 1024 >-> id)
2  void fft(double complex v0[], double complex v1[], double complex out[1024])
3  {
4    uint32_t v21;
5    uint32_t v23;
6    uint32_t v22;
7    uint32_t v24;
8    uint32_t v25;
9    uint32_t v26;
10   uint32_t v27;
11
12   assert(out);
13   copyArray(out, v1);
14   out = setLength(out, sizeof(double complex), 1024);
15   for (uint32_t v18 = 0; v18 < 10; v18 += 1)
16   {
17     v21 = (1024 >> v18);
18     v22 = min(v21, (512 >> v18));
19     v23 = min(v22, (v21 - v22));
20     v24 = (512 >> v18);
21     assert(out);
22     for (uint32_t v20 = 0; v20 < 1024; v20 += 1)
23     {
24       v25 = (v20 % v21);
25       v26 = (v21 * (v20 / v21));
26       v27 = (v25 - v23);
27       if ((v25 < v23))
28       {
29         out[v20] = (out[(v26 + v25)] + out[(v26 + (v25 + v24))]);
30       }
31       else
32       {
33         out[v20] = ((out[(v26 + v27)] - out[(v26 + (v27 + v24))]) * v0[(v27 << v18
       )]);
34       }
35     }
36   }
37 }
```

Listing 1.9: FFT_{1024}

```
5  ghci> icompile' defaultOptions{rules=nativeArrayRules} "fft" (fft -:: tVec1 (
       tComplex tDouble) . newLen 512 >-> newLen 1024 >-> id)
6  void fft(double complex v0[], double complex v1[], double complex out[1024])
7  {
8    uint32_t v35;
9    uint32_t v36;
10   uint32_t v38;
11   uint32_t v37;
12   uint32_t v39;
13   uint32_t v40;
14   double complex v41;
15   double complex v42;
16 
17   assert(out);
18   copyArray(out, v1);
19   out = setLength(out, sizeof(double complex), 1024);
20   for (uint32_t v25 = 0; v25 < 10; v25 += 1)
21   {
22     v35 = (1 << v25);
23     v36 = (1024 >> v25);
24     out = setLength(out, sizeof(double complex), 1024);
25     for (uint32_t v29 = 0; v29 < v35; v29 += 1)
26     {
27       v37 = min((1024 - min(1024, (v36 * v29))), v36);
28       v38 = min((v37 >> 1), ((v37 + 1) >> 1));
29       v39 = (v36 * v29);
30       v40 = (min((1024 - min(1024, v39)), v36) >> 1);
31       for (uint32_t v30 = 0; v30 < v38; v30 += 1)
32       {
33         v41 = out[(v30 + v39)];
34         v42 = out[((v30 + v40) + v39)];
35         out[(v39 + v30)] = (v41 + v42);
36         out[(v39 + (v30 + v38))] = ((v41 - v42) * v0[(v30 << v25)]);
37       }
38     }
39   }
40 }
```

Listing 1.10: FFT_{1024} Push

1.5.5 Benchmark

The `criterion` Haskell library provides a framework for running and analysing statistically robust benchmarks. With the help of the technique from section 1.4, the `criterion` library can be used to analyse the performance of `Feldspar` programs. The benchmark suite code can be found in listing 1.11.

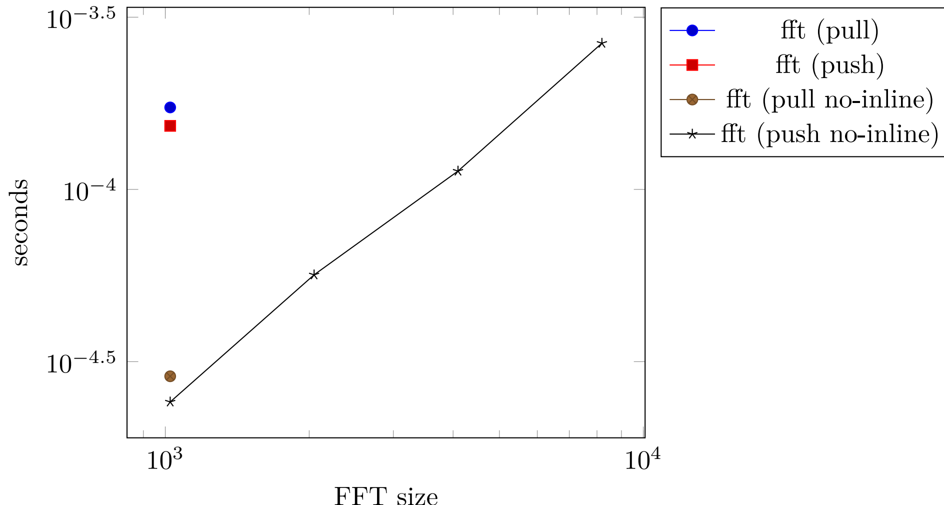

Figure 1.3: FFT Performance

The benchmarks were run on a single core on a 2GHz Intel Core i7. The implementations in this paper can not compete with best in class implementatios on that platform, since our generated code does not use any vector instructions. However, an analysis internally at Ericsson, show that on single issue hardware our implementation has the same number of instructions per bit as a handwritten FFT.

```
23 fftPull    vs = FFT.Pull.fft (twids 1024) (newLen 1024 vs)
24 fftPush    vs = FFT.Push.fft (twids 1024) (newLen 1024 vs)
25 fftPullT   vs = share (twids 1024) $ λws → FFT.Pull.fft ws (newLen 1024 vs)
26 fftPushT   vs = share (twids 1024) $ λws → FFT.Push.fft ws (newLen 1024 vs)
27 fftPullNI  vs = FFT.Pull.fft (noInline (twids 1024)) (newLen 1024 vs)
28 fftPushNI  vs = FFT.Push.fft (noInline (twids 1024)) (newLen 1024 vs)
29 fftPushNI2 vs = FFT.Push.fft (noInline (twids 2048)) (newLen 2048 vs)
30 fftPushNI4 vs = FFT.Push.fft (noInline (twids 4096)) (newLen 4096 vs)
31 fftPushNI8 vs = FFT.Push.fft (noInline (twids 8192)) (newLen 8192 vs)
32
33 $(loadFunOpts ["-optc=-O2"] [ 'fftPull, 'fftPush, 'fftPullT, 'fftPullNI
34                             , 'fftPushT, 'fftPushNI, 'fftPushNI2, 'fftPushNI4
35                             , 'fftPushNI8
36                             ])
37
38 testdata :: Length → [Complex Double]
39 testdata len = P.replicate l2 1 P.++ P.replicate l2 0
40   where l2 = P.fromIntegral $ P.div len 2
41
42 myConfig = Criterion.Config.defaultConfig { cfgPerformGC   = ljust True
43                                           , cfgReport      = ljust "report.html"
44                                           , cfgSummaryFile = ljust "summary.csv"
45                                           }
46
47 main :: IO ()
48 main = with def $ λout → do
49     -- pack data before calling the compiled function
50     let pck l = pack (testdata l) P. »= evaluate
51     [real1, real2, real4, real8] ← P.mapM pck [1024,2048,4096,8192]
52     -- compile and load all functions before benchmarking
53     P.mapM_ evaluate [ c_fftPull_builder, c_fftPush_builder
54                      , c_fftPullT_builder, c_fftPushT_builder
55                      , c_fftPullNI_builder, c_fftPushNI_builder
56                      , c_fftPushNI2_builder, c_fftPushNI4_builder
57                      , c_fftPushNI8_builder
58                      ]
59     -- benchmark suite
60     defaultMainWith myConfig (return ())
61       [ bench "FFT (pull) 1024"          $ c_fftPull_raw    real1 out
62       , bench "FFT (push) 1024"          $ c_fftPush_raw    real1 out
63       , bench "FFT (pull-tabled)   1024" $ c_fftPullT_raw   real1 out
64       , bench "FFT (pull-noinline) 1024" $ c_fftPullNI_raw  real1 out
65       , bench "FFT (push-tabled)   1024" $ c_fftPushT_raw   real1 out
66       , bgroup "FFT (push-noinline)"
67         [ bench "1024" $ c_fftPushNI_raw  real1 out
68         , bench "2048" $ c_fftPushNI2_raw real2 out
69         , bench "4096" $ c_fftPushNI4_raw real4 out
70         , bench "8192" $ c_fftPushNI8_raw real8 out
71         ]
72       ]
```

Listing 1.11: FFT Benchmark suite

1.6 Design Flow of a Feldspar function

The examples in the case study show how we handle the aspects from section 1.1.1; Functionality (1), Architecture (2), Parallelism (3) and Data Size (9). To design a Feldspar function we first start with a functional specification. The specification can be expressed either as a function (in Feldspar or Haskell) or as a set of properties that the function should have. The specification acts as our safety net when exploring the design space of the function.

In the case study we created four variations on the Fourier transform and they illustrate the different tradeoffs necessary.

Name	Complexity	Memory	Flexibility
dft	N^2	N	any size
dft (tabled)	N^2	$2N$	fixed size
fft	$N \log N$	$1.5N$	fixed size, power of two
fft push	$N \log N$	$1.5N$	fixed size, power of two

The `dft` while slow is the most flexible implementation. The generated code can handle input data of any length.

By precalculating the twiddle factors we gain a lot of speed, at the cost of flexibility and memory size. The table requires as much memory as the input vector.

The `fft`s have, due to a more efficient algorithm, much better performance. They suffer from the same inflexibility as the `dft`, but the memory requirement is lower since the algorithm utilizes the symmetry around the real axis when tabling the twiddle factors.

The `fft`s are built from the combinators `chunk` and `butterfly`, which in turn are built from combinators from the `Vector` library. Even though these combinators are specified outside the `Vector` library, the fusion guarantees still hold. Since the combinators and functions are polymorphic, it is even possible to generate visualisations (figure 1.2) from the same code. The combinators are highly reusable and are in fact the first parts of a small DSEL for expressing algorithms such as `fft` and bitonic mergers.

1.7 Related Work

The MathWorks Matlab Coder can generate C and C++ code from a subset of Matlab features. Matlab programs are imperative and not compositional, unless

they can be expressed as matrix operations.

Cryptol[2] [18] from Galois, Inc is a DSL for specification of cryptographic algorithms. Cryptol compilers can generate C and C++ software implementations and Verilog/VHDL hardware implementations from the same description. While not an embedded language, Cryptol shares many of its design philosophies with Feldspar. An important part is the ability to do rapid development, by running and testing the specification before compilation to software or hardware. But, since it is not embedded, all tools have to be developed and cannot benefit from the ecosystem of tools available in a host language.

The Spiral project[3] uses a DSL [24] to express decompositions of transforms, and then searches for the best composition for a given context, permitting the generation of high performance library functions. Even though we do not (yet) use search in Feldspar, decompositions similar to those in Spiral, combined with our fusion technology, enable elegant descriptions of transforms that generate compact C implementations. The FFT examples (listings 1.5 and 1.6 on pages 27 and 28), illustrate this, showing how the intermediate arrays are fused away.

Single-Assignment C (SAC)[4] [25] is an array programming language for numerically intensive applications, including signal processing. SAC has an imperative syntax quite similar to C. But, internally the syntax is interpreted as a functional program, hence the single assignment part of the name. The type system of SAC supports what are called *shapely* types. This means that it is possible to write programs that are shape polymorphic. In Feldspar, the `Feldspar.Repa` module supports shape polymorphic arrays [16].

In SAC, the programmer specifies the content of an array using a generator. For example, the following code

```
1 with
2   ([1,1] ≤ iv < [3,3]): a[iv]
3   default: 0
4 genarray([4,4)
```

corresponds to the following array

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & a_{1,1} & a_{1,2} & 0 \\ 0 & a_{2,1} & a_{2,2} & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{1.3}$$

The elements $a_{x,y}$ are drawn from a previously defined array, for indices selected by the first rule of the `with` clause.

[2] www.cryptol.net
[3] www.spiral.net
[4] www.sac-home.org

The programming experience is rather different and in particular less modular than in Feldspar.

Obsidian [29] is a embedded language for array programming on the GPU (Graphics Programming Unit). Obsidian, like Feldspar, uses both pull and push arrays [11] to allow the programmer to structure computations. It gently but firmly restricts what programs can be writte, via the type system, in order to be able to generate good code for the GPU, which can only deal with computations structured in a particular way. This makes Obsidian into much more of a straitjacket than Feldspar is.

Accelerate [8] combines a deep embedding with algorithmic skeletons. Common patterns of array computation (like map and fold) are JIT compiled into individual CUDA kernels, with the actual worker functions spliced into previously defined CUDA templates. This can give good speedups for Haskell users who do not wish to become familiar with the details of low level GPU programming. The downside is that the approach may result in too many kernel calls and too much data transfer between host and GPU. Recent work [20] on Accelerate has concentrated on the addition of fusion, giving fewer kernel calls and a considerable improvement in performance.

Ivory and Tower are two other languages from Galois. Ivory is a langaue for embedded systems, with provable memory and type safety. Tower, like our future system layer, is a language to compose Ivory programs into real-time tasks. The languages were developed as part of the SMACCMPilot[5] project. Both languages will serve as great inspiration for future work on Feldspar.

1.8 Discussion

Performance, measured in cycles, bytes or watts, remains the most important aspect to consider when designing embedded software. To achieve this performance, developers optimize their applications by taking advantage of the low-level details of the processor execution pipeline and memory subsystem. To get access to the low-level details, the programs are expressed using equally low-level imperative programming languages, such as C or assembler. Since the languages are imperative and allow implicit side effects, the programs are hard to analyse and optimize.

However, these low-level languages provide little support for abstraction, generalization and modularity.

It is difficult to provide abstractions when types are limited to scalars, structures and pointers. The lack of parametric polymorphism forces us to either write monomorphic programs or throw away most type information to write generic and reusable functions.

[5] http://smaccmpilot.org

Adding parallelism and concurrency to a program can be very difficult.

The Feldspar language and compiler presented in this thesis, are work in progress to create a language that allows modular, generic and portable implementation of embedded software, while still providing predictable performance.

The language is modular in the perspectives of both the implementer and the user. For the language implementer, the use of a mixed shallow and deep embedding approach (section 1.2.4), makes it easy to extend the language with new features and interpretations. For example, support for monadic expressions was added without changing the pure parts of the language and compiler. The monadic expressions now serve as a foundation for mutable expressions and deterministic parallelism. More information on the monadic extension can be found in paper C. While powerful, the monadic interface for mutable arrays is a low-level interface with explicit indexing. Similar to the parallel arrays, we have created a library of convenient combinators in the `PushVector` library.

From the perspective of the language user, it is possible to write modular and generic code, through the use of costless abstractions. For example, in the case study (section 1.5), the `butterfly` structure of the FFT was constructed using combinators and functions from the vector library. Since the vector library guarantees fusion of vector operations, the use of combinators neither incurs any extra storage of intermediate data structures, nor any extra traversals.

1.9 Future Work

Feldspar version 0.7 is focused on the functional specification of algorithms. However, the language and compiler needs improvements in certain areas.

Several of the decision aspects in section 1.1.1 are not possible to express. Work is ongoing to add support for SIMD parallelism and to add other backends (e.g. LLVM). The memory allocation will be augmented with a simplified region inference.

But the majority of future work lies in adding a "system layer" to Feldspar. The system layer provides a means of partitioning, distributing and orchestrating Feldspar kernels on heterogenous multi-core devices.

1.10 Conclusion

Currently, Feldspar is focused on the data processing part of signal processing in embedded systems. The Feldspar compiler generates single C functions that can be integrated into existing systems. The Feldspar language is based on a combination of shallow and deep embedding, which makes it modular both for the

language developer and the end user. This modularity made it easy to add monadic expressions to support mutable updates and deterministic parallelism.

Feldspar is designed to be a language with predictable performance. Because of fusion, the loop structure of code generated from Feldspar is satisfactory and it is easy for the programmer to influence the shape of the resulting code. To support the programmer in testing and benchmarking, we have provided the means to integrate the generated C code back into the host language environment. The main remaining deficiency in the generated code is excessive copying which can happen, for example, when tuples are used as state in loops.

Future work includes avoiding the excessive copying, as well as generation, deployment and orchestration of systems of data processing functions and control processing.

Appendix A

Terminology

This appendix introduces concepts that may be unfamiliar to readers without Haskell experience.

A.1 Parametric Polymorphism

In a strongly typed language like Haskell, one might expect it to be difficult to express *generic* functions that will handle values identically without depending on their type.

For example, a function that applies a function to each element in a list might be implemented like this:

```
1  mapInt :: (Int → Int) → [Int] → [Int]
2  mapInt _ []     = []
3  mapInt f (x:xs) = f x : mapInt f xs
```

However, this is a construct that would be useful for many other types of lists and it is tedious to reimplement the construct over and over again. We note that the `mapInt` function preserves the structure of the list; in both clauses, the list is reconstructed in the right hand side in the same shape as was de-constructed in the left side.

This indicates that the `mapInt` function could be used for lists of any type. In Haskell, we use *parametric polymorphism* [27] to allow a strongly typed function to operate on partially known types.

A parametrically polymorphic implementation of our mapping function looks like this:

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

The function `map` will traverse the structure of the list and apply the function `f` to each element, regardless of their type. It is worth noting that, since we are constructing a new list and not modifying the input list, we can also allow the function `f` to change the type of the contained elements.

We also note a thing about Haskell type signatures. The type signature `a → b` is shorthand for `forall a b. a → b`. The `forall` here specifies that the types `a` and `b` are universally quantified (∀) and cannot be inspected.

The concept can be generalized even further by recognizing that a list is also a `Functor`. Members of the `Functor` type class (see section A.4) are types that can be mapped over.

A.2 Algebraic Data Type

An Algebraic Data Type (ADT) is a composite type formed by combining other types. Common classes of ADTs include *product types* (tuples and records) and *sum types* (tagged unions or variants).

Values of product types consist of fields, of possibly different types, where all fields are available at the same time.

Values of a sum type contain variants of the union type.

As an example of an algebraic data type we consider the singly linked list. A `List` type is a sum type with two variants; `Nil` represents an empty `List`, and `Cons` builds a `List` by pre-pending an element to an existing list. The vertical bar `|` separates the constructor variants.

```
data List a = Nil | Cons a (List a)
```

The value of an algebraic data type is analysed using pattern matching on the constructors. We can calculate the length of a list by counting the number of elements.

```
length :: List a → Int
length Nil         = 0
length (Cons _ xs) = 1 + length xs
```

A.3 Generalized Algebraic Data Type

Algebraic data types (section A.2) give us the means to construct composite types from smaller components. However, it is still possible to write functions over these type that are non-total. Consider the function that returns the first element of a `List`.

```
1 head :: List a → a
2 head Nil        = error "Empty list"
3 head (Cons x _) = x
```

In the case of an empty `List`, we have no value to return and we cannot produce a default value since we have to be parametric in the type `a` (section A.1). This error can not be caught by the type checker since both `Nil` and `Cons` are valid values for `List`, leaving us vulnerable to a runtime error occurring when we apply `head` to an empty list.

With Generalized Algebraic Data Types (GADT) [23, 1, 9, 33, 26] we can create an algebraic data type where the constructors can return different types.

Now we can define a safe variant of our `List`.

```
1 data Empty
2 data NonEmpty
3
4 data SafeList a b where
5   Nil  :: SafeList a Empty
6   Cons :: a → SafeList a b → SafeList a NonEmpty
```

Here, `Empty` and `NonEmpty` are phantom (non-inhabited) types. By specializing the `head` function to only accept non-empty lists we can make it safe.

```
1 safeHead :: SafeList a NonEmpty → a
2 safeHead (Cons x _) = x
```

We do not have to have a case for `Nil` since the only way to construct a `SafeList a NonEmpty` is to use the `Cons` constructor.

A.4 Type Class

The purpose of the type class type system construct [31] is to support *ad-hoc* polymorphism. The class acts as a constraint on a type variable in a parametrically polymorphic type.

```
1 class Eq a where
2   (==) :: a → a → Bool
```

The class declaration above introduces the class `Eq` with the operator `(==)` representing the equality between two values of the type `a`. The class only represents the concept of equality; to actually implement it, the programmer defines an instance for a specific type.

```
instance Eq Int where
  (==) = eqInt
```

To be able to use the `(==)` operator, the programmer must assert to the compiler that any type used in fact implements the operator. This is done by using a concrete type that implements the operator, or by *constrained parametric polymorphism*.

In Haskell we can do this by specifying one or more constraints before the type signature:

```
(... constraint, constraint ...) => type signature
```

where a constraint must mention the class and the constrained type variable. For example, equality can be extended to arbitrary lists of any type `a`, provided that `a` supports equality.

```
instance (Eq a) => Eq [a] where
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
  _      == _      = False
```

The first clause of the definition recursively evaluates if two lists are equal by first checking that the heads of the lists are equal and second checking that the tails are equal. Logical `and` (written `&&`) is used to combine the Boolean expressions. Note that, in the first clause, the middle `==` uses the `Eq a` variant, while the rightmost `==` uses the `Eq [a]` variant. The second clause will catch any remaining cases where the structures of the lists differ.

Continuing our example from section A.1, we can introduce the `Functor` type class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

This class generalizes the concept of mapping a function `a → b` over a list `[a]` to any compatible container or context.

The list can naturally be made a `Functor` instance.

```
instance Functor [] where
  fmap = map
```

The contained type `a` is not even mentioned in the instance, which should make it clear that `fmap` is only concerned with mapping the function over the structure.

References

[1] Lennart Augustsson and Kent Petersson. *Silly Type Families * DRAFT*. 1994 (cit. on p. 41).

[2] Emil Axelsson. “A generic abstract syntax model for embedded languages”. In: *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*. ICFP ’12. ACM, 2012, pp. 323–334 (cit. on p. 11).

[3] Emil Axelsson, Koen Claessen, and Mary Sheeran. “Wired: Wire-aware circuit design”. In: *Correct Hardware Design and Verification Methods*. Vol. 3725. Lecture Notes in Computer Science. Springer, 2005, pp. 5–19 (cit. on p. 10).

[4] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. “The Design and Implementation of Feldspar – an Embedded Language for Digital Signal Processing”. In: *22nd International Symposium, IFL 2010*. Vol. 6647. LNCS. 2011 (cit. on pp. 1, 17).

[5] Emil Axelsson, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András 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 Computer Society, 2010 (cit. on p. 1).

[6] Per Bjesse. “Automatic verification of combinational and pipelined FFT circuits”. In: *Computer Aided Verification*. Springer. 1999, pp. 380–393 (cit. on p. 25).

[7] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. “Lava: Hardware Design in Haskell”. In: *In International Conference on Functional Programming*. 1998 (cit. on p. 10).

[8] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. “Accelerating Haskell array codes with multicore GPUs”. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14 (cit. on p. 36).

[9] James Cheney and Ralf Hinze. *First-class phantom types*. Tech. rep. Cornell University, 2003 (cit. on p. 41).

[10] Koen Claessen and John Hughes. “QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs”. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP ’00. New York, NY, USA: ACM, 2000, pp. 268–279. ISBN: 1-58113-202-6. DOI: 10.1145/351240.351266. URL: http://doi.acm.org/10.1145/351240.351266 (cit. on pp. 10, 16, 24).

[11] Koen Claessen, Mary Sheeran, and Bo Joel Svensson. “Expressive array constructs in an embedded GPU kernel programming language”. In: *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*. ACM. 2012, pp. 21–30 (cit. on pp. 19, 36).

[12] James W Cooley and John W Tukey. “An algorithm for the machine calculation of complex Fourier series”. In: *Mathematics of computation* 19.90 (1965), pp. 297–301 (cit. on p. 25).

[13] Conal Elliott, Sigbjørn Finne, and Oege de Moor. “Compiling embedded languages”. In: *Journal of Functional Programming* 13.3 (2003), pp. 455–481 (cit. on pp. 9, 10, 15, 18).

[14] Paul Hudak. “Modular domain specific languages and tools”. In: *Software Reuse, 1998. Proceedings. Fifth International Conference on*. IEEE. 1998, pp. 134–142 (cit. on pp. 9, 10).

[15] John Hughes. “Why functional programming matters”. In: *The computer journal* 32.2 (1989), pp. 98–107 (cit. on p. 8).

[16] Gabriele Keller, Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. “Regular, shape-polymorphic, parallel arrays in Haskell”. In: *ACM Sigplan Notices*. Vol. 45. 9. ACM. 2010, pp. 261–272 (cit. on p. 35).

[17] Donald E Knuth. “Structured Programming with go to Statements”. In: *ACM Computing Surveys (CSUR)* 6.4 (1974), pp. 261–301 (cit. on p. 7).

[18] Jeffrey R Lewis and Brad Martin. “Cryptol: High assurance, retargetable crypto development and validation”. In: *Military Communications Conference, 2003. MILCOM’03. 2003 IEEE*. Vol. 2. IEEE. 2003, pp. 820–825 (cit. on p. 35).

[19] Simon Marlow, Ryan Newton, and Simon Peyton Jones. “A monad for deterministic parallelism”. In: *ACM SIGPLAN Notices*. Vol. 46. 12. ACM. 2011, pp. 71–82 (cit. on p. 18).

[20] Trevor L McDonell, Manuel MT Chakravarty, Gabriele Keller, and Ben Lippmeier. “Optimising purely functional GPU programs”. In: *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*. ACM. 2013, pp. 49–60 (cit. on p. 36).

[21] Anders Persson, Emil Axelsson, and Josef Svenningsson. “Generic monadic constructs for embedded languages”. In: *23rd International Symposium on Implementation and Application of Functional Languages, IFL 2011*. Vol. 7257. 2011 (cit. on pp. 1, 18).

[22] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003 (cit. on p. 10).

[23] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. “Simple unification-based type inference for GADTs”. In: *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional programming (ICFP)*. Portland, Oregon, USA: ACM, 2006, pp. 50–61 (cit. on p. 41).

[24] Markus Püschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. “SPIRAL: Code generation for DSP transforms”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 232–275 (cit. on p. 35).

[25] Sven-Bodo Scholz. “Single Assignment C: efficient support for high-level array operations in a functional setting”. In: *Journal of Functional Programming* 13.6 (2003), pp. 1005–1059 (cit. on p. 35).

[26] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. “Complete and decidable type inference for GADTs”. In: *Proc. 14th ACM SIGPLAN international conference on Functional programming*. Edinburgh, Scotland: ACM, 2009, pp. 341–352 (cit. on p. 41).

[27] Christopher Strachey. “Fundamental concepts in programming languages”. In: *Higher-order and symbolic computation* 13.1-2 (2000), pp. 11–49 (cit. on p. 39).

[28] Josef Svenningsson and Emil Axelsson. “Combining Deep and Shallow Embedding for EDSL”. In: *Trends in functional programming*. Vol. 7829. LNCS. Springer-Verlag. 2013 (cit. on p. 11).

[29] Joel Svensson, Mary Sheeran, and Koen Claessen. “Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors”. In: *Implementation and Application of Functional Languages 2008*. Vol. 5836. Lecture Notes in Computer Science. Springer. 2011, pp. 156–173 (cit. on pp. 10, 36).

[30] Philip Wadler. “Comprehending monads”. In: *Mathematical Structures in Computer Science* 2.04 (1992), pp. 461–493 (cit. on p. 18).

[31] Philip Wadler and Stephen Blott. “How to make ad-hoc polymorphism less ad hoc”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1989, pp. 60–76 (cit. on p. 41).

[32] Arie Van Deursen, Paul Klint, and Joost Visser. “Domain-Specific Languages: An Annotated Bibliography.” In: *Sigplan Notices* 35.6 (2000), pp. 26–36 (cit. on p. 9).

[33] Hongwei Xi, Chiyan Chen, and Gang Chen. “Guarded recursive datatype constructors”. In: *ACM SIGPLAN Notices*. Vol. 38. 1. ACM. 2003, pp. 224–235 (cit. on p. 41).