Program Transformation and Derivation for the Lift Language

Piotr Jander

MInf Project (Part 1) Report
Master of Informatics
School of Informatics
University of Edinburgh

2018

Abstract

The research area of program derivation delivered theory and examples showing how efficient programs can be calculated from their specification. However, widely used tools today have no direct support for complex program transformation which depend on the algebraic properties of the program.

This project aims to implement Pointfree Lift, a system which combines the functionality of a functional programming language with capabilities of program transformation. The correctness and effectiveness of the implementation are demonstrated with several examples, including a novel derivation involving matrix-vector multiplication.

An interactive user interface for program transformation is included, and the performance gains from program transformation are measured in benchmarks. The benchmarks confirm theoretically predicted gains from the transformations described in this report.

Table of Contents

Chapter 1

Introduction

This project explores a number of techniques for transforming programs with rewrite rules. Rewrite rules are widely used in functional programming, where rich semantics and declarative style allows for sophisticated program transformation. In particular, rewrite rules are used for program derivation, that is, the process of obtaining one program from another in a series of steps, where a step is justified by an application of a rewrite rule.

Program derivation can be done by hand on a piece of paper. However, as part of this project, a framework named Pointfree Lift was implemented for applying rewrite rules and performing derivations. It simplifies bookkeeping and ensures that no errors are made when using rules. It has scripting capabilities, as well as an interactive user interface.

Although this project only implements a simple interpreter, one can imagine that Pointfree Lift could even compile derived programs to a real-world language like Lift IR. Then the user could write a program specification, then transform it to obtain a suitable and efficient implementation, and then have it compiled in one go. Presumably, such development workflow could save the programmer from errors resulting from writing the program in an ad hoc manner.

Pointfree Lift's capabilities are demonstrated by performing a number of well-known derivations, including a derivation of an efficient implementation of the maximum segment sum problem [12]. An original contribution of this project is a derivation of a program for matrix-vector multiplication where the matrix is in a sparse format. Pointfree Lift can perform a derivation for the CSR format, for example.

Capabilities of Pointfree Lift are, of course, limited. Being based on rewrite rules, it cannot take advantage of the categorical approach to program derivation [7]. For this reason, some derivations included in this work will be discussed without actually implementing them in Pointfree Lift, as is the case with the program for matrix-vector multiplication using the BSR format.

The name Pointfree Lift is inspired by the Lift project. The Lift project is a system which compiles high-level functional programs to OpenCL. Its ability to produce efficient OpenCL code is possible in part thanks to using rewrite rules. However, the current implementation of rewrite rules has limited expressivity, and complex syntax of Lift IR makes it difficult to implement new rules.

Pointfree Lift was born out of the fact that Lift IR, and its rewrite rules, is too rigid and unwieldy for program derivation. Consequently, Pointfree Lift is both similar to and different from Lift IR. Like Lift IR, Pointfree Lift relies on functional patterns like map and reduce. Unlike Lift IR, Pointfree Lift has minimalistic syntax based on function application and composition, which facilitates writing highly expressive rewrite rules.

When this report discusses examples of functional code, Pointfree Lift is pretty-printed to resemble standard Haskell syntax, as well as GHC [3] syntax for rewrite rules.

1.1 Motivation

To make the rest of the report more concrete, this section presents some use cases for rewrite rules.

The first use of rewrite rules is to express optimisations. For example, the rewrite rule below states that mapping a function un-distributes through composition.

$$\forall f, g.\ map\ f\ (map\ g\ v) = map\ (f \circ g)\ v$$

The reader should be able to convince himself as to the correctness of this rule.

Rewrite rules for optimisations When rewrite rules are used to describe optimisations, it is implied that the right-hand side is preferable to the left-hand side. Indeed, the right side applies f and g to every element of the argument list at the same time, and in contrast to the left side, the right side avoids creating an intermediate data structure.

In fact, Lift IR already support this and many other simple rewrite rules as means for optimisation. When it comes to expressing simple rewrite rules, Pointfree Lift is only superior to Lift IR in that rewrite rules are expressed more succinctly.

Rewrite rules for algebraic properties However, Pointfree Lift supports more complex rewrite rules, which will be needed for more interesting experiments in this paper. Another use case for rewrite rules is equational reasoning about functional programs, also referred to as program derivation. For instance, the rule below states that map and filter can commute

$$\forall p, f.\ filter\ p\ \circ map\ f = map\ f \circ filter\ (p \circ f)$$

This rewrite rule bears some similarity to the previous rewrite rule stating that map un-distributes through composition. But arguably, the right-hand side is not always preferable to the left-hand side. Instead, this rule states extensional equality rather than preference.

Programs as equations Such rules can be used for equational reasoning about functional programs, also called program derivation. A program can be transformed into an equivalent program by rewriting its subparts with equivalences such as filter-map commutativity.

Equational reasoning can help the programmer focus on correctness and adhere to the specification, and subsequently improve the program's performance by transforming it to equivalent, but possibly more efficient versions. Derivations using proven rewrite rules at each step can be seen as a proof of the derived program's correctness up to a specification.

The technique is not widely used in industry, perhaps due to lack of adequate tooling. Attempts to develop a system for equational reasoning in Haskell was made [27] but did not gain traction. It should be noted, however, that rewriting terms is a basic technique in theorem provers like Coq or Agda.

Applications As part of this project, equational reasoning was applied to a number of problems including the maximum segment sum problem. By deriving an efficient implementation from the specification, time complexity can be reduced from cubic to linear, and this project confirmed this with a benchmark.

Rewrite rules for matrix-vector multiplication programs A final contribution of the project concerns matrix-vector multiplication (MVmult). In many real-world applications, matrices are stored and processed in sparse formats. Each sparse matrix formats needs a distinct implementation of matrix operations like MVmult. It was observed that MVmult for the dense format is equivalent to the pair of programs: firstly, converting a dense matrix to a sparse format, and secondly, multiplying the sparse matrix with the vector. Pointfree Lift made it possible to derive programs for MVmult for the CSR formats. The value of this contribution is more conceptual than practical; however, one could consider the idea of describing sparse formats as conversions from the dense format, and then recovering the program for MVmult from the conversion.

1.2 Goals and contributions

As part of this project, the following contributions were made:

1. Implemented a system which allows for convenient description of rewrite rules and transforming programs using those rules. This system is called Pointfree Lift.
2. Implemented an algorithm for systematically pattern matching and rewriting a program using rewrite rules.

3. Implemented a polymorphic type system for Pointfree Lift. The ability to type-check programs adds another assurance of correctness to programs and helps ensure that there are no invalid rewrite rules which change the meaning of the program.
4. Showed how standard rewrites involving map, reduce and zip patterns can be expressed in Pointfree Lift.
5. Showed how equational reasoning with rewrite rules allows us to derive an efficient implementation of the program specification.
6. Showed how matrix-vector multiplication for the CSR and BSR sparse matrix formats can be derived from the program for dense matrix-vector multiplication.
7. Developed an interactive user interface for program derivation, where the user can choose a sequence of rewrite steps and resolve non-determinism when a rule is applied.
8. Implemented an interpreter for Pointfree Lift and used it to measure performance gains from program transformation.

The eventual direction of this project was rather different from the original goals. Initially, the project was supposed to focus on sparse matrix formats and perform rewriting of matrix programs within the Lift compiler. But as the complexity of rewriting matrix formats proved unwieldy in Lift, a dedicated system for performing rewrites was invented, Pointfree Lift. With the help of the system, there was scope for demonstrating program derivations beyond sparse matrix formats.

1.3 Overview and organisation of the report

Chapter 2 explains background topics which will be needed in later sections.

Chapter 3 contains a literature review and evaluation of exisiting work.

Chapter 4 explains the design of Poinfree Lift.

Chapter 5 discusses the implementation of Pointfree Lift, including the interpreter and the interactive user interface.

Chapter 6 present several derivations which can be performed in Pointfree Lift, among them, an original derivation for a sparse matrix format.

Chapter 7 reports the results of benchmarks, where the interpreter is used to confirm the predicted gains from transforming the programs.

Finally, Chapter 8 contains a summary of the results accomplished, and an outline of plans for the second part of the project.

Chapter 2

Background

This chapter provides background information on a number of loosely related topics, including rewrite rules, catamorphisms, sparse matrix formats, and the Lift project. Familiarity with these themes will be useful in later chapters.

2.1 Rewrite rules

This section discusses the concept of rewrite rules as means of optimisation or as steps in programs derivations. It explains the concept, as well as conventions assumed in this report.

An example of a rewrite rule is

$$\forall f, g.\ map\ f \circ map\ g = map\ (f \circ g)$$

The left-hand side of the rule consists of a function application or function composition. Unlike regular code, rewrite rules can contain variables. In the example, variables are f and g.

Pattern matching In the process of rewriting, a rewrite rule is matched against all nodes in the AST of the program. For a rule to match, the left-hand side must be syntactically equal to a syntax tree node, with variables replaced with subexpressions of the program.

To simplify pattern matching, there are some constraints on the use of variables on the left-hand side. Variables can only occur in the rule but not in the program, and any variable can only appear once on the left-hand side. For example $map\ (f \circ f)$ is not a valid LHS of a rewrite rule.

When it's possible to pattern match a rewrite rule against a node in the syntax tree, the program is transformed by replacing the matched node with the right-hand side of the rewrite rule. Variables are substituted for accordingly to the matched patterns.

For example, we can apply the rewrite rule for map distributivity over composition to the following program.

```
map (plus one) . map (mult one)
```

As we pattern match the left-hand side of the rule with the program, f corresponds to *plus one* and g corresponds to *mult one*. When we substitute these into the right-hand side of the rewrite rule, we obtain

```
map (plus one . mult one)
```

It is important to note that, since we try to match a rewrite rule against all nodes of the program syntax, it can apply in zero or more places. Therefore, the operation of applying a rewrite rule typically returns a list of all possible resulting programs. In this way, we model non-determinism. In particular, observe that a rule may not apply to a program at all.

2.2 Catamorphisms and Horner's rule

This section introduces the concept of catamorphism and argues that correctness of many rewrite rules used follows directly from properties of catamorphisms. It also discusses generalised Horner's rule, a non-trivial law which will be used in a derivation in a later chapter.

Most of the rewrite rules used in this work are relatively self-evident, and could easily be proved with an equational or inductive proof. However, those rewrite rules can also be seen as stemming from category theory interpretation of data structures and computations on them. Assuming the categorical perspective provides benefits. It allows us to abstract from specific details of a given rule and to use generals laws concerning categorical structures. It allows us to perform non-trivial generalisations, like that of Horner's rule for evaluating polynomials (introduced in this chapter). Finally, the existence of fundamental mathematical theory behind the practical matter at hand indicates the possibility of rigour and formalism in our reasoning.

Catamorphisms Central to this section is a concept of catamorphism. To define a catamorphism formally, we would need a few more chapters to introduce basic concepts and laws of category theory. We cannot do it here and instead will refer an interested reader to [7].

Instead, we could informally explain a catamorphism as follows. Given a recursive data type (like a linked list or a tree), a catamorphism is a recursive function on the data type such that the structure of the computation mirrors the data type. This is summarized in a quotation from Meertens [21]:

> "Catamorphisms are functions on an initial data type (an inductively defined domain) whose inductive definitional pattern mimics that of the type.

These functions have powerful calculation properties by which inductive reasoning can be replaced by equational reasoning."

Linked list catamorphisms A functional programmer likely uses catamorphisms even if not aware of it. For example, the *foldr* function in Haskell embodies catamorphisms for the linked list data type. The linked list data type is commonly defined as

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

The data type declaration above says that the *List* data type, parametrised by the type *a*, has two constructors: (1) *Nil*, which takes no arguments, and (2) *Cons*, which take an argument of type *a* and an argument of type *List a*. Notice that the data structure is recursive in the sense that its constructor contains itself as a field.

We now consider the type signature and definition of the *foldr* function. *foldr* takes three parameters: a function f, an initial value of *acc* of type *b*, and a list of elements of type *a*. The function f takes a list element of type *a* and an accumulated value of type *b* and produces another value of type *b*. The result of *foldr* is the result of "folding" the list element by element, starting with the initial value, to obtain the final value of type *b*. In the definition below, notice the use of infix notation $a`f`b$, which means that a binary function is applied to arguments *a* and *b*.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc [] = acc
foldr f acc (x:xs) = x `f` fodlr f acc xs
```

Finally, we look at the result of applying a generic fodlr to a list. Data type constructors like *Cons* are also functions, so we can apply them using the infix notation *a* '*Cons*' *b*.

```
let xs = x `Cons` (y `Cons` (z `Cons` []))
foldr f acc xs
-- evaluates to
x `f` (y `f` (z `f` acc))
```

Observe that in the result of the application, *nil* is replaced by *acc* and each *cons* is replaced by f. Crucially, a catamorphism can be viewed as an operation where the data structure constructors are replaced by specific values or functions.

Lists and binary tree catamorphisms In the remainder of this chapter, we will model a list (in the sense of an ordered multiset) with a different data structure (or different initial algebra, using categorical terminology). Specifically, the data structure will be a binary tree where leaves store values and nodes concatenate lists contained in subtrees. Catamorphisms on such data structure are suitable for rewriting in the map/reduce and split/join computation paradigm of Lift IR.

This data structure has the following definition:

```
data TList a
  = [a]
  | (TList a) ++ (TList a)
```

In this definition, $[a]$ is a value-storing leaf, and $(TList\ a) +\!\!+ (TList\ a)$ is a node which concatenates the lists in the subtrees. It is crucial that we only view a list as a tree for the sake of categorical reasoning. In an actual system, a list could as well be implemented as a flat array.

Definition of a catamorphism The discussion of theorems and laws concerning list catamorphisms, below, is due to [12]. We follow Gibbon's way of introducing list catamorphisms closely, as his treatment is as comprehensive and structured as it could be.

Suppose that $f :: a \to b$ and $\oplus :: b \to b \to b$. Then the two equations given below have a unique solution $h :: [a] \to b$.

$$\begin{aligned} h\ [x] &= [f\ x] \\ h\ (xs\ +\!\!+ ys) &= h\ xs \oplus h\ ys \end{aligned}$$

This fact is known as the unique extension property. We write this solution as $(\!|f, \oplus|\!)$ (it is completely determined by f and $\oplus$) and call it a list catamorphism. Stated another way, the list catamorphism $(\!|f, \oplus|\!)$ satisfies

$$\begin{aligned} (\!|f, \oplus\,|\!)\ [x] &= [f\ x] \\ (\!|f, \oplus\,|\!)\ (xs +\!\!+ ys) &= (\!|f, \oplus\,|\!)\ xs \oplus (\!|f, \oplus\,|\!)\ ys \end{aligned}$$

and in fact is the only solution of these equations.

The list catamorphism can be thought of as relabelling, replacing every occurrence of [_] by f and every occurrence of $+\!\!+$ by $\oplus$. For example

$$(\!|f, \oplus\,|\!)\ ([a] +\!\!+ [b] +\!\!+ [c]) = f\ a \oplus f\ b \oplus f\ c$$

Examples of catamorphisms Many useful functions are list catamorphisms. Some examples are:

$$\begin{aligned} id &= (\!|id, +\!\!+|\!) \\ map\ f &= (\!|[\_] \circ f, +\!\!+|\!) \\ last &= (\!|id, \gg|\!) \\ sum &= (\!|id, +|\!) \\ product &= (\!|id, \times|\!) \\ max &= (\!|id, \uparrow|\!) \\ join &= (\!|id, +\!\!+|\!) \end{aligned}$$

where $a \gg b = b$ and $a \uparrow b$ is the greater of a or b.

Note that in each of these examples, the second component of the list catamorphisms is associative.

Note that the notation $(\!|f, \oplus|\!)$ is a concise way of expressing the mapping and folding aspects of a list catamorphism, which emphasises that these two aspects represent a single operation of processing a list into a single value (which can still be a list). We have the equality

$$(f, \oplus) = reduce \oplus \circ\ map\ f$$

Some properties of catamorphisms A key to proving some of our rewrite rules is the Promotion Theorem for list catamorphisms. We state it below; see [12] for a full proof.

If $\oplus$ and $\otimes$ are associative, and $h\ (x \oplus y) = h\ x \otimes h\ y$ for all x and y (we say h is $\oplus$ to $\otimes$ promotable), then

$$h \circ (f, \oplus) = (h \circ f, \otimes)$$

The proof of the Promotion Theorem is by the Unique Extension Property.

As a corollary, we have the catamorphism promotion law

$$(\!|f, \oplus|\!) \circ join = (\!|id, \oplus|\!) \circ map\ \ (\!|f, \oplus|\!)$$

A special case of this is called map promotion

$$map\ f \circ join = join \circ\ map\ (map\ f)$$

2.2.1 Generalized Horner's rule

The reader might be familiar with Horner's rule for evaluating polynomials. In general, if we evaluate each term of the polynomial by performing i multiplications for the term $a_i x^i$, then the number of steps needed the evaluate the polynomial is quadratic in the degree of the polynomial. Using Horner's rule reduces that time to linear.

For example, consider the polynomial

$$a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

Using Horner's rule yields a recipe for evaluating it which runs in linear time.

$$a_0 + x(a_1 + x(a_2 + x a_3))$$

We can generalise Horner's rule to expressions of the form

$$(\!| id, \oplus |\!) \circ map\ (\!| id, \otimes |\!) \circ tails \tag{2.1}$$

where

$$tails\ [x_1, \ldots, x_n] = [[x_1, \ldots, x_n], \ldots, [x_{n-1}, x_n], [x_n], []]$$

and $\otimes$ ditributes backwards through $\oplus$, and $\otimes$ has left unit e.

Then we can rewrite Equation 2.1 as $foldl\ (*)\ e$ where $a * b = (a \otimes b) \oplus e$, which evaluates in linear time in the length of the argument list.

We will use this form of Horner's rule in some of our derivations to reduce the time complexity of a part of our algorithm from quadratic to linear.

The usefulness of Horner's rule and similar results is summarised by [12].

> "This generalised Horner's Rule illustrates a technique (where one can) encapsulate common patterns of computation as higher-order operations and to identify general purpose theorems concerning these higher order operations - often with algebraic properties of their component operations of side conditions. Such general-purpose theorems can lead to efficient solutions to algorithmic problems."

2.3 Sparse matrice formats

In real-world applications, matrices are often stored in a sparse format. Kjostadt et al. give an illustration of how using sparse formats allows to store data which would not be feasible to store in a dense format:

> "Examples of large datasets used in data analytics include Netflix Ratings, Facebook Activities, and Amazon Reviews (Tensor Compiler). In particular, the Amazon Reviews dataset, if viewed as a collection of dense matrices, contains 1.5×1019 components corresponding to 107 exabytes of data (assuming 8 bytes are used per component), but only 1.7×109 of the components (13 gigabytes) are non-zero." [17]

Sparse formats for matrix storage are characterised by not physically storing all or some of zero elements. There is a number of standard sparse formats, as well as an arbitrary number of hybrid formats. Below, we explain two formats which we use in this work: CSR and BSR.

2.3.1 Compressed Sparse Row Format (CSR)

The CSR format does not physically store any zero elements. Instead, it stores non-zero entries together with their column index. For example, Figure 2.1a show the original matrix and Figure 2.1b the matrix in CSR format, with non-zero entries annotated with column indices and zero entries removed.

$$\begin{bmatrix} 2.0 & 0 & 5.0 \\ 9.0 & 2.0 & 7.0 \\ 0 & 0 & 0 \\ 0 & 6.0 & 0 \end{bmatrix}$$

(a) Original matrix

$$\begin{bmatrix} [(0,2.0) & (2,5.0)] & * \\ [(0,9.0) & (1,2.0) & (2,7.0)] \\ * & * & * \\ [(1,6.0)] & * & * \end{bmatrix}$$

(b) Matrix in CSR format

Figure 2.1: Conversion to CSR format

Observe that the type of a matrix with n rows in CSR format is $[[(Int, Float)]]_n$, or potentially $([[Int]]_n, [[Float]]_n)$ if we decoupled elements from their column indices. Often when CSR is defined in terms of flat arrays, a third array is introduced, which stores offset to rows in the element and column index arrays. However, we don't need that in the setting of the Lift language, which supports nesting arrays of different lengths, and does bookkeeping of offsets by itself.

2.3.2 Block Compressed Row Format (BSR)

The BSR format is in principle similar to CSR format. However, the original matrix is divided into blocks or tiles, and then only blocks with non-zero elements are physically stored (that is, blocks with all zero elements are not stored). An example of conversion from the dense format to the BSR format is given in Figure 2.2.

Observe that after tiling, the matrix becomes a matrix of blocks. If we speak of rows-of-blocks and columns-of-blocks, then in each row-of-blocks, we only store non-zero blocks, annotated with their column-of-blocks index.

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 3 & 6 \end{bmatrix}$$

(a) Original matrix

$$\left[\begin{array}{cc|cc} 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 4 & 5 \\ 0 & 0 & 3 & 6 \end{array}\right]$$

(b) Tiled matrix (divided into blocks)

$$\begin{bmatrix} \left[\left(0, \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}\right)\right] & * \\ \left[\left(1, \begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}\right)\right] & * \end{bmatrix}$$

(c) Matrix in BSR format

Figure 2.2: Conversion to BSR format

Suppose that the original matrix has size $mk \times nk$ and is divided into blocks of size $k \times k$. Then the type of the that matrix stored in BSR format is $[[(Int, [[Float]_k]_k)]]_n$, or potentially $([[Int]]_n, [[[[Float]_k]_k]]_n)$.

2.4 Algebraic data types and pattern matching in Scala

This section describes several techniques in Scala which a reader familiar only with Java might want to know. In particular, we present a Scala idiom for creating immutable, algebraic data types, and a mechanism for pattern matching on those data types.

Consider a definition of the *Optional* data type, which models a value which may be absent.

```
data Optional a
  = Some a
  | None
```

The *Optional* type is parametrized by the type a. It has two constructors: *Some*, which holds a value of type a, and *None*, which holds nothing.

Scala doesn't have an explicit notion of sum types with many constructors, but an equivalent functionality can be achieved by having several case classes extending an abstract class.

> "Case classes can be seen as plain and immutable data-holding objects that should exclusively depend on their constructor arguments.
>
> This functional concept allows us to
>
> - use a compact initialisation syntax $(Node(1, Leaf(2), None)))$
> - decompose them using pattern matching
> - have equality comparisons implicitly defined
>
> In combination with inheritance, case classes are used to mimic algebraic datatypes." [24]

The *Optional* data type would be defined, using algebraic data types in Scala, like this:

```
sealed abstract class Optional[A]
final case class Some(v: A) extends Optional[A]
final case object None extends Optional
```

The desired functionality is achieved using Scala keywords. *sealed* requires that the class is only extended in the file where it is defined; thus, all class children are statically known and pattern matching can be checked for exhaustiveness. Case classes are made *final* because the constructors of the data type must be immediate children of the abstract class. Finally, *None* is a *case object* rather than a *case class* since it doesn't hold any values.

A particularly useful feature of case classes is an ability to pattern match on them. The following function's output depends on the constructor of the *Optional* type:

```
def patternMatch(optName: Optional[String]): Unit = optName match {
  case Some(name) => println("The␣name␣is␣" + name)
  case None => println("No␣name␣given")
}
```

Algebraic data types and pattern matching are used extensively in the implementation.

2.5 Lift project

This work originated in an attempt to express relatively complex rewrite rules in the Lift language [26], and for this reason, Lift deserves mention. Although the objective of compiling Pointfree Lift to Lift IR was not realized, Pointfree Lift could still aid deriving efficient programs which can then be manually translated to Lift IR. Another way in which this work could impact the Lift project is by providing a proof of concept that the user can interactively resolve the choice of rewrites.

The Lift compiler accepts programs, or kernels, written in Lift IR, functional intermediate representation based on parallel patterns. The compiler applies rewrite rules to the kernel in order to search for an efficient realisation of the kernel and compiles such optimised kernel to OpenCL for execution on a GPGPU.

Rewrite rules in Lift Lift's ability to explore different variants of a given program is based on the use of rewrite rules. Informally, in the context of Lift, a rewrite rule is a pair of expressions which can be used interchangeably without changing the extensional properties of the program, i.e. applying a rewrite rule will yield a kernel which will give the same output for a given input. However, applying rewrite rules can change intensional properties of the program or its inner structure. Rewrite rules can be used to avoid creation of intermediate data structures or change the flow of data within the program.

Lift is an improvement over manually writing kernels for GPUs in OpenCL or CUDA. However, writing kernels for matrix operations on sparse matrices in different formats can still be cumbersome and error-prone. These difficulties could be addressed by continuations of this project.

Among the programs we derive in this project, programs for sparse matrix-vector multiplication are particularly relevant to Lift. One possible continuation of this project would be to use our derivations to describe sparse matrix formats and then automatically generate programs for *spMVmult* from such descriptions.

2.6 Summary

This chapter discussed several topics, familiarity with which is assumed in later chapter. In particular, rewrite rules, catamorphisms, generalised Horner's rule, selected sparse matrix formats, and the Lift project were covered.

Chapter 3

Related work

In this chapter, we review literature which is related to our project. Specifically, we trace the historically evolving notion of the relation between a program's specification and implementation, in a broad sense.

3.1 Towards structured programming and denotational semantics

Functional programming and denotational semantics The foundations for mathematical reasoning about programming languages and programs were laid in the 60s and 70s by several landmark papers which advocated grounding the meaning of programs in mathematical objects. One such paper was "The next 700 programming languages" by Peter Landin [18], which presents a design of a programming language whose constructs correspond to mathematical objects. The paper effectively introduced the concept of denotational semantics, and the proposed programming language ISWIM arguably influenced, directly or indirectly, almost all subsequent designs.

Structured programming Edsger's Dijkstra "Notes on Structured Programming" [11] was another influential paper which advocated programming in a style which reflects the specification of the program rather than the inner workings of the machine which executes the program. This was to be achieved through the use of structured control flows with while-loops and if-statements, as opposed to unstructured goto statements. Dijkstra motivated his appeal like this:

> "It is my (unproven) claim that the ease and reliability [of programming] depends critically upon [. . .] the nature of sequencing control. In vague terms, we may state the desirability that the structure of the program text reflects the structure of the computation. Or, in other terms, "What can we do to shorten the conceptual gap between the static program text (spread out in "text space") and the corresponding computations (evolving in time)?""

The legacy of Dijkstra's letter is hard to overlook for any modern programmer.

Logic of computer programs The benefits of structured programming were independently exposed by Tony Hoare in his work ”An axiomatic basis for computer programming,” [15] which presents a formal system where a set of logical rules corresponding to a program's constructs allows for fully rigorous proofs about imperative programs. Thanks to Hoare's logic, an imperative program can not only be derived from a specification but also proven to adhere to it.

3.2 Program derivation with Bird-Meertens formalism

While structured programming applies to imperative programs, there emerged in the 80s another, more functional approach. One of its early proponents was Lambert Meertens, who in his paper ”Algorithmics — Towards programming as a mathematical activity” [20] argues that the programming community should invent a systematic process of deriving efficient implementations from correct specifications.

Bird-Meertens formalism Meertens believed that a notation was needed which would be capable of expressing both the specification and the implementation in a single language, even if the more abstract parts of it would not be executable. Meertens had a hope that with the help of a powerful, rigorous notation, high-level theorems about programs could be discovered, comparable in their scope and power with mathematical results like Cauchy's Integral Theorem. In the end, the notation which emerged from the work of Lambert Meertens and Richard Bird came to be known as the Bird-Meertens formalism. The notation was facetiously dubbed Squiggol, alluding to Algol on the one hand, and to the “squiggly” symbols used on the other.

There are several program derivations in literature, many published in the late 80s and early 90s in “The Squiggolist”. “An introduction to the Bird-Meertens formalism” by Jeremy Gibbons shows how to derive an efficient implementation for the maximum segment sum problem [12]. (We express this derivation in Pointfree Lift in a later chapter.) Another paper by Gibbons presents a derivation of radix sort [13]. A result by Bird and Meertens [8] shows how to use program transformation to develop alpha-beta pruning from the specification of the minimax algorithm [9]. The literature contains an abundance of other examples.

Program transformation using laws of category theory Another direction of research in the field of program derivation was to use results from category theory to prove the correctness of abstract rewrite rules. A survey of results can be found in “Algebra of Programming” by Bird and De Moor [7]. On the more practical side, “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire” by Erik Meijer et al. [22] presented a set of programming patterns motivated by category theory.

Program transformation implementations Although Bird-Meertens formalism is just a notation, and it was never implemented as a real-world executable system, it did influence functional languages like Haskell. Although Haskell by itself has no program derivation capabilities, there were attempts to build a tool for deriving Haskell programs. One such attempt was the PATH (Programmer Assistant for Transforming Haskell), developed by Mark Tullsen [27], who motivated his work like this

> "This example demonstrates the general principle that a clear program is more easily seen to be correct, is faster to develop, and is easier to maintain. Likewise, an efficient program is usually less clear, is slower to develop, and is harder to maintain. However, in software development we want clarity and efficiency. There are two major approaches to getting both."

The first approach is verification, done after the software is written. The second approach, taken by Tullsen, is deriving a program from its specification. Tullsen acknowledged obstacles to widespread adoption of program derivation, like lack of industrial strength tools and a deficit of necessary expertise among programmers. Tullsen hoped that his project would simplify the process; unfortunately, to this day, program derivation remains a niche technique.

3.3 Program transformation as an optimisation technique

Program transformation has been more successful in its limited form, where simple transformations are used as a means of optimising the performance of programs. The concept of transforming programs is as old as the compiler, but the gap between optimising compilers and algebraic program transformation is bridged by the concept of rewrite rules, which serve as a simple, mathematically-grounded way to declare optimising transformations.

GHC rewrite rules A practical implementation of rewrite rules was described in "Playing by the rules: rewriting as a practical optimisation technique in GHC" by S.P. Jones, C.A.R. Hoare, and A. Tolmach [16]. GHC [3] accepts rewrite rules written in the Haskell language itself, making them arguably the simplest interface for specifying compiler optimisations. One of the main applications of rewrite rules in GHC is to eliminate intermediate data structures.

The Lift project Another project which makes prominent use of rewrite rules in the Lift compiler [26]. Lift programs are written in a functional intermediate language; such high-level form allows the compiler to optimise programs with rewrite rules before part of the structure is lost through compilation to OpenCL. The process of compilation from functional to procedural code is one instance of deriving an efficient program from a semantically rich one.

3.4 Dependently-typed programs as proofs

There have recently been attempts to connect programs with their specifications using dependent types. According to a result known as Curry-Howard isomorphism [25], a dependent type of a program is equivalent to a logical statement, and the typed program corresponds to a proof of the statement. Thus dependent types allow, or in fact force, the programmer to embed the proof of correctness in the program itself.

A dependently typed program might not be efficient by itself, but some system can extract executable programs. The Coq proof assistant [2], which is based on dependent types, can extract programs in Haskell and OCaml, among others. The Idris language [4] is an attempt to build an efficient, dependently-typed system.

3.5 Domain specific languages as program specifications

Under a broad interpretation, domain-specific languages can be seen as program specifications from which an actual implementation is derived. Indeed, the idea behind DSLs is to have the programmer focus on expressing the meaning of a program, and have the compiler transform it to an efficient, executable form.

DSLs for sparse matrix formats One relevant piece of work is a paper by Grewe [14], which introduced a notation for describing matrix formats. From the abstract description, efficient OpenCL kernels for matrix operations can be generated. This work is relevant inasmuch we derive programs for different matrix formats in this project. However, we could not use Grewe's results directly, since his format description relies on random access to matrix elements, whereas Pointfree Lift functional programs need to process a matrix with aggregate patterns like map and reduce.

3.6 Cricitical evaluation and the project's contributions

Progress in the field of programming languages developed understanding of programs as mathematical objects rather than as entities tied to the particular machine which executes a program. Programs as mathematical object can be reasoned about in terms of algebraic properties and equivalences. Today's compilers use such properties to apply simple optimisations to programs, for example to eliminate intermediate data structures. More complex applications like formal derivation of programs from their specifications are not widely used in real-world applications, but remain instructive for programmers.

The main contribution of this project is an attempt to bridge the gap between the field of program transformation and the world of optimising compilers represented by the

Lift project. This was done by desiging and implementing a system for a language which is similar in its character to Lift IR but supports program transformation and derivation. It was also demonstrated that even a small work like this can make tiny contributions to the field of program derivations, by deriving one program for matrix-vector multiplication from another.

Chapter 4

Pointfree Lift: design

Pointfree Lift is a language we designed specifically for the project. It is expressive enough for matrix-vector multiplication on different sparse matrix formats, and it can be easily extended by adding new primitives to its interpreter. It is similar in its spirit to Lift IR, especially because it is based on functional patterns acting on arrays. Whereas the Lift IR compiler performs low-level rewrites as a means to optimise the generated OpenCL code, Pointfree Lift is capable of applying high-level rewrite rules which change the structure of the whole algorithm.

4.1 Design decisions

Pointfree Lift is distinct from Lift IR for two reasons. Firstly, its abstract syntax is simpler, and thus easier to rewrite. But more importantly, Pointfree Lift avoids lambda abstractions and binding values to names. Instead, a Pointfree Lift program, which is a function itself, is made up of compositions of functions (this is where Pointfree Lift derives its name from). Reliance on function composition, as opposed to a context of named values, is another characteristic of Pointfree Lift which makes it suitable for rewriting.

4.2 Type system

Pointfree Lift features a polymorphic type system (complex types are parametrised, like in Java generics).

Simple types are

`Int Float Bool`

Complex types are, in terms of type parameters a and b

Pair type: `(a, b)`

List type: `[a]`

Function type: $a \to b$

Additionally, the type system uses lowercase Latin letter to denote type variables for parametric polymorphism. Since the language is pointfree and there is no type context, there is no need to distinguish between bound and unbound type variables like in Hindley-Milner.

4.3 Syntax

The syntax of Pointfree Lift is given by a very simple grammar:

$\langle expr \rangle$ ::= $\langle symbol \rangle$
  | $\langle expr \rangle$ '`.`' $\langle expr \rangle$
  | $\langle expr \rangle$ $\langle expr \rangle$

where the second line represents function composition, and the third line represents function application. As we mentioned, functional (lambda) abstractions are not supported by design.

An example program would be pretty-printed as follows. Incidentally, this would parse as a pointfree subset of Haskell; functional languages are similar in essence. Following Haskell, we will the use of the :: symbol to give type annotations for symbols.

```
denseMV :: [[Float]] -> [Float]
denseMV = map (fold plus . map (uncurry mult) . zip vector)
```

4.4 Built-ins

A Pointfree Lift program is a term built from symbols, combined using function application and composition. The user is free to add any symbol, as long as its type and method of evaluation are specified. Below we list the primitives which are needed to perform CSR matrix-vector multiplication, together with their types (we will use Haskell-like style). Note that the choice of symbols is arbitrary – typically the choice of symbols reflect the granularity we need for a particular set of rewrite rules.

```
identity :: a -> a
map :: (a -> b) -> List(a) -> List(b)
reduce :: (a -> a -> a) -> a -> List(a) -> a
filter :: (a -> Bool) -> List(a) -> List(a)
uncurry :: (a -> b -> c) -> Pair(a, b) -> c
zip :: List(a) -> List(b) -> List(Pair(a, b))
unzip :: List(Pair(a, b)) -> Pair(List(a), List(b))
access :: List(a) -> Int -> a
neq :: a -> a -> Bool
```

```
snd :: Pair(a, b) -> b
plus :: Float -> Float -> Float
mult :: Float -> Float -> Float
zero :: Float
one :: Float
vector :: List(Float)
enumeration :: List(Int)
```

The meaning of most of these built-ins should be apparent to a reader who is acquainted with a polymorphic functional language like Haskell. We will briefly discuss ones which might not be self-explanatory. Note that functions are curried by default.

$fold$ takes a binary operation on a set a, the initial element and a list of elements of type a. It reduces the list to a value of type a using the operation.

$access$ takes a list and an integer i. It returns the i-th element of the list.

$vector$ is a list of floats. We designed Lift to be pointfree, i.e. not to use values bound to names. Such constraint simplifies some functions but complicates others. Since we focus on the problem of matrix-vector multiplication, we hardcode the vector value as part of the language, thus circumventing the limitations of pointfree style.

$enumeration$ is an infinite (lazy) list of integers from 0 onwards. When zipped with any other list of type $List(a)$, it produces its enumeration of type $List(Pair(Int,a)$.

4.5 Type inference

The types of complex expressions are inferred from the types of subexpressions, using inference rules for function application and composition. The process relies on unification of types.

4.5.1 Unifying types

In the next section, we will discuss function application and composition, whose typing rules rely on unification of type expressions. Type unification works in a similar manner as in the Hindley-Milner type system [23].

Recall that a type expression can be made up of simple types like $Float$, complex types like $a \to b$, and type parameters a. Unification is a process of solving an equation involving two type expressions, where type parameters become variables. If a solution exists, it is represented as a substitution, that is, a mapping from type variables to type expressions.

For example, unifying $Pair(a,List(b))$ with $Pair(List(Int),List(Int))$ yields a substitution $a/Pair(Int),b/Int$. On the other hand, an attempt to unify $Pair(a,a)$ with $Pair(Float,Int)$ yields no solution, as a cannot be a $Float$ and an Int at the same time.

Type expressions do not contain free variables. This is possible since our language is pointfree, and we do not have a type context where names are bound to types. This also means that we can rename type variable without changing the meaning of the type: $a \to b$ could be expanded to $\forall a, b. a \to b$ and is the same as $c \to d$.

4.5.2 Type inference for function application and composition

Type inference for function application and composition are given by the following rules

$$\frac{f : a \to b \quad e : a' \quad \sigma = unify(a, a')}{f\ e : b\sigma}\ \text{Application}$$

$$\frac{f : b' \to c \quad g : a \to b \quad \sigma = unify(b, b')}{f \circ g : a\sigma \to c\sigma}\ \text{Composition}$$

An inference rule is composed of one or more premises and the conclusion. When the premises hold, the conclusion also holds. For example, the inference rule for function application can be read as follows. Given that f has type $a \to b$, and e has type a', and types a and a' can be unified with a substitution σ, then the application $f\ e$ has type $b\sigma$, which is the type expression b with the substitution σ applied.

As an example of type inference for function application, consider the application $fold\ plus$. Recall that

$$\begin{gathered} reduce :: (a \to a \to a) \to a \to List(a) \to a \\ plus :: Float \to Float \to Float \end{gathered}$$

The type expression $a \to a \to a$ unifies with $Float \to Float \to Float$ for $a \leftarrow Float$. The inference rule for application is instantiated to

$$\frac{\begin{gathered} reduce :: (a \to a \to a) \to a \to List(a) \to a \\ plus :: Float \to Float \to Float \\ \sigma = a/Float \end{gathered}}{reduce\ plus :: Float \to List(Float) \to Float}$$

As for function composition, consider the composition

$$map\ (plus\ one) \circ map\ (access\ vector)$$

The expression is made up of function applications who types are

$$map\ (plus\ one) :: List(Float) \to List(Float)$$
$$map\ (access\ vector) :: List(Int) \to List(Float)$$

The inference rule for function composition is instantiated to

$$\frac{\begin{array}{c} map\ (plus\ one) :: List(Float) \to List(Float) \\ map\ (access\ vector) :: List(Int) \to List(Float) \\ \sigma = \{\} \end{array}}{map\ (plus\ one) \circ map\ (access\ vector) :: List(Int) \to List(Float)}$$

$List(Float)$ trivially unifies with itself, so the type of the function composition is $List(Int) \to List(Float)$.

Note that when we perform type inference for application $f\ e$ or composition $f \circ e$, we should ensure that the subexpressions f and e do not share any type variables. To this end, we peform variable renaming, also called alpha conversion. This can be done trivially, as there is no type context, and an occurrence of a type variable x in f is unrelated to an occurence of x in e. For example, suppose we have a function $f :: Pair(Int, a) \to a$ which we apply to a value $e :: Pair(a, Float)$. Clearly, unification of $Pair(Int, a)$ and $Pair(a, Float)$ would fail. But if we rename type variables to $e :: Pair(b, Float)$, then we can unify $Pair(Int, a)$ and $Pair(b, Float)$ for $a \leftarrow Float, b \leftarrow Int$, and the inferred type for the application $f\ e$ will be $Float$.

4.6 Example programs in Pointfree Lift

We present three example programs in Pointfree Lift. To avoid the clutter of AST nodes names, we use a pretty-printed, Haskell-like representation.

```
denseMV = [[Float]] -> [Float]
denseMV = map (fold plus . map (uncurry mult) . zip vector)

csrMV :: [[(Int, Float)]] -> [Float]
csrMV = map (fold plus . map (uncurry (mult . access vector)))

denseToCsr :: [[Float]] -> [[(Int, Float)]]
denseToCsr = map (filter (neq zero . snd) . zip enumeration)

-- observe that
-- denseMV = csrMV . denseToCsr
```

denseMV performs multiplication with a vector on a dense matrix. Each row is zipped with the vector, and then the dot product is computed by multiplying each pair of numbers and then summing them all.

csrMV performs matrix-vector multiplication on a matrix in a CSR format. In the input, each element of the row comes in a pair with its column position within its row. When we multiply it with the corresponding element of the vector, we use the position to access the vector's element.

denseToCsr converts a matrix from the dense format to the CSR format. For each row, the elements are enumerated, and then zero elements are filtered out.

4.7 Summary

The design decision of making the language pointfree (not use variable bindings) was intended to make the language easy to transform, as transformations involving lambda abstractions and bound variables are notoriously difficult and only recently researched. But this design has the benefit of simplicity: complex expressions are built by applying and composing functions, and a lack of free/bound variables makes type inference easy. In spite of its simplicity, a clever choice of primitives makes Pointfree Lift powerful enough to express the program transformations discussed in Chapter 6.

Chapter 5

Pointfree Lift: implementation

This chapter discusses the implementation of Pointfree Lift. Its organisation mirrors the organisation of Chapter 4 on the design of the language. Functional aspects of Scala allow the implementation to correspond closely to the mathematical specification. This chapter includes concise snippets of code in the hope that the declaritive code is understandable enough to be useful.

5.1 Design decisions

Pointfree Lift is implemented in Scala for several reasons

1. Scala has very good support for the functional programming paradigm. Implementation is done in a functional style, which allows the code to correspond closely to the specification of Pointfree Lift.
2. Scala supports advanced programming techniques like monads, which were used for implementing more complex algorthms, including unification.
3. The Lift project, which inspired this project, is implemented in Scala.

5.2 Type system

Types are implemented as algebraic data types in Scala

```
sealed abstract class Type {...}
case object TInt extends Type
case object TFloat extends Type
case object TBool extends Type
case class TList(a: Type) extends Type
case class TPair(a: Type, b: Type) extends Type
case class TArrow(a: Type, b: Type) extends Type
case class TVar(n: Int) extends Type
```

5.3 Abstract syntax

A Pointfree Lift program has type *Expr* and is made up of subexpressions combined with *Application* and *Composition*. Primitives, application and composition are implemented as case classes extending the *Expr* class.

```
sealed abstract class Expr {
  override def typ: Type
  override def evaluate: Value
}

case class Application(f: Expr, e: Expr) extends Expr {
  override def typ: Type = {...}
  override def evaluate: Value = {...}
}

case class Composition(f: Expr, g: Expr) extends Expr {
  override def typ: Type = {...}
  override def evaluate: Value = {...}
}

case object SomePrimitive extends Expr {
  override def typ: Type = {...}
  override def evaluate: Value = {...}
}

...
```

5.4 Primitives

Every primitive is implemented as a case class extending *Expr*. In the definition, we specify the type and the method of evaluation. A couple of examples are given below.

```
case object Map extends Expr {
  override def typ: Type = (A ->: B) ->: TList(A) ->: TList(B)
  override def evaluate: Value = {...}
}

case object Uncurry extends Expr {
  override def typ: Type = (A ->: B ->: C) ->: TPair(A, B) ->: C
  override def evaluate: Value = {...}
}
```

5.5 Type inference for function application and composition

The type inference rules for application and composition are implemented in their respective *typ* methods.

```
case class Application(f: Expr, e: Expr) extends Expr {
  override def typ: Type = {
    val renaming = f.typ alphaConversion e.typ
    val TArrow(a, b) = f.typ substitute renaming
    val Some(subst) = a unify e.typ
    b substitute subst
  }
}

case class Composition(f: Expr, g: Expr) extends Expr {
  override def typ: Type = {
    val renaming = f.typ alphaConversion g.typ
    val TArrow(a, b) = g.typ
    val TArrow(b_, c) = f.typ substitute renaming
    val Some(subst) = b unify b_
    (a substitute subst) ->: (c substitute subst)
  }
}
```

Alpha conversion, substitution and unification are implemented as methods on the *Type* class. Observe that both *substitute* and *_unify* methods recurse on complex types *Pair*, *List*, and *Arrow*. The *_unify* method is implemented using a state monad, provided by the Scala library Scalaz.

```
sealed abstract class Type {
  def alphaConversion(rhs: Type): Substitution = {
    val lhsVars = this.typeVariables().toSet
    val rhsVars = rhs.typeVariables().toSet
    val allVars = lhsVars union rhsVars
    val repeatedVars = lhsVars intersect rhsVars
    _alpha(allVars.toList, repeatedVars.toList)
  }

  def _alpha(all: List[Int], repeated: List[Int]): Substitution =
  repeated match {
    case Nil => immutable.Map[Int, Type]()
    case v :: rest =>
      val free: Int = (0 until Int.MaxValue)
              .find(!all.contains(_)).get
      _alpha(free :: all, rest) + (v -> TVar(free))
  }

  def substitute(subst: Substitution): Type = this match {
    case TList(a) => TList(a substitute subst)
    case TPair(a, b) =>
        TPair(a substitute subst, b substitute subst)
    case TArrow(a, b) =>
        TArrow(a substitute subst, b substitute subst)
    case TVar(n) => subst.getOrElse(n, this)
    case _ => this
  }

  def _unify(t: Type): State[Substitution, Unit] =
    (this, t) match {
      case (TVar(n), a) => modify(s => s + (n -> a))
      case (a, TVar(n)) => modify(s => s + (n -> a))
      case (TList(a), TList(b)) => a _unify b
      case (TPair(a, b), TPair(c, d)) =>
            (a _unify c) >> (b _unify d)
      case (TArrow(a, b), TArrow(c, d)) =>
            (a _unify c) >> (b _unify d)
      case (a, b) => if (a == b) State(s => (s, ()))
                     else throw UnificationException
    }
}
```

5.6 Interpreter

The implementation of Pointfree Lift comes with an interpreter. The interpreter can be used to measure efficiency gains from program transformations, and we will use it in Chapter 7.

All AST nodes need to implement the *evaluate* method, which returns a *Value* object.

```
sealed abstract class Expr {
  def evaluate: Value
}
```

Value is an algebraic data type, where constructors correspond to Pointfree Lift types. As a rule, the constructors are simple wrappers around Scala data types.

```
sealed abstract class Value
case class VInt(v: Int) extends Value
case class VFloat(v: Float) extends Value
case class VBool(v: Boolean) extends Value
case class VList(v: List[Value]) extends Value
case class VPair(fst: Value, snd: Value) extends Value
case class VFun(v: Value => Value) extends Value
case object VUndefined extends Value
```

Of particular interest is the $VFun$ constructor, which wraps around a function $Value \Rightarrow Value$. In fact, our interpreter belongs to the family of (HOAS) (Higher Order Abstract Syntax) interpreters. [29]

> "Higher-Order Abstract Syntax (HOAS) is a technique for implementing the lambda calculus in a language where the binders of the lambda expression map directly onto lambda binders of the host language [in our implementation, Scala] to give us substitution machinery in our custom language by exploiting [Scala's] implementation." [10]

The *evaluate* method for *Application* and *Composition* evaluates the subexpressions, extracts the lambda from the $VFun$ object, and applies it.

```
case class Application(f: Expr, e: Expr) extends Expr {
  override def evaluate: Value = f.evaluate match {
    case VFun(ff) => ff(e.evaluate)
  }
}
```

```
case class Composition(f: Expr, g: Expr) extends Expr {
  override def evaluate: Value = (f.evaluate, g.evaluate) match {
    case (VFun(ff), VFun(gg)) => VFun(arg => ff(gg(arg)))
  }
}
```

Each primitive (class extending *Expr*) needs to provide a custom evaluation method. We give two examples below. Recall that *Map* applies a function to every element of a list and *Uncurry* takes a function which takes two arguments, and it wraps it into a function which takes one argument which is a pair.

```
case object Map extends Expr {
  override def evaluate: Value = VFun {
    case VFun(f) => VFun {
      case VList(list) =>
        VList(list.map(e => f(e)))
  }}
}

case object Uncurry extends Expr {
  override def evaluate: Value = VFun {
    case VFun(f) => VFun {
      case VPair(a, b) => f(a) match {
        case VFun(g) => g(b)
  }}}
}
```

Thanks to the use of the HOAS technique, our interpreter is simple and straightforward.

5.7 Implementation of rewriting

In this section, we discuss Pointfree Lift's machinery for declaring and applying rewrite rules. We speak of rewrite rules from the implementation point of view, deferring the discussion of individual rules until later chapters.

There exists a rewrite rule which says that "map un-distributes through composition":

$$\forall f, g.\ map\ f \circ map\ g = map\ (f \circ g)$$

In Pointfree Lift, a rewrite rule is declared as an instance of the *Equiv* class.

```
val mapUnDistributesThroughComposition = Equiv(
    name = "map␣undistributes␣through␣composition",
    left = Map(A) *: Map(B) *: Rest,
    right = Map(A *: B) *: Rest,
    transform = s => Some(s)
)
```

We can see that a rule is described by its left-hand side and its right-hand side, where the right-hand side depends on expression variable (*EVar*) instances from the left-hand side, denoted with capital letters A, B, C, etc. In the process of applying a rule, an occurrence of the left-hand side is replaced with the right-hand side.

Additionally, some rules have a non-trivial *transform* function, which can be used to modify the substitution or even invalidate the match of the rule. The *transform* function can be used to make a pattern match conditional on algebraic properties of the function which is matched on, or to 'transform' the substitution in other ways.

The *rewrite* method on an Expr is defined as follows

```
1  def rewrite(equiv: Equiv): List[Expr] = {
2    val Equiv(_, lhs, rhs, transform) = equiv
3    (lhs unify this).flatMap(transform).map(rhs substitute).toList
       ++
4      (this match {
5        case Application(f, e) =>
6          f.rewrite(equiv).map(Application(_, e)) ++
7            e.rewrite(equiv).map(Application(f, _))
8        case Composition(f, g) =>
9          f.rewrite(equiv).map(Composition(_, g)) ++
10           g.rewrite(equiv).map(Composition(f, _))
11       case _ => Nil
12     })
13 }
```

The method takes an *Equiv* rewrite rule as its argument. The method is non-deterministic in the sense that it returns the list of expressions resulting from all possible applications of the rule; lines 4-12 of the snippet are responsible for the traversal of the expression in search of matches for the rule.

Matching on the current AST node (line 3) can be seen as proceeding in steps. Firstly, the left-hand side of the rule is matched against the current AST node with the *unify* method. Then, the *transform* function is applied to modify the substitution (or invalidate the match, if it evaluates to *None*). Finally, the substitution is applied to the right-hand side. The use of *Option* monad allows for seamless exception handling.

The *unify* method for expressions *Expr* implements a limited form of unification. This is allowed because only one side of the equation can have variables, and each variable can only appear once. Thus, unification takes the form of naive pattern matching. The definition of *Expr.unify* is given below; again, notice the use of the *State* monad.

```
def _unify(e: Expr): State[Substitution, Unit] =
(this, e) match {
  case (EVar(n), a) => modify(_ + (n -> a))
  case (Application(a, b), Application(c, d)) =>
      (a _unify c) >> (b _unify d)
  case (Composition(a, b), Composition(c, d)) =>
      (a _unify c) >> (b _unify d)
  case (a, b) => if (a == b) State((_, ()))
      else throw UnificationException
}
```

5.8 Interactive user interface

In the previous chapters, we described the design and implementation of Pointfree Lift and showed that a Pointfree Lift program could be interpreted to compare the efficiency of different programs, in particular, a specification-program and its efficient derivation. One could dismiss Pointfree Lift as unnecessary since programs can be derived with a pen and paper and then implemented in a standard programming language for benchmarking. However, in this chapter, we present an application which is, as far as we know, unique to Pointfree Lift: a user interface for interactive derivations.

5.8.1 Walkthrough

The screen is divided into four panels. The 'Programs' panel displays the list of programs in Pointfree Lift's database. In the example screenshot 5.1, these programs are (1) the specification program for the MSS problem, and (2) the program for dense matrix-vector multiplication. The user can select a program to set it as the current program.

The 'Current program and type' panel display the current program and its type. The current program is the program from the database, possibly transformed with some rewrite rules. The type of the program should not change in the course of derivation, and displaying it serves as another assurance that the rewrite rules used and the derivation are correct.

The 'Applicable rules' displays all possible applications of all applicable rules to the current program (for a given program, a rule may be applied in several places). The application of a rule is described by the rule's name and the program resulting from applying the rule. The part of the program which changes is highlighted in red. The user can select the rewrite rule; upon confirming the rule, the program derivation and the current program are updated.

Finally, the 'Derivation' panel shows the log for the current derivation. The derivation consists of a series of intermediate programs, interleaved with names of rules applied at every step.

5.8.2 Implementation

The application is developed with Scurses, a Scala library for writing terminal user interfaces (TUIs) [5]. We employ a number of design patterns to implement the user interface: the Observer pattern for subscribing to changes, application view as a function of the state, and event-driven mechanism for modifying the state. The state of the application is kept in objects of type $Varying[A]$ provided by the library, which can subscribe to changes in one another. The displayed content is a function of the state. User input from the keyboard provides feedback to the state.

The so-called front end of the application connects to the back end of Pointfree Lift through a small number of functions. The $Expr.typ : Type$ method allows for typing the current program. The $Expr.rewrite(equiv : Equiv) : List[Expr]$ method allows for obtaining the list of currently applicable rules.

5.8.3 Reception by users

Although we did not conduct official tests, a few users tried using the interface. With some guidance, they were able to complete the derivations for the MSS problem and matrix-vector multiplication in the CSR format. All users commented that they would not have been able to perform the derivations without the help of the user interface for Pointfree Lift.

5.8.4 Use cases

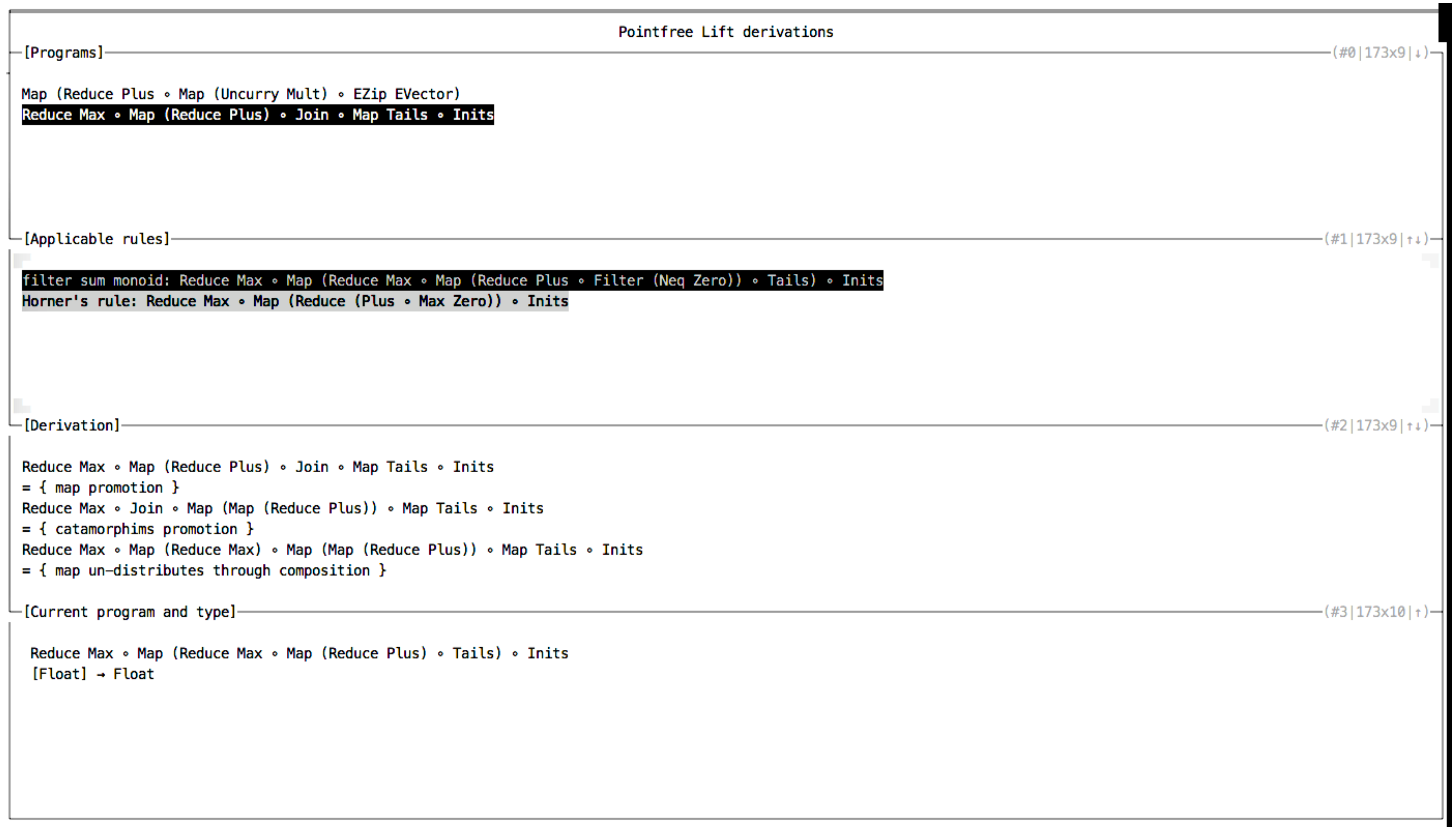
Our implementation of the user interface could be improved in terms of its user-friendliness and capabilities; at the current stage, it should be viewed as a proof of concept. We can see two kinds of uses for a tool like this. First of all, it could have educational purposes. We believe that if we ask a computer science student to derive an efficient program, then she will find it much easier if the interactive tool displays only the applicable rules and does all the bookkeeping of rewriting. In fact, there are often so few applicable rules that derivation often comes down to mechanically selecting the next rule.

Second of all, a similar tool could be incorporated into an existing compiler like Lift. The Lift compiler transforms programs by applying rewrite rules, but currently, the process is based on directed search, and there is limited scope for the human user to provide insight to guide the process. The inclusion of an interactive interface would allow the user to aid the compiler in resolving the choice of rewrite rules.

5.9 Summary

The use of functional features of Scala made it relatively straightforward to implement the complex design of an interpretable, type-checkable language. Algebraic data types were used to model types and syntax nodes, state monads were used for type unification and pattern matching, and lambda functions as first-class objects in the language allowed for the use of the PHOAS technique for implementing the interpreter.

The interactive user interface provides a user-friendly way to explore the implementation.



```
                                        Pointfree Lift derivations
─[Programs]──────────────────────────────────────────────────────────────(#0|173x9|↓)─

 Map (Reduce Plus ∘ Map (Uncurry Mult) ∘ EZip EVector)
 Reduce Max ∘ Map (Reduce Plus) ∘ Join ∘ Map Tails ∘ Inits

─[Applicable rules]──────────────────────────────────────────────────────(#1|173x9|↑↓)─

 filter sum monoid: Reduce Max ∘ Map (Reduce Max ∘ Map (Reduce Plus ∘ Filter (Neq Zero)) ∘ Tails) ∘ Inits
 Horner's rule: Reduce Max ∘ Map (Reduce (Plus ∘ Max Zero)) ∘ Inits

─[Derivation]────────────────────────────────────────────────────────────(#2|173x9|↑↓)─

 Reduce Max ∘ Map (Reduce Plus) ∘ Join ∘ Map Tails ∘ Inits
 = { map promotion }
 Reduce Max ∘ Join ∘ Map (Map (Reduce Plus)) ∘ Map Tails ∘ Inits
 = { catamorphims promotion }
 Reduce Max ∘ Map (Reduce Max) ∘ Map (Map (Reduce Plus)) ∘ Map Tails ∘ Inits
 = { map un-distributes through composition }

─[Current program and type]──────────────────────────────────────────────(#3|173x10|↑)─

  Reduce Max ∘ Map (Reduce Max ∘ Map (Reduce Plus) ∘ Tails) ∘ Inits
  [Float] → Float
```

Figure 5.1: A screenshot of the terminal user interface for Pointfree Lift

Chapter 6

Applications of Pointfree Lift: program derivation and transformation

Pointfree Lift was designed to facilitate program derivation. In fact, it does not support binding variables to names specifically in order to make it easier to transform programs. In this chapter, we show two derivations which can be expressed in Pointfree Lift: one borrowed from literature, and the other discovered as part of this project.

6.1 Applying equational reasoning to the Maximum Segment Sum problem

As we mentioned in the introduction, one major application of rewrite rules is deriving an efficient implementation from the program specification, while maintaining correctness of the program. In this section we will follow [12] and use equational reasoning to transform the specification for the Maximum Segment Sum (MSS) problem, reducing its time complexity. Our contribution is showing that all steps of the transformation can be expressed in Pointfree Lift. Thus Pointfree Lift guarantees the correctness of the derivation process, up to the rewrite rules.

The Maximum Segment Sum problem is: given a list of numbers (which can be negative), find a segment (slice) of the list such that the sum of the segment is maximized. For example, consider the list

$$[12,\ -5,\ 0,\ 32,\ -10,\ 5,\ -5]$$

The segment which has maximum sum is $[12,\ -5,\ 0,\ 32]$, with sum 39. It is not possible to obtain a sum greater than 39 by taking any other segment.

We can give a specification of the problem in terms of a functional expression in Pointfree Lift. First, we need to specify a set of Pointfree Lift primitives and functions in terms of their types and informal definitions.

tails evaluates to a lists of suffixes of its argument list

$$\begin{gathered} tails :: [a] \to [[a]] \\ tails\ [x_1, \ldots, x_n] = [[x_1, \ldots, x_n], \ldots, [x_{n-1}, x_n], [x_n] \end{gathered}$$

inits evaluates to a list of prefixes of its argument list

$$\begin{gathered} inits :: [a] \to [[a]] \\ inits\ [x_1, \ldots, x_n] = [[], [x_1], \ldots, [x_1 \ldots x_{n-1}], [x_1 \ldots x_n]] \end{gathered}$$

segments is defined as

$$\begin{aligned} &segments :: [a] \to [[a]] \\ &segments = join \circ map\ inits \circ tails \end{aligned}$$

The MSS problem is defined by the program

$$mss :: [Int] \to Int \tag{6.1}$$

$$mss = max \circ map\ sum \circ segments \tag{6.2}$$

Finally, recall that *scan* can be defined like this:

$$\begin{aligned} scan &:: (a \to b \to b) \to b \to [a] \to [b] \\ scan &= map\ (fold\ (+)\ acc) \circ tails \\ scan\ f\ acc\ [x_1, \ldots, x_n] &= [fold\ f\ acc\ [], fold\ f\ acc\ [x_1], \ldots, fold\ f\ acc\ [x_1 \ldots x_n]] \end{aligned}$$

Program transformation can change time complexity Notice that the specification of the MSS problem in Program 6.2 is executable, but it takes cubic time ($O(n^3)$) (in the length of the input list). A derivation of an efficient, linear ($O(n)$) implementation of the MSS problem is given below.

(Explanation: At each application of a rule, we undeline the matched expressionlstlisting with a dashed line and underline the change in the rewritten expression with a solid line. Since at each stage, the program needs to be underlined in two different places, we duplicate each program expression and separate the two copies with the '~' symbol.)

```
 mss
= { definition of mss }
max . map sum . segments
~
max . map sum . segments
= { definition of segments }
max . map sum . join . map tails . inits
~
max . map sum . join . map tails . inits
= { map promotion }
max . join . map (map sum) . map tails . inits
~
max . join . map (map sum) . map tails . inits
= { catamorphism promotion }
max . map max . map (map sum) . map tails . inits
~
max . map max . map (map sum) . map tails . inits
= { map un-distributes through composition }
max . map (max . map sum . tails) . inits
~
max . map (max . map sum . tails) . inits
= { Horner's rule: let a ⊕ b = (a `max` 0) + b }
max . map (fold ⊕) . inits
~
max . map (fold ⊕) . inits
= { scan }
max . scan ⊕
```

In Pointfree Lift, we would specify this derivation as follows. Note that the above sketch of the derivation is only slightly different from the Pointfree Lift's output when running the derivation.

```
@Test
def rewriteMaxSegSum(): Unit = {
  (Programs.maxSegSum :: Nil)
    .tap(println)
    .rewrite(mapPromotion)
    .rewrite(catamorphismPromotion)
    .rewrite(mapDistributesThroughComposition)
    .rewrite(mapDistributesThroughComposition)
    .rewrite(hornersRule)
    .rewrite(scan)
    .typecheck()
}
```

The rewrite rules used are specified in Pointfree Lift as follows. We present the source code here for completeness; implementing a rewrite rule in Pointfree Lift is trivial.

```
val mapUnDistributesThroughComposition = Equiv(
  name = "map␣un-distributes␣through␣composition",
  left = Map(A) *: Map(B) *: Rest,
  right = Map(A *: B) *: Rest
)

val mapPromotion = Equiv(
  name = "map␣promotion",
  left = Map(A) *: Join *: Rest,
  right = Join *: Map(Map(A)) *: Rest
)

val catamorphismPromotion = Equiv(
  name = "catamorphims␣promotion",
  left = Reduce(A) *: Join *: Rest,
  right = Reduce(A) *: Map(Reduce(A)) *: Rest
)

val hornersRule = Equiv(
  name = "Horner's␣rule",
  left = Reduce(A) *: Map(Reduce(B)) *: Tails *: Rest,
  right = Reduce(B *: A(D)) *: Rest,
  transform =
    s => neutralElement.get(s(B)).map(
          neutral => s + (D.n -> neutral))
)

val foldToScan = Equiv(
  name = "fold␣to␣scan",
  left = Map(Reduce(A)) *: Inits *: Rest,
  right = Scan(A) *: Rest
)
```

Correctnes of rewrite rules used In the remainder of this section, we will give (very informal) proofs that the rewrite rules used are correct. Correctness of three of them follows in a straightforward way from the properties of list catamorphisms.

```
-- map promotion: follows directly from the Promotion Theorem
forall f xs. map f . join = join . map (map f)

-- catamorphism promotion: similarly follows from the Promotion Theorem
forall. max . join = max . map max

-- map un-distributes through composition: another consequence of the PT
forall f g. map f . map g = map (f . g)
```

Correctness of the generalised Horner's rule was informally proved in Chapter 2. Its application reduces time complexity of the algorithm from cubic to quadratic.

Finally, the *fold to scan* rule simply folds the definition of *scan*. Application of this rule further reduces the time complexity from quadratic to linear.

The derivation of an efficient implementation for the MSS problem is well-known but non-trivial to implement. In this section, we showed that Pointfree Lift is robust enough to carry it out easily. We saw the design principles behind Pointfree Lift in the previous section; here we saw how this design allows the user to specify rewrite rules and derivations in a simple, declarative style.

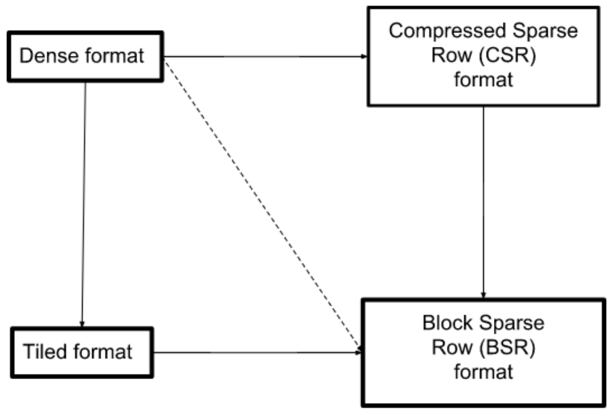
6.2 Rewriting sparse matrices

Another contribution we made was to show that we can derive a program for sparse matrix-vector multiplication by equational reasoning. We start with a program for dense MVmult, and then successively apply rewrite rules. In this way, we can obtain two programs, one for conversion from dense to the given sparse format, and another for actually performing MVmult on the sparse matrix. The composition of those two programs is extensionally equivalent to dense MVmult.

Original derivation for the CSR format As far as we are aware, the attempt to derive a program for sparse matrix-vector multiplication is original, and there is no literature on the subject. While not difficult conceptually, it provides a proof of concept that different versions of the same algorithm, operating on different data formats, can be derived from one another. Additionally, once we know a derivation for a specific data format, we could try to use this derivation to describe the data format and transform other programs to operate on this data format. Exploring this idea would be a possible continuation of this work.

Parallels between the CSR and BSR formats We consider two sparse formats, CSR and BSR, described in the Background section. We show a derivation for the CSR format in terms of Pointfree Lift rewrites. The BSR format, on the other hand, proved difficult to derive in this style. Nonetheless, we show an informal derivation of BSR (on paper, rather than in Pointfree Lift). Our derivation makes use of the insight that the derivation of BSR is parallel to the derivation of CSR, the difference being that rewrites are ‘lifted’. We will discuss this in details in a later section.

The flow of derivations is illustrated in the diagram below.


Dense format
Compressed Sparse Row (CSR) format
Tiled format
Block Sparse Row (BSR) format

6.2.1 Deriving CSR from the dense format

The Compressed Sparse Row format is characterised by only storing non-zero elements of rows. This means to need to able to recover the column position of included non-zero elements. For this reason, each non-zero entry is stored in a pair with its column position. If the type of a matrix in the dense format is $[[Float]_n]_m$, then the type after conversion to CSR would be $[[(Int, Float)]]_m$.

Our derivation of CSR proceeds as follows. As with the derivation of MSS, we mark the changes with underlined text.

```
 map (reduce plus . map (uncurry mult) . zip vector)
= { filter sum monoid }
map (reduce plus . filter (neq zero) . map (uncurry mult) . zip vector)
~
map (reduce plus . filter (neq zero) . map (uncurry mult) . zip vector)
= { map over zipped enumeration }
map (reduce plus . filter (neq zero) . map (uncurry (mult . access
vector)) . zip enumeration)
~
map (reduce plus . filter (neq zero) . map (uncurry (mult . access
vector)) . zip enumeration)
= { filter map mult absorber }
map (reduce plus . map (uncurry (mult . access vector)) . filter (neq
zero . snd) . zip enumeration)
~
map (reduce plus . map (uncurry (mult . access vector)) . filter (neq
zero . snd) . zip enumeration)
= { zip / unzip at (Int, Float) }
map (reduce plus . map (uncurry (mult . access vector)) . uncurry
zip . unzip . filter (neq zero . snd) . zip enumeration)
~
map (reduce plus . map (uncurry (mult . access vector)) . uncurry
zip . unzip . filter (neq zero . snd) . zip enumeration )
= { split at conversion / computation boundary}
map (reduce plus . map (uncurry (mult . access vector)) . uncurry
zip ) . map (unzip . filter (neq zero . snd) . zip enumeration )
```

The rewriting process can be specified in Pointfree Lift as follows:

```
@Test
def rewriteCsr(): Unit = {
  (Programs.denseMV :: Nil)
    .rewrite(filterSumMonoid)
    .rewrite(mapOverZippedEnumeration)
    .rewrite(filterMapMultAbsorber)
    .identityRewrite(zipUnzip(TInt, TFloat))
}
```

Correctness of rewrite rules used In the remainder of this section, we will explain why the individual rules we used are correct. Since the rules have the property that they do not change the meaning of the subprogram they transform, and the meaning of the program is determined by the meaning of its subparts, it follows that the sequence of rewrites preserves the meaning of the whole program.

Firstly, the *filterSumMonoid* rule states that whenever we fold over a list with an operation which has a neutral element, then we can filter this neutral element from the list. In the case of addition, the neutral element is zero, and so we filter out zeros from the list.

```
// forall f neutral xs. fold f neutral xs =
  fold f neutral . filter (/= neutral) $ xs
  val filterSumMonoid = Equiv(
    name = "filter␣sum␣monoid",
    left = Reduce(A),
    right = Reduce(A) *: Filter(Neq(B)),
    transform = s => neutralElement.get(s(A)).map(neutral => s + (B.
        n -> neutral))
  )
```

Secondly, the *mapOverZippedEnumeration* rule states that if we map a function over a zip of two lists, then we can instead zip one list with its enumeration, and then access elements of the other list by their index. For the matrix-vector multiplication in the CSR format, this amounts to pairing (non-zero) elements with their column indices and then accessing the corresponding vector element by its index.

```
// forall f xs ys. map (uncurry f) (zip xs ys) =
  map (uncurry(f . access xs)) . zip [0..] $ ys
  val mapOverZippedEnumeration = Equiv(
    name = "map␣over␣zipped␣enumeration",
    left = Map(Uncurry(A)) *: EZip(B) *: Rest,
    right = Map(Uncurry(A *: Access(B))) *: EZip(Enumeration) *:
        Rest
  )
```

Next, the *filterMapMultAbsorber* rule states that when we map a function $((*).g)$ over the list *xs*, then filter *zero* elements from the result, and then fold with the operation *f* whose neutral element is *zero*, then we can move the filter before the map. In this case, the map multiplies two numbers. For multiplication, zero is an absorber, meaning that multiplying with zero produces a zero. Moving the filter in front of the map filters all elements which are zero in their second component. The *map* can still produce zero values because of the first component, but this does not matter since the result of the map is consumed by $fold(f)(0)$, for which zero is a neutral element.

In the derivation of CSR, this corresponds moving the filter from the matrix-vector multiplication part to the conversion from dense to CSR part.

```
// forall f g xs. fold f 0 (filter (/= 0) (map ((*) . g) xs)) =
  fold f 0 . map ((*) . g) . filter ((/= 0) . snd) $ xs
  val filterMapMultAbsorber = Equiv(
    name = "filter␣map␣mult␣absorber",
    left = Fold(B)(Zero) *: Filter(Neq(Zero)) *: Map(Uncurry(Mult *:
        A)) *: Rest,
    right = Fold(B)(Zero) *: Map(Uncurry(Mult *: A)) *: Filter(Neq(
       Zero) *: Snd) *: Rest
  )
```

Finally, there is a *zipUnzip* rule, which changes the view of data. Unzipping takes a list of pairs and evaluates to a pair of lists. By composing *unzip*s, we can take a matrix of pairs and get a pair of matrices.

```
// forall. indentity = uncurry(zip) . unzip
object IdentityEquiv {
  def zipUnzip(a: Type, b: Type): IdentityEquiv =
    TList(TPair(a, b)) |- Uncurry(EZip) *: Unzip
}
```

These rules suffice to derive the CSR version of matrix-vector multiplication.

6.2.2 Deriving MVmult program for the BSR format

Like was mentioned before, we were unable to derive the program for matrix-vector multiplication for the BSR format using local rewrite rules of Pointfree Lift. Instead, we show on paper that the processes of performing matrix-vector multiplication for the CSR and BSR formats proceed in analogous steps, but for BSR, operations are in a way 'lifted' to the block level. Figure 6.1 illustrates this analogy with an example, and Figure 6.2 shows how subexpressions in the programs correspond to each other.

Since Table 6.2 refers to subexpressions of the programs, we provide Pointfree Lift programs for a composition of (1) conversion from the dense format, and (2) matrix-vector multiplication, for the CSR and BSR programs.

```
 csr = map (reduce (+) .  map (uncurry ((*) .  access vector))) .  map
(filter ((neq zero) .  snd) .  zip enumerate)

bsr = join .  map (reduce (zipWith plus) .  map (map (reduce plus .  map
(uncurry mult)) . uncurry (map . zip . access (split vector)))) .
map (filter (not . all (all (eq 0)) . snd)) . map (zip enumeration)
. map (map transpose . split . transpose) . split
```

Notice that, especially in the case of the BSR program, a downside of the pointfree style is revealed: longer programs become difficult to read.

6.3 Summary

This chapter showed how Pointfree Lift can be used to peform derivations. Two examples were used: the maximum segment sum problem, and matrix-vector multiplication for the CSR program.

CSR	BSR
(1) we have a matrix of numbers	(1) lift to matrix of matrices of numbers
$\begin{bmatrix}1&2&0&0\\1&0&0&0\\7&0&4&5\\0&8&3&6\end{bmatrix}\begin{bmatrix}5\\4\\6\\8\end{bmatrix}$	$\begin{bmatrix}\begin{matrix}1&2\\1&0\end{matrix} & \begin{matrix}0&0\\0&0\end{matrix}\\ \hline \begin{matrix}7&0\\0&8\end{matrix} & \begin{matrix}4&5\\3&6\end{matrix}\end{bmatrix}\begin{bmatrix}5\\4\\6\\8\end{bmatrix}$
(2) enumerate elements	(2) lift to enumeration of blocks
$\begin{bmatrix}(0,1)&(1,2)&(2,0)&(3,0)\\(0,1)&(1,0)&(2,0)&(3,0)\\(0,7)&(1,0)&(2,4)&(3,5)\\(0,0)&(1,8)&(2,3)&(3,6)\end{bmatrix}\begin{bmatrix}5\\4\\6\\8\end{bmatrix}$	$\begin{bmatrix}\left(0,\begin{bmatrix}1&2\\1&0\end{bmatrix}\right)&\left(1,\begin{bmatrix}0&0\\0&0\end{bmatrix}\right)\\\left(0,\begin{bmatrix}7&0\\0&8\end{bmatrix}\right)&\left(1,\begin{bmatrix}4&5\\3&6\end{bmatrix}\right)\end{bmatrix}\begin{bmatrix}5\\4\\6\\8\end{bmatrix}$
(3) filter zero elements from rows	(3) lift to filtering all-zeros blocks
$\begin{bmatrix}(0,1)&(1,2)&\\(0,1)&&\\(0,7)&(2,4)&(3,5)\\(1,8)&(2,3)&(3,6)\end{bmatrix}\begin{bmatrix}5\\4\\6\\8\end{bmatrix}$	$\begin{bmatrix}\left(0,\begin{bmatrix}1&2\\1&0\end{bmatrix}\right)&\\\left(0,\begin{bmatrix}7&0\\0&8\end{bmatrix}\right)&\left(1,\begin{bmatrix}4&5\\3&6\end{bmatrix}\right)\end{bmatrix}\begin{bmatrix}5\\4\\6\\8\end{bmatrix}$
(4) access vector element by index	(4) lift to accessing vector chunk by block index and zip the chunk with all rows of the block
$\begin{bmatrix}(5,1)&(4,2)&\\(5,1)&&\\(5,7)&(6,4)&(8,5)\\(4,8)&(6,3)&(6,6)\end{bmatrix}$	$\begin{bmatrix}\begin{bmatrix}\left(\begin{bmatrix}5&4\end{bmatrix},\begin{bmatrix}1&2\end{bmatrix}\right)\\\left(\begin{bmatrix}5&4\end{bmatrix},\begin{bmatrix}1&0\end{bmatrix}\right)\end{bmatrix}&\\\begin{bmatrix}\left(\begin{bmatrix}5&4\end{bmatrix},\begin{bmatrix}7&0\end{bmatrix}\right)\\\left(\begin{bmatrix}5&4\end{bmatrix},\begin{bmatrix}0&8\end{bmatrix}\right)\end{bmatrix}&\begin{bmatrix}\left(\begin{bmatrix}6&8\end{bmatrix},\begin{bmatrix}4&5\end{bmatrix}\right)\\\left(\begin{bmatrix}6&8\end{bmatrix},\begin{bmatrix}3&6\end{bmatrix}\right)\end{bmatrix}\end{bmatrix}$
(5) multiply a matrix element and a vector element	(5) lift to taking the dot product of a block row and a vector chunk
$\begin{bmatrix}5&8&\\5&&\\35&24&40\\32&18&36\end{bmatrix}$	$\begin{bmatrix}\begin{bmatrix}13\\5\end{bmatrix}&\\\begin{bmatrix}35\\32\end{bmatrix}&\begin{bmatrix}64\\66\end{bmatrix}\end{bmatrix}$
(6) take the sum of the row	(6) lift to taking the pointwise sum of each block-row
$\begin{bmatrix}13\\5\\99\\98\end{bmatrix}$	$\begin{bmatrix}\begin{bmatrix}13\\5\end{bmatrix}\\\begin{bmatrix}99\\98\end{bmatrix}\end{bmatrix}$
(7) done	(7) join the chunks corresponding to block rows
$\begin{bmatrix}13\\5\\99\\98\end{bmatrix}$	$\begin{bmatrix}13\\5\\99\\98\end{bmatrix}$

Figure 6.1: Comparison: performing matrix-vector multiplication for CSR and BSR formats

CSR	BSR
(1) we have a matrix of numbers	(1) lift to matrix of matrices of numbers
`id`	`map (map transpose . split . transpose) . split`
(2) enumerate elements	(2) lift to enumeration of blocks
`map (zip enumeration)`	`map (zip enumeration)`
(3) filter zero elements from rows	(3) lift to filtering all-zeros blocks
`map (filter ((neq 0) . snd))`	`map (filter (not . all (all (eq 0)) . snd))`
(4) access vector element by index	(4) lift to accessing vector chunk by block index and zip the chunk with all rows of the block
`map (first (access vector))`	`map (uncurry (map . zip . access (split vector)))`
(5) multiply a matrix element and a vector element	(5) lift to taking the dot product of a block row and a vector chunk
`map (uncurry mult)`	`map (reduce plus . map (uncurry mult))`
(6) take the sum of the row	(6) lift to taking the pointwise sum of each block-row
`reduce plus`	`map (reduce (zipWith plus))`
(7) done	(7) join the chunks corresponding to block rows
`id`	`join`

Figure 6.2: The processes of performing matrix vector multiplication for the CSR and BSR formats are analogous to each other: juxtaposition of program subexpression

Chapter 7

Evaluation: benchmarking performance

In this chapter, we take the design and implementation of Pointfree Lift, as well as the derivations we discussed, and perform an evaluation. When discussing derivations, we argued that the derived program was more efficient in some sense. Here we present benchmarks which confirm that, for each example, the derived implementation is as efficient as predicted.

7.1 Benchmarking sparse matrix formats

One primary application of Pointfree Lift is derivation of programs for sparse formats like CSR and BSR. In this section, we compare the running time of matrix-vector multiplication as the matrix format varies.

7.1.1 Benchmarking the CSR format

Table 7.1: Four different ways of performing MVmult in the first (CSR) benchmark

	dense	**CSR**
interpreted Pointfree Lift	dense	csr
Scala function	scalaDense	scalaCsr

In the first benchmark, we compare the dense format and the CSR format. We parametrise our experiments by what we call matrix sparsity. Matrix sparsity is the inverse of the percentage of non-zero entries in the matrix. Thus a matrix such that $\frac{1}{2}$ of its entries are non-zero has sparsity 2, one which has density $\frac{1}{3}$ has sparsity 3, and so on. In each experiment, we use a 1000×1000 matrix with randomly generated entries. Matrix sparsity ranges from 1 through 2, 4, 8 to 16. For each sparsity, we generate the

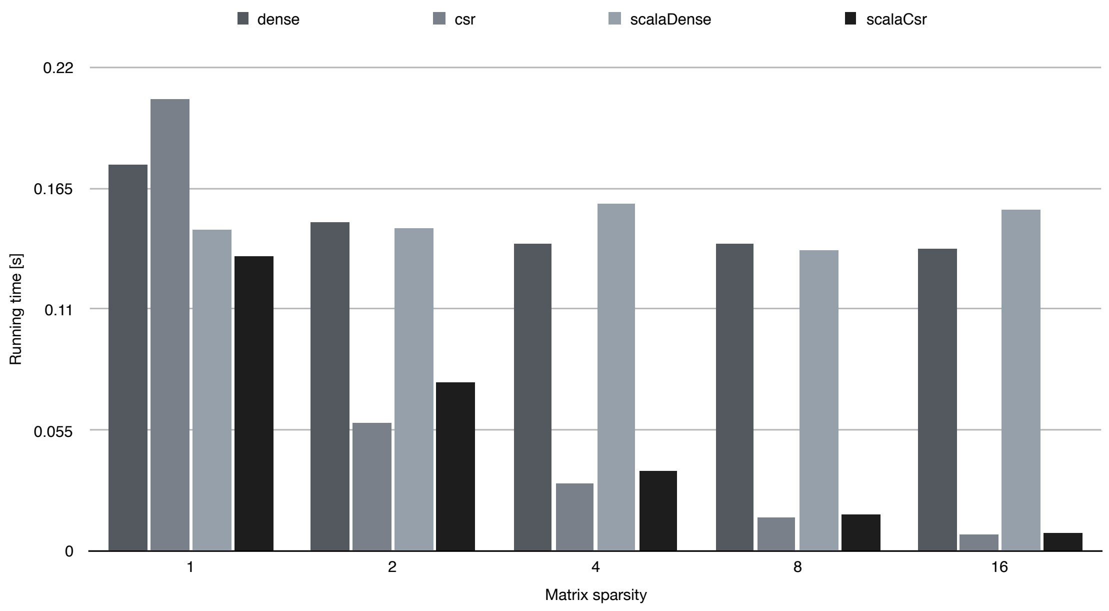


Figure 7.1: Results of the benchmark for CSR format. We perform five experiments, for matrix sparsity of 1, 2, 4, 8, 16. In each experiment, we use either dense or CSR format, and either Pointfree Lift program or a Scala function, which gives rise to four different ways of computation. For each matrix sparsity and each format, we run the experiment ten times and take the median. Time is given in seconds. In each experiment the matrix' size is 1000×1000.

	1	2	4	8	16
dense	0.175	0.149	0.139	0.139	0.137
csr	0.205	0.058	0.030	0.015	0.007
scalaDense	0.146	0.146	0.158	0.137	0.154
scalaCsr	0.133	0.077	0.036	0.016	0.008

Figure 7.2: Execution time in seconds. Benchmark for the CSR format.

matrix in two formats: dense and CSR. The vector of length 1000 is fixed between experiments.

For each generated matrix with a specific sparsity, we perform matrix-vector multiplication in four different ways. For each format, we either interpret a Pointfree Lift program or use a Scala function directly Table 7.1. In this way, not only can we compare the dense and CSR format, but we can also compare the performance of our Pointfree Lift interpreter with an ordinary Scala implementation.

For each matrix and each program, we run the program ten times and take the median. The numerical results are presented in Figure 7.2 and plotted in Figure 7.1. We can see from the chart that, in each experiment, the running time of the CSR program is inversely proportional to the matrix sparsity. This agrees with our predictions, since in the case of the CSR format, the number of arithmetic operations of addition and multiplication which need to be performed is proportional to the number of non-zero entries, whereas in the case of the dense format it is proportional to the matrix size.

A perhaps surprising insight from the experiments is that the interpreted programs are roughly as fast as Scala functions. This result contradicts the conventional wisdom that interpreters are slower than compiled programs (including compiled to bytecode, as is the case with Scala). We believe that this could be explained by the fact the interpreter performs a one-to-one mapping from Pointfree Lift abstract syntax to higher-order Scala functions. The only difference between a Scala function and an interpreted Pointfree Lift program is that values and lambda abstraction need to be packed and unpacked from *Value* constructors, but this cost appears to be insignificant.

7.1.2 Benchmarking the BSR format

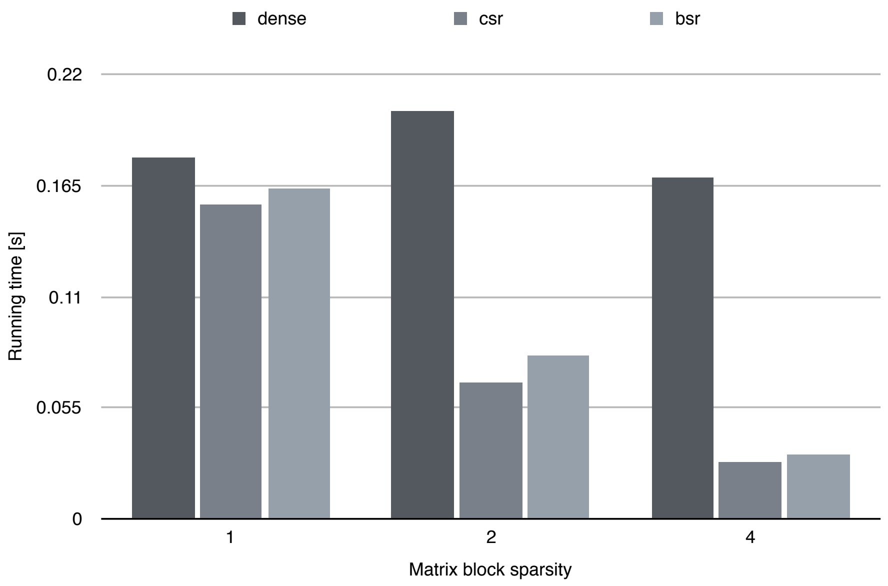


Figure 7.3: Results of the benchmark for the BSR format. We perform three experiments for matrices made of a 10×10 grid of 100×100 blocks with block density of 1, 2, and 4. In each experiment, we use three formats: dense, CSR, and BSR. For each block density and format, we perform ten time measurements and take the median. Time is given in seconds.

	1	2	4
dense	0.178	0.202	0.169
csr	0.155	0.067	0.028
bsr	0.163	0.080	0.031

Figure 7.4: Execution time in seconds. Benchmark for the BSR format.

In the second benchmark, we include the BSR format. Since it only makes sense to use the BSR format for matrices which have large blocks of zero entries, we generate 1000×1000 matrices made up of a 10×10 grid of 100×100 blocks. A block can either contain random numbers or only zero values. We parametrize our experiments by what we call block sparsity, defined analogously to sparsity in the previous section.

If on average one in x blocks of the matrix is non-zero, then we say that the block sparsity is x.

We run experiments for block sparsities 1, 2, and 4. Similarly to the CSR benchmark, for each experiment we generate a blocked matrix of a given block sparsity and then convert it to dense and CSR formats. For each block sparsity and each format, we do time measurements ten times and take the median. The results are presented in Figure 7.3 and Figure 7.4.

The results agree with what can be expected from analysing the running time of programs for different formats. For CSR and BSR programs, the number of steps is proportional to the number of non-zero blocks. For the dense format, it is proportional to the size of the matrix. Indeed, we can see that the running time for the dense format is constant, whereas the running time for the CSR and BSR formats is inversely proportional to the block density.

7.2 Benchmarking the maximum segment sum problem

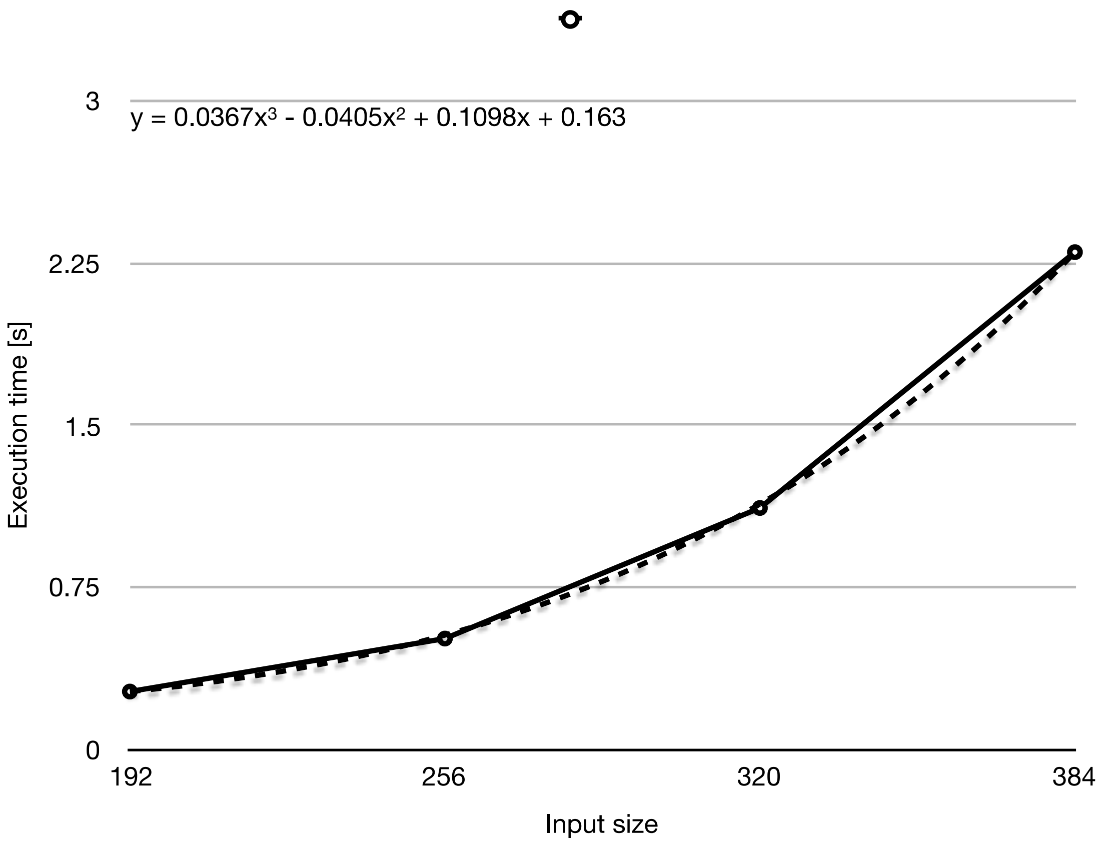


Figure 7.5: Results of the benchmark for the cubic $O(n^3)$ version of the MSS program. Execution time in seconds was measured ten times for each input size, and median taken. A line of best fit is juxtaposed.

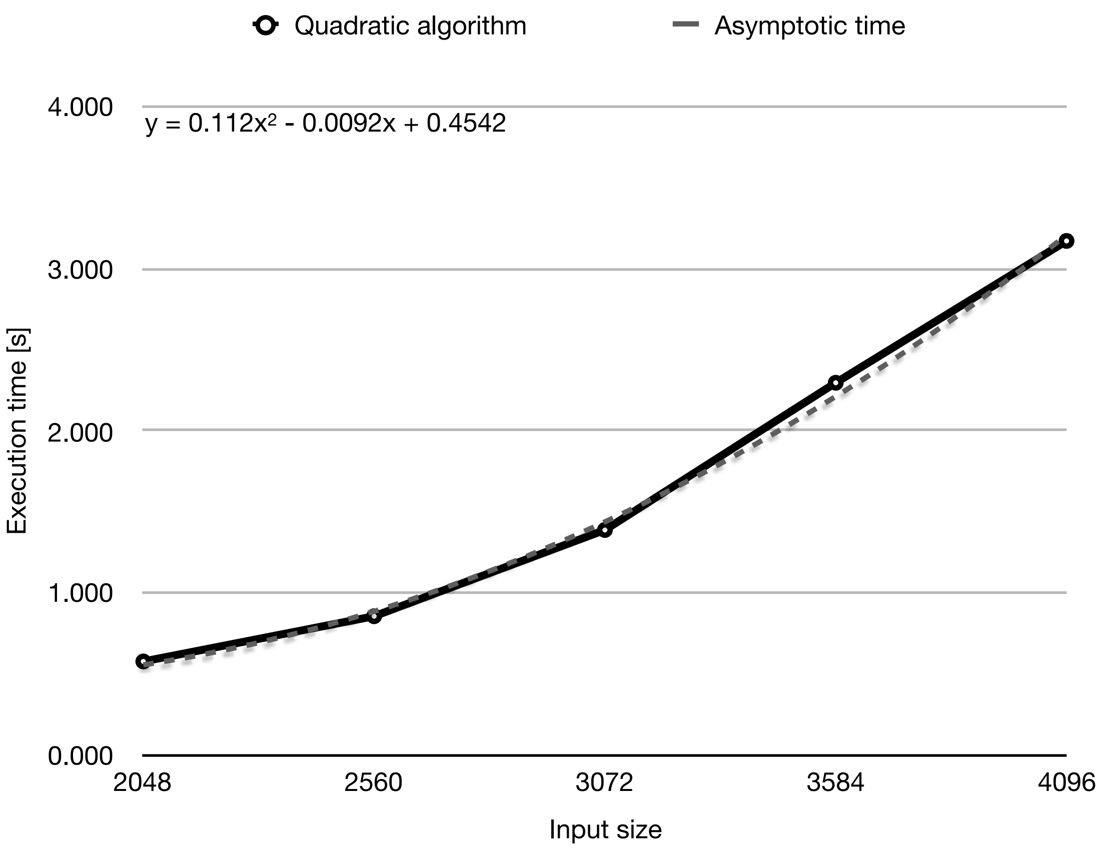


Figure 7.6: Results of the benchmark for the quadratic $O(n^2)$ version of the MSS program. Execution time in seconds was measured ten times for each input size, and median taken. A line of best fit is juxtaposed.

In Section 6.1 we started with an inefficient but comprehensible program for the maximum segment sum problem and then showed how Pointfree Lift could be used to express the derivation of an efficient program for the MSS problem. We argued that the original program has cubic running time in the length of the input list, whereas the derived program runs in linear time. In this section, we support those claims with measurements. As well as the original cubic program and the resulting linear program, we also include an intermediate program where the running time is reduced by applying the generalised Horner's rule, but before it is further reduced by replacing a fold with a scan, and which therefore runs in quadratic time.

Since the three different version have running time which differs by orders of magnitude, experiments were run on different input sizes, and are presented in different figures: the cubic version in Figure 7.5, the quadratic version in Figure 7.6, and the linear version in Figure 7.7. In each case, it is possible to find a line of best fit which conforms to the expected asymptotic running time.

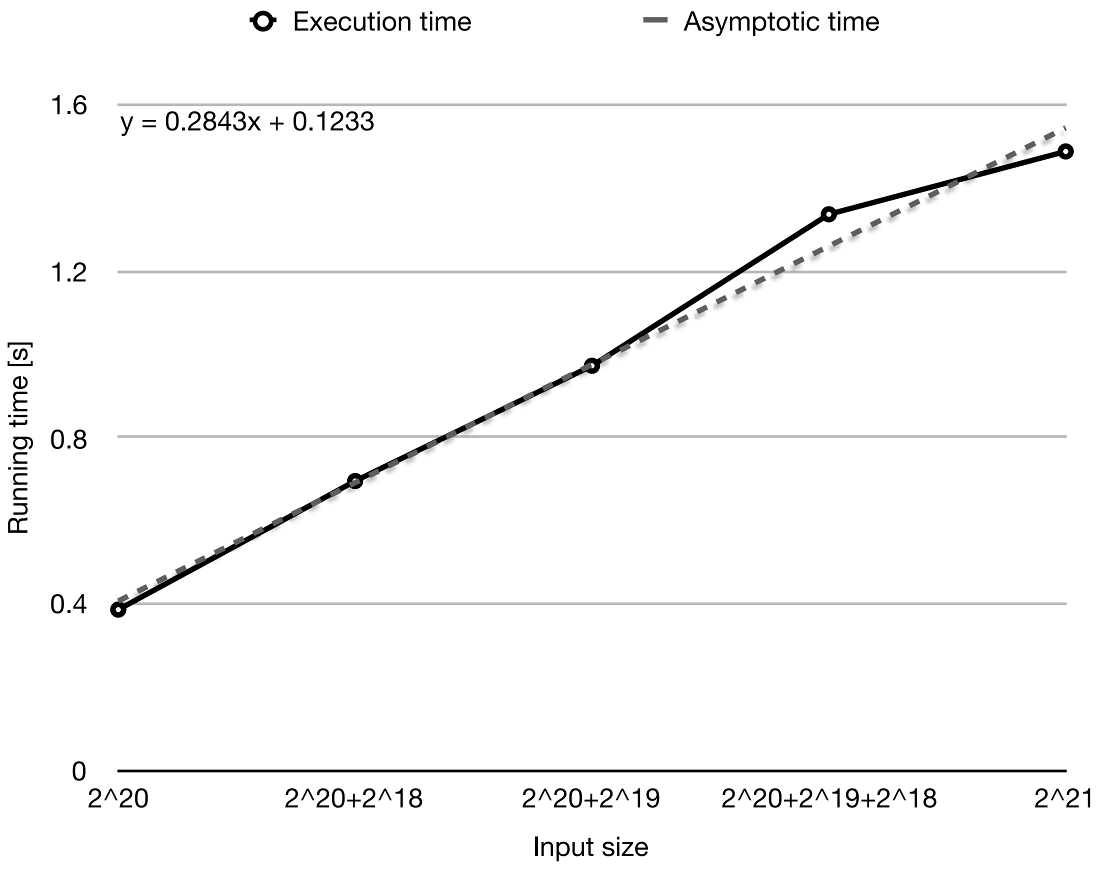


Figure 7.7: Results of the benchmark for the linear $O(n)$ version of the MSS program. Execution time in seconds was measured ten times for each input size, and median taken. A line of best fit is juxtaposed.

Chapter 8

Conclusion

The main deliverable of this project is a system for rewriting and transforming point-free functional programs, Pointfree Lift. The implementation is complete to the extent that it allows for performing medium-sized derivations, as demonstrated by the examples, and it supports an interactive user interface. Nonetheless, there exist program transformations which cannot be expressed in Pointfree Lift.

Pointfree Lift has functionality which one could expect from a functional language, that is, it can be type checked and executed (interpreted). Additionally, it is designed to facilitate moderately complex transformations. It is by no means a complete language, and due to its lack of support for general recursion, there are computations it cannot express. However, its expressive power is based on built-in primitives, some of which are higher-order. By adding more primitives, which is straightforward, Pointfree Lift could express a broad range of programs.

The unique contribution of this project is not only the implementation but also two proofs of concepts it contains. Firstly, the example of the derivation of matrix-vector multiplication for the CSR format demonstrated that a program could be transformed to accept a different input format. While this project did not deliver practical applications of this insight, the idea can be explored in the second part; for example, a program could be described in terms of the data format it accepts.

Secondly, by implementing a functional user interface, the project demonstrates that the user can be involved in the process of resolving the choice of program transformations. This result could be applied to the Lift project, for example, where the user's insight could supplement the automatic search of efficient OpenCL kernels.

Finally, by providing a graphical user interface for program transformation, the project has educational value. Many program derivations have been published, but their accessibility to a non-specialist programmer might have been limited by their abstract nature. This project delivers a tool which allows the user to perform a derivation step by step, with assistance from the system, thus making the process more concrete.

8.1 Critical analysis

The project departed from its original goals in the sense that program transformation was done in a dedicated system which was built for this project, rather than in the Lift compiler. The change was necessitated by inadequate flexibility of Lift; unfortunately, it means that the contributions of this project cannot be readily used in Lift. However, working within a custom system meant greater freedom and allowed the project to go beyond sparse matrix formats into general program derivation.

While Pointfree Lift is powerful enough to perform a range of derivations, perform benchmarking for the obtained program, and expose an interactive user interface, there is scope for improvement. Program derivations consist of a series of applications of equational rewrite rules; in comparison, sophisticated program transformation systems like Path [27] make use of complex tools, for example, a dedicated logic. Additionally, correctness of program transformation is only guaranteed as long as the rewrite rules are correct. A more mature tool would require that rewrite rules themselves can be proven, perhaps in a proof language.

This project does not aim at providing a tool which would be usable for real-world software development. Attempts at developing such tools were made but did not gain traction in the industry [27]. The research area of algebraic program derivations has received less interest in recent decades. In this light, Pointfree Lift could be used as a tool to present historical results to the modern audience in an accessible way. Pointfree Lift's interactive mode might be integrated into the Lift project as a tool for selecting optimisations.

8.2 Future work and second part of the project

The plan for the second part of the project is to address the limitations of Pointfree Lift, in particular, to support transformations which deal with lambda abstractions and variable binding. Although pointfree style is expressive, it is unfeasible for larger programs. On the other hand, program transformations involving terms with variable binding or some kind of lambda-abstraction can be challenging to work with as they do not fit the classic algebraic datatypes.

One method to reason about systems with binding is the type-and-scope-safe representation of Allais, Chapman, McBride and McKinna [6]. Introduced last year, it gives a straightforward way to model such systems and carry out machine-assisted proofs about them. For example, it was used earlier this year to verify the triangulation proof method [19].

The second part of the project will explore the effectiveness type-and-scope-safe methods for verifying transformations used in compiling of functional languages. This will probably be done in the dependently-typed Agda language [1].

A paper by Wang and Nadathur [28] carries out a similar exploration for techniques based on higher-order abstract syntax: they examine typed closure conversion, code hoisting, and transformation to continuation-passing style. Those two papers will be taken as a starting point, and the ideas from them will be explored in the context of the first part.

Bibliography

[1] The Agda programming language.

[2] The Coq proof assistant.

[3] Glasgow Haskell Compiler.

[4] Idris — a language with dependent types.

[5] Scurses — Scala terminal user interface library.

[6] Guillaume Allais, James Chapman, Conor McBride, and James McKinna. Type-and-scope safe programs and their proofs. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 195–207. ACM, 2017.

[7] Richard Bird and Oege De Moor. The algebra of programming. In *NATO ASI DPD*, pages 167–203, 1996.

[8] R.S. Bird and John Hughes. The alpha-beta algorithm: An exercise in program transformation. *Information Processing Letters*, 24(1):53 – 57, 1987.

[9] Wikipedia contributors. Minimax — Wikipedia, the free encyclopedia.

[10] Stephen Diehl. PHOAS — what i wish i knew when learning haskell.

[11] Edsger Wybe Dijkstra. Notes on structured programming, 1970.

[12] Jeremy Gibbons. An introduction to the Bird-Meertens formalism. 1994.

[13] Jeremy Gibbons. A pointless derivation of radix sort. *Journal of Functional Programming*, 9(3):339–346, 1999.

[14] Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 12. ACM, 2011.

[15] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[16] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell workshop*, volume 1, pages 203–233, 2001.

[17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):77, 2017.

[18] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[19] Craig McLaughlin, Ian Stark, James McKinna, et al. Triangulating context lemmas. 2017.

[20] Lambert Meenens. Towards programming as a mathematical activity.

[21] Lambert Meertens. Paramorphisms. *Formal aspects of computing*, 4(5):413–424, 1992.

[22] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[23] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[24] Stack Overflow. What is the difference between Scala's case class and class?

[25] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[26] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015.

[27] Mark Anders Tullsen and Paul Hudak. *Path, a program transformation system for Haskell.* Yale University, 2002.

[28] Yuting Wang and Gopalan Nadathur. A higher-order abstract syntax approach to verified transformations on functional programs. In *European Symposium on Programming Languages and Systems*, pages 752–779. Springer, 2016.

[29] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ACM SIGPLAN Notices*, volume 38, pages 249–262. ACM, 2003.